DIPLOMARBEIT

Variable Neighborhood Search for the Partition Graph Coloring Problem

ausgeführt unter der Leitung von

Mag. rer. soc. oec. Dipl.-Ing. Dr. techn. Martin Gruber

eingereicht an der

Höheren Technischen Bundeslehr- und Versuchsanstalt Wien Ottakring

Abteilung für Informationstechnologie und Elektronik

von

Lorenz Leutgeb und **Moritz Wanzenböck**

Wien, am 8. Mai 2013

HÖHERE TECHNISCHE BUNDESLEHRANSTALT WIEN 16
Höhere Abteilung für Informationstechnologie
Ausbildungsschwerpunkt *Netzwerktechnik*

Projektnummer: 16 IT 12 08		Wien, am 11. Oktober 2012

Ansuchen um Genehmigung einer Aufgabenstellung für die

DIPLOMARBEIT

Jahrgang: **5AHITN** Schuljahr: **2012/13**

Thema: **VNS für das Partition Coloring Problem**

Aufgabenstellung: Für eine effiziente Ausnutzung eines auf Basis von Wavelength Division Multiplexing (WDM) operierenden Glasfasernetzes ist die Zuordnung von Wellenlängen zu einzelnen Verbindungen ein komplexes, NP-schwieriges Optimierungsproblem, da über eine beliebige Teilstrecke des Netzwerkes immer nur dann Datenströme gleichzeitig transportiert werden können, wenn sie unterschiedliche Wellenlängen nutzen. Diese Zuweisung von Wellenlängen zu Verbindungen kann graphentheoretisch durch das Partition Coloring Problem beschrieben werden.
Ausgehend von aktuellen wissenschaftlichen Publikationen zu diesem Themenbereich ist es Gegenstand dieser Arbeit, eine variable Nachbarschaftssuche (engl. Variable Neighborhood Search, VNS) auf Basis neuer, für dieses Optimierungsproblem zugeschnittener Nachbarschaften, zu entwerfen und zu implementieren.

Anzahl der Beiblätter: 0

Zuordnung zu den Fachgebieten: • Netzwerkmanagement (NTMA)

Kandidatin(nen)/ Kandidat(en): Lorenz LEUTGEB .....

Moritz WANZENBÖCK .....

Betreuerin(nen)/ Betreuer: Prof. Mag. Dipl.-Ing. Dr. Martin GRUBER .....

Höhere Technische Bundeslehranstalt 6 WIEN 16.

..... zKg. .....

Dir. Dipl.-Ing. Peter Bachmair AV Mag. Michael Brunn

Als Diplomarbeit zugelassen

.....

LSI Dipl.-Ing. Judith WESSELY-KIRSCHKE

Eigenständigkeitserklärung

Hiermit erklären wir, Lorenz Leutgeb und Moritz Wanzenböck, an Eides statt, dass wir die vorliegende Diplomarbeit selbstständig und nur mit den angegebenen Hilfsmitteln verfasst haben.

Wir versichern ausdrücklich, dass sämtliche in der Arbeit verwendeten fremden Quellen, auch aus dem Internet, als solche kenntlich gemacht wurden. Insbesondere bestätigen wir, dass ausnahmslos sowohl bei wörtlich übernommenen Aussagen bzw. unverändert übernommenen Tabellen, Grafiken u. Ä. (Zitaten) als auch bei in eigenen Worten wiedergegebenen Aussagen bzw. von uns abgewandelten Tabellen, Grafiken u. Ä. anderer Autorinnen und Autoren (indirektes Zitieren) die entsprechende Quelle angegeben wurde.

Uns ist bewusst, dass Verstöße gegen die Grundsätze der Selbstständigkeit als Täuschung betrachtet und entsprechend der Gesetzgebung geahndet werden.

Die vorliegende Arbeit wurde in gleicher oder ähnlicher Form bisher bei keiner anderen Institution eingereicht.

Lorenz Leutgeb	Moritz Wanzenböck

Abstract

The internet as global communication platform evolved to a pillar of modern society. Increasing demand to exchange data keeps putting a fundamental question to internet service providers: How to fulfill the consumers requests? When expanding communication networks one can either build new and expand exissting networks, which means being forced to install additional cables and expensive hardware, or adapt the network to utilize existing devices and lines more efficiently.

This is where the aspect of optimizing networks comes to mind. In this work we will show in detail how to model and solve the real world problem of assigning wavelengths to communication lines in fibre networks relying on *Wavelength Division Multiplexing* (WDM) using the *Partition Graph Coloring Problem* (PCP). We will present a solution using metaheuristic approaches, namely a *Variable Neighborhood Search* (VNS).

Zusammenfassung

Das Internet, in seiner Funktion als globale Kommunikationsplattform, entwickelte sich zu einem Stützpfeiler der modernen Gesellschaft. Das ständig steigendene Datenaufkommen stellt Internet Service Provider immer wieder vor die Frage: Wie kann man die Kundenwünsche erfüllen? Um ein Netzwerk leistungsfähiger zu machen, kann man entweder bestehende Infrastruktur ausbauen und erweitern, was natürlich auch neue, kostspielige Anschaffungen erfordert, oder das bestehende Netz so anpassen, dass es bestehende Ressourcen effizienter nützt.

Effizientere Ressourcennutzung ist eng verbunden mit Optimierung. In dieser Arbeit werden wir im Detail zeigen, wie man das Problem der Wellenlängenzuweisung in einem Glasfasernetzwerk durch das Partition Graph Coloring Problem darstellen und lösen kann. Wir werden eine ein neuartiges Lösungsverfahren für dieses Problem präsentieren, welches eine Variable Nachbarschaftssuche als Ausgangspunkt nimmt.

Inhaltsverzeichnis

1. Problemstellung

In der heutigen Zeit, in der das weltweite Kommunikationsbedürfnis in ungeahnte Höhen steigt und, bis auf Weiteres, kein Ende des Anstiegs in Sicht ist, werden Computernetzwerke, besonders solche, die ganze Kontinente umspannen, immer stärker beansprucht. Diese Entwicklung ist vor allem auf das erhöhte und sich grundlegend ändernde Konsumverhalten der Menschen in der heutigen Zeit zurückzuführen. Immer mehr Informationen in Netzwerken werden nicht in Form von Texten sondern, aufwendig aufbereitet, als Bild und Ton zum Endkunden bzw. Konsumenten übertragen. Dieses Bild- und Tonmaterial erzeugt natürlich eine höhere Auslastung von Datenleitungen, was, im Zusammenspiel mit der gestiegenen Anzahl an Informationsbedürfnissen insgesamt, zu einer Verknappung der Ressourcen in Computernetzen führt.

Um diesen neuen Gegebenheiten gerecht zu werden gibt es zwei Möglichkeiten: entweder ein Ausbau und Neubau von Netzen, oder eine bessere Ausnutzung von bestehenden Strukturen. Aus- und Neubau bringen allerdings neue Probleme mit sich. Erstens ist Netzwerkinfrastruktur teuer, besonders wenn sie über mehrere Jahrzehnte hinweg halten und ihren Dienst verrichten soll ohne eine ständige Wartung mit sich zu ziehen. Zweitens ist es manchmal aus verschiedenen Gründen gar nicht möglich, ein Netz weiter auszubauen. Sollte sich eine bestehende Infrastruktur bereits in ihrer letzten Ausbaustufe befinden, ist es überhaupt notwendig ein gänzlich neues Netz aufzubauen, was wiederum mit noch höheren Kosten verbunden ist. Zu guter Letzt sorgt der Betrieb von zusätzlicher Infrastruktur auch für größeren Strombedarf, was sich nicht nur negativ auf die Kosten auswirkt, sondern durchaus auch eine Belastung für die Umwelt darstellt.

Die zweite Möglichkeit, die bessere Ausnutzung von bestehenden Netzen, kann zwar die letztendliche Notwendigkeit von neuen Netzen nicht verhindern, den Zeitpunkt des Eintretens dieser letzten Option aber hinauszögern. Durch eine optimierte Nutzung können bereits vorhandene Netze über einen längeren Zeitraum genutzt werden ohne einen Engpass zu bilden. Wie diese

Optimierungen im konkreten Fall aussehen hängt unter anderem von der Größe und der Art des Netzes, aber auch von der Art des Kommunikationsbedürfnisses ab. In dieser Arbeit konzentrieren wir uns auf optische Netze und eine genau typisierte Kommunikation.

In rein optischen Netzen, also Netzwerken, in denen Information als Lichtimpulse ausschließlich über optische Leiter wie Glasfaser übertragen wird und das Signal auf der ganzen Strecke zu keinem Zeitpunkt in ein elektrisches zurück gewandelt wird, kann *Wavelength Division Multiplexing* (WDM) zur parallelen Übertragung mehrerer Datenströme über einzelne Leitungen betrieben werden. Die Idee hinter WDM ist es, für zwei verschiedene Datenströme unterschiedliche Farben des Lichts zu verwenden. Der englische Name *Wavelength Division* (deutsch: „Wellenlängen Unterteilung“) kommt von der Tatsache, dass unterschiedliche Farben des Lichts verschiedene, eindeutige Wellenlängen haben. Da sich solche Farben bei entsprechend großer Wellenlängendifferenz nicht gegenseitig beeinflussen, können viele Farben gleichzeitig mithilfe eines einzelnen Lichtwellenleiter übertragen werden und am anderen Ende des Leiters wieder problemlos auseinander gehalten werden. Dies funktioniert sogar so gut, dass Lichtwellen, deren Wellenlänge sich nur um wenige Nanometer unterscheidet, immer noch eindeutig zuordenbar sind.

Des Weiteren wird oft angestrebt, einem einzelnen Datenstrom auf seinem gesamten Weg durch das Netzwerk dieselbe Farbe zuzuweisen. Dies bietet vor allem Vorteile im Bereich der Einfachheit der Implementierung, da optische *Switches* (deutsch sinngemäß „Schaltorgan“, „Vermittlung“) innerhalb des Netzwerkes keine Umwandlung von einer Farbe in eine andere vornehmen müssen. Hinzu kommt ein geringerer Stromverbrauch im Netzwerk, da bei ebendieser Umwandlung auch Energie verloren geht.

Zu guter Letzt ist noch die spezielle Art des Kommunikationsbedürnisses zu beachten. Anders als zum Beispiel im Internet, wo die Kommunikation für einen Netzbetreiber unberechen- und vor allem unverhersehbar abläuft, gehen wir hier von jenem Fall aus, in dem der größte Teil der Kommunikationswünsche bereits von vornherein bekannt ist. Vorstellbar wäre solch ein Fall zum Beispiel bei einer gemieteten Standleitung eines Unternehmens zu dessen entlegenenen Zweigstelle. Auch die Kommunikationsbedürfnisse zwischen Internetprovidern (z. B. Durchleitung von Daten) können im großen Maßstab als ein über einige Wochen oder Monate hinweg relativ konstantes Datenaufkommen betrachtet werden. Obwohl dies nun eine starke

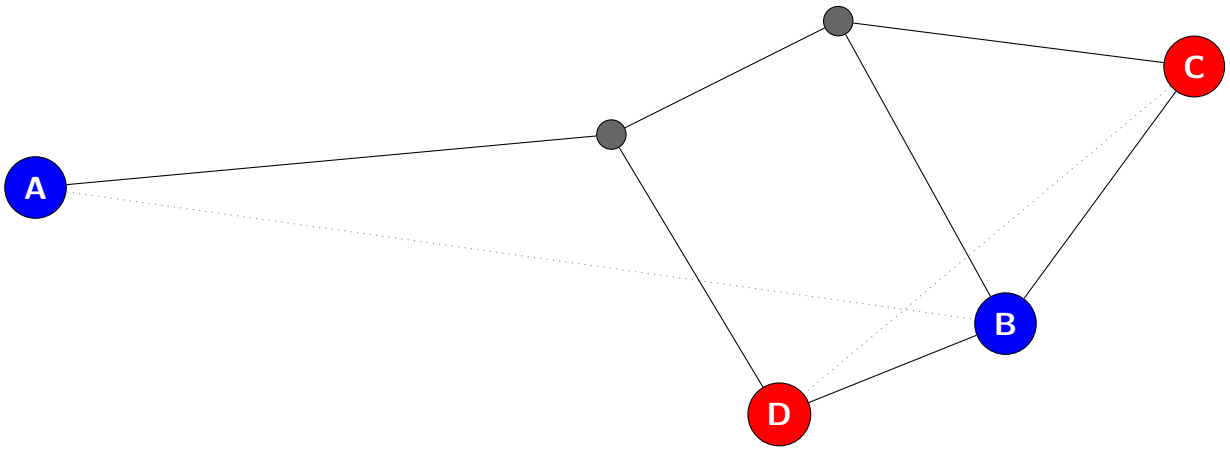
Einschränkung des Einsatzgebietes vermuten lässt, ist der Nutzen dennoch nicht zu unterschätzen. Wenn zum Beispiel der Aufwand für den Betrieb solcher Standleitungen minimiert werden kann, bleiben mehr Ressourcen für die Erfüllung der vorhin erwähnten unvorhersehbaren Kommunikationsbedürfnisse im Netzwerk frei.

1.1. Partition Graph Coloring Problem

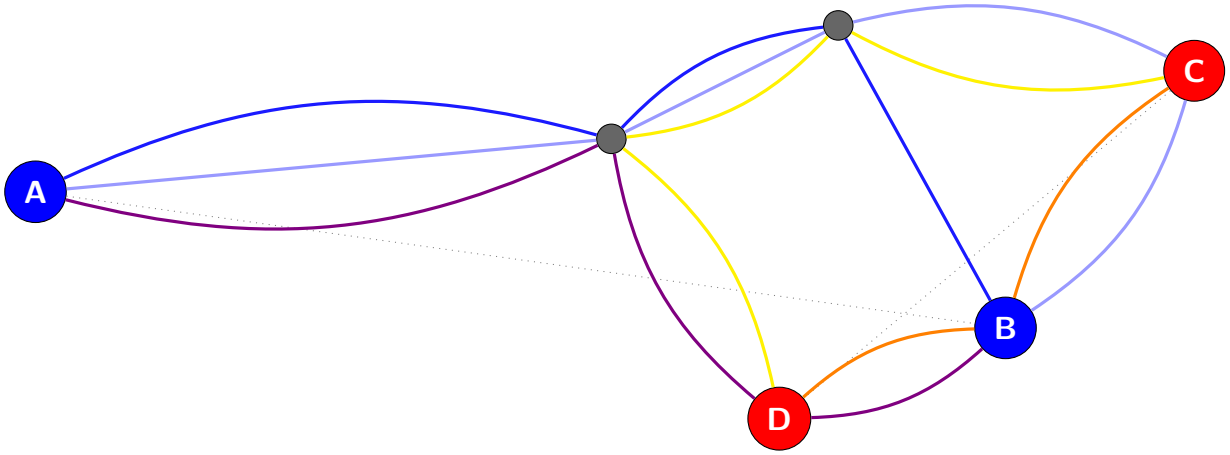
Mit Hilfe des *Partition Graph Coloring Problems* (PCP) wird versucht, diesen sehr spezifischen Anwendungsfall mathematisch zu beschreiben. Um das Problem zu lösen wird ein Graph aufgebaut, wodurch dann eine möglichst gute, wenn nicht sogar optimale Lösung für das Problem berechnet werden kann. Ein Graph ist ein mathematisches Konstrukt, das aus Knoten besteht, die durch Kanten verbunden werden. Ein Knoten A kann mit Knoten B durch so eine Kante verbunden werden, Knoten A muss aber zum Beispiel keine Verbindung mit Knoten C besitzen. Wie schon von Li, Simha und Williamsburg (2000) gezeigt wurde, handelt es sich bei dem PCP um ein NP-schwieriges Problem, das heißt es kann auf einem Computer wie man ihn heute kennt in polynomieller Laufzeit keine Lösung gefunden werden, welche für die konkrete Problemstellung optimal ist. Polynomielle Laufzeit wird als Grenze zwischen Problemen angesehen, welche in vertretbarer Laufzeit noch zu berechnen sind und Problemen, für die der Lösungsaufwand in keinem Verhältnis zum Ergebnis steht, namentlich, deren Laufzeit ein exponentielles Verhalten an den Tag legt.

Um das Problem besser lösen zu können, wird ein einheitliches Ziel und eine einheitliche Eingabe benötigt. Dazu wird die Problemstellung nun in einem ersten Schritt auf folgende Punkte fixiert:

- Ein Netzwerk mit seinen möglichen Kommunikationsverbindungen zwischen den einzelnen Netzwerkknoten ist gegeben.
- Alle Kommunikationsbedürfnisse, welche aus einem Start- und einem Endpunkt bestehen, sind bekannt.
- Für jedes Kommunikationsbedürfnis gibt es einen oder mehrere Wege, sprich Routen, zwischen Start- und Endpunkt durch das Netzwerk.

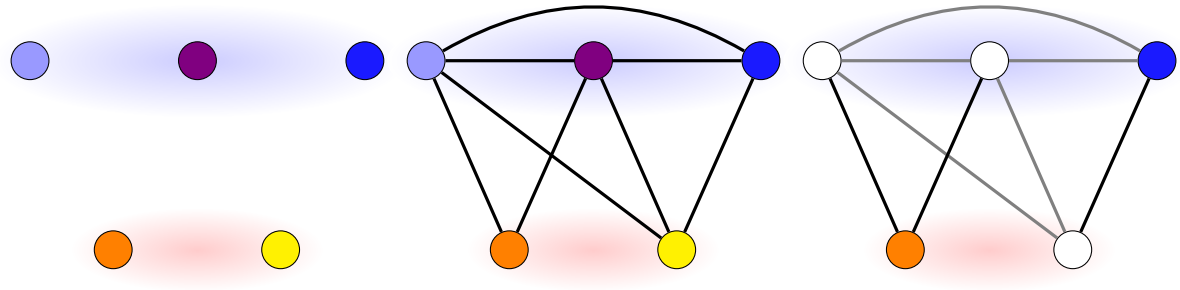


(a) Repräsentatives Beispiel für ein Glasfasernetzwerk innerhalb Österreichs. Es gibt zwei Kommunikationswünsche und zwar zwischen A und B sowie zwischen C und D.



(b) Beispiel für verschiedene Routen zwischen insgesamt vier Kommunikationspartnern. Verschiedene Möglichkeiten der Wegfindung zwischen A und B werden in Hell- und Dunkelblau sowie Violett dargestellt. Die alternativen Routen zwischen C und D sind gelb und orange eingefärbt.

Abbildung 1.1.: Ein Beispiel für eine Probleminstanz des Partition Graph Coloring Problems



(a) Der aus Abbildung 1.1b generierte Konfliktgraph. Jede Route wird als Knoten repräsentiert. Die verwendete Farbe stimmt mit den für die Routen verwendeten Farben überein. Die Partitionierung wird durch das Oval im Hintergrund der Knoten angedeutet.

(b) Der Konfliktgraph wird um die Beziehungen der Knoten untereinander erweitert. Jede Route, welche eine Teilstrecke mit einer anderen Route gemeinsam hat, wird mit dieser Route verbunden. Die Kanten zwischen den Knoten stellen diese Beziehung dar.

(c) Eine mögliche Lösung des PCP. Von jeder Partition muss nur ein Knoten ausgewählt werden, alle anderen werden ausgeblendet. Die Auswahl für die Strecke zwischen A und B hat kein gemeinsames Teilstück mit der für die Strecke $\overline{CD}$ getroffenen Auswahl. Daher kann ein und dieselbe Wellenlänge für beide Übertragungen verwendet werden.

Abbildung 1.2.: Aufbau und Lösung eines Problem-Graphen des Partition Graph Coloring Problems

- Sollte eine Route einen Teil des Weges auf der selben Strecke zurücklegen wie eine andere Route, müssen sie unterschiedliche Farben verwenden um sich nicht gegenseitig zu stören.
- Jede Route verwendet genau eine Farbe und zwar für die gesamte Strecke.
- Die Lösung besteht aus genau einer Route für jedes Kommunikationspaar und genau einer dazugehörigen Farbe.

Aus diesen Bedingungen kann ein Graph abgeleitet werden, der genau jene Bedürfnisse erfüllt, um ein einfaches Lösen des Problems zu ermöglichen. Der Konfliktgraph, auf dem dann die Lösung berechnet wird, besteht aus Partitionen, welche eine Menge von zusammengehörigen Knoten darstellt und Kanten, welche die einzelnen Knoten verbinden. Ein Kommunikationsbedürfnis zwischen zwei Partnern wird als eine Menge von Knoten gesehen. Die Knoten repräsentieren dabei je eine Route zwischen den beiden Partnern. Steht also nur eine Route zwischen zwei Kommunikationsendpunkten zur Auswahl, gibt es nur einen Knoten in dieser Menge. Diese Menge an Knoten wird Partition genannt und ist damit namensgebend für das PCP. Dieser Schritt wird in Abbildung 1.2a veranschaulicht.

Die Kanten bilden nun die Beziehungen zwischen verschiedenen Routen ab. Teilen sich zwei Routen das selbe Teilstück im optischen Netzwerk, werden sie mit einer Kante verbunden. Zu beachten ist, dass dies nicht nur für Knoten in der selben Partition gilt sondern auch für Knoten aus unterschiedlichen Partitionen. Verwendet also Route X der Kommunikation ($\overline{AB}$) ein Stück der Strecke, das auch von der Route Y der Kommunikation ($\overline{CD}$) verwendet wird, werden diese beiden Routen, welche ja jeweils als Knoten repräsentiert werden, im Konfliktgraphen mit Hilfe einer Kante verbunden. Außerdem ist noch zu beachten, dass für jedes in dieser Art verbundene Routenpaar nur eine Kante verwendet wird. Teilen sich Routen X und Y mehr als ein Teilstück, so werden sie trotzdem nur einmal per Kante verbunden. Die Vervollständigung des Konfliktgraphen wird in Abbildung 1.2b dargestellt.

Nachdem der Konfliktgraph generiert wurde, kann eine Lösung für das PCP berechnet werden. Für das durch die Partition symbolisierte Kommunikationsbedürfnis muss genau eine Route ausgewählt werden und diese Knoten müssen dann so eingefärbt werden, dass sich keine zwei mit einer Kante verbunden Knoten die gleiche Farbe teilen. Der letztere Teil, also das Einfärben von Knoten in Abhängigkeit der Farben der Nachbarknoten, ist als klassisches *Graph Coloring* (GC) bekannt. Es kommt zum Beispiel bei der Einfärbung von politischen Landkarten

zum Einsatz, wo benachbarte Länder nicht die selbe Farbe haben dürfen. Was das PCP vom klassichen GC unterscheidet, ist die zusätzliche Aufgabe, nur einen Knoten aus einer Partition als Repräsentanten auszuwählen und die anderen quasi auszublenden. Ein Beispiel einer solchen Lösung des Problems ist in Abbildung 1.2c zu finden.

Die Schwierigkeit des Problems liegt in dem Zusammenspiel von verschiedenen Möglichkeiten, ein und den selben Graphen einzufärben und andererseits mit einer anderen Auswahl an Knoten gleich den gesamten Lösungsgraphen zu verändern. Das Kriterium für die Güte einer Lösung ist die Anzahl der verwendeten Farben. Im Originalproblem entspricht das der Anzahl an Wellenlängen, die für die vorab bekannte Kommunikation im Netzwerk benötigt werden. Um sich nur auf diese Anzahl konzentrieren zu können, wird angenommen, dass jede Route zwischen zwei Kommunikationspartnern gleich gut ist. Eventuelle Umwege (Routen länger als die kürzeste mögliche) werden also im PCP ignoriert.

Außerdem geht man bei der Lösung des PCP von einem homogenen Netzwerk aus, in dem jede Leitung jede Wellenlänge unterstützt. In der Realität kann es durchaus vorkommen, dass ein Lichtwellenleiter, welcher aus einem chemisch minimal anders zusammengesetzten Stoff besteht als seine Gegenstücke in anderen Teilen des Netzwerkes, für bestimmte Wellenlängen ungünstige Eigenschaften hat. Dieser Aspekt wird beim Lösen des PCP nicht beachtet, jede Strecke beziehungsweise jede Route unterstützt alle Wellenlängen. Außerdem wird angenommen, dass jeder Leiter eine unbegrenzte Anzahl an Wellenlängen unterstützt. Reale Leiter können nur eine begrenzte Anzahl an Wellenlängen unterstützen, bis ihr Spektrum ausgeschöpft ist.

1.1.1. Formale Definition

Gegeben sei ein ungerichteter Graph $G = (V,\ E)$, wobei V die Menge der Knoten des Graphen G und E die Menge der Kanten (2-Tupel der Form $(s,\ t)$, deren Werte jeweils Start- und Endknoten der Kante bezeichnen) die V in G verbinden, darstellen. Außerdem seien $V_1,\ V_2,\ V_3,\ \ldots,$ V_k disjunkte Teilmengen von V, $V_i \cap V_j = \emptyset\ \forall i,\ j = 1,\ \ldots,\ k$ mit $i \neq j$, wobei $\bigcup_{i=1}^{k} V_i = V$, welche folgend *Partition*, *Cluster* oder auch *Komponente* genannt werden.

Das *Partition Graph Coloring Problem* (PCP) besteht nun darin, eine Menge V' zu finden, sodass $|V' \cap V_i| = 1,\ \forall\ i = 1,\ \ldots,\ k$ gilt (V' also genau einen Knoten für jede Partition enthält)

und weiter die Färbung S des durch die Menge V' und der für E implizierten Untermenge an Kanten $E' = \{(i,\ j) \in E \mid (i \in V') \vee (j \in V')\}$ konstruierbaren Graphen $G' = (V', E')$ minimal ist.

Als Lösung des Problems kann die Menge der *Repräsentanten* in V' gemeinsam mit deren Färbung S angesehen werden.

1.2. Ausgangsmaterial

Das Partition Graph Coloring Problem beschränkt sich nur auf den Weg vom Konfliktgraphen zum Lösungsgraphen. Für die Findung alternativer Routen durch ein Netzwerk gibt es bereits ausgiebig beschriebene und getestete Algorithmen, welche auch auf optische Netzwerke anwendbar sind. Mit Hilfe dieser Algorithmen (z. B. kürzeste Wege Algorithmen) können die für den Konfliktgraphen benötigten Knoten für jedes Kommunikationspaar gefunden werden. Wichtig hierbei ist es, dass es wenn möglich mehrere Alternativen für eine Übertragung geben sollte, da es sich sonst um ein normales *Graph Coloring* (GC) handeln würde. Die Findung der Kanten, welche Knoten verbinden, die dieselbe Teilstrecke verwenden, ist auch ein gelöstes Problem, welches das PCP hier nicht beantwortet.

Das PCP generiert eine Lösung aus dem Konfliktgraphen, welche eindeutig für jeden Kommunikationswunsch eine Route wählt und dieser eine Farbe zuordnet. Je dichter der Konfliktgraph, das heißt je mehr Kanten die verschiedenen Knoten verbinden, desto schwieriger ist es, eine geringe Anzahl an Farben für die optimierte Lösung zu finden. Dabei legt ein Algorithmus, der das PCP löst, nicht strikt die zu verwendenden Wellenlängen fest, sondern weist gleichen Wellenlängen nur die gleiche Zahl zu. Gleiches gilt für die Auswahl einer Route. Anstelle einer fixen Beschreibung, welchen Weg die ausgewählte Route durch das Netzwerk nimmt, zeigt ein Algorithmus für das PCP lediglich auf, welcher Knoten verwendet wurde. Die Zuordnung von Knoten zur Route muss daher, wie auch schon bei den Wellenlängen, später erfolgen.

1.3. Bisherige Ansätze

Bevor die in dieser Arbeit verwendete Lösungsstrategie für das PCP erläutert wird soll kurz erwähnt werden welche Lösungsansätze zu Beginn der Diplomarbeit evaluiert wurden.

1.3.1. Heuristisch

Li, Simha und Williamsburg (2000) definieren das PCP, beweisen, dass das Problem NP-schwierig ist und vergleichen verschiedene heuristische Ansätze. Aufgrund der Ergebnisse des Vergleichs wurde die Konstruktionsheuristik für die Variable Nachbarschaftssuche gewählt (siehe Abschnitt 2.2).

1.3.2. Metaheuristisch

Noronha und Ribeiro (2006) beschreiben eine Tabu-Suche, ein iteratives metaheuristisches Verfahren, welche durchnschnittlich um 20% bessere Ergebnisse liefern als die in Abschnitt 1.3.1 erwähnten Heuristiken.

1.3.3. Exakt

Y. Frota, Maculan, Noronha und Ribeiro (2010) zeigen einen Lösungsweg mithilfe von Branch-and-Cut und Hoshino, Y. A. Frota und Souza (2011) einen mit Branch-and-Price. Beide basieren auf der *ganzzahligen linearen Optimierung* (engl. *integer linear programming*, ILP) und verfolgen einen fundamental anderen Lösungsansatz als erwähnte (Meta-)Heuristiken bzw. die in dieser Arbeit vorgestellten Variablen Nachbarschaftssuche.

2. Lösungsansatz der Variablen Nachbarschaftssuche

Als Ausgangspunkt für unser Projekt setzen wir auf bereits bewährte Techniken aus anderen Bereichen der Optimierung und versuchen, diese bestmöglich auf das Partition Graph Coloring Problem zu adaptieren.

2.1. Aufbau der Variablen Nachbarschaftssuche

Bei der Variablen Nachbarschaftssuche handelt es sich um eine metaheuristische Methode, um verschiedenste Optimierungsprobleme zu lösen. Als Metaheuristik bezeichnet man dabei Algorithmen, die andere heuristische (und manchmal auch für Teilprobleme exakte) Optimierungsverfahren steuern und deren Ergebnisse sammeln und kombinieren.

Wie der Name bereits suggeriert versucht eine VNS auf Basis unterschiedlicher Nachbarschaften eine bereits bestehende, nicht optimale Lösung zu verbessern. Eine Nachbarschaft wird im Normalfall durch sogenannte *Moves*, also Züge, definiert, die beschreiben, wie man von einer Lösung zu einer Nachbarlösung kommt, die hoffentlich besser als die Ursprungslösung ist. Ein solcher Zug kann z.B. die Auswahl eines anderen Knotens als Repräsentant eines Clusters sein. Kann eine Lösung mit Zügen aus einer Nachbarschaft nicht mehr verbessert werden, dann befindet man sich in einem *lokalen Optimum*, das leider im Normalfall nicht dem globalen Optimum – der besten möglichen Lösung – entspricht.

Die Variable Nachbarschaftssuche setzt nun in ihrer eigentlichen Optimierungskomponente (*Variable Neighborhood Descent*, VND) nicht nur eine sondern gleich mehrere unterschiedliche Nachbarschaften hintereinander ein in der Hoffnung, dass ein lokales Optimum einer Nach-

barschaft nicht zwangsläufig auch ein lokales Optimum einer anderen Nachbarschaft ist. Man setzt also darauf, dass eine Lösung, die in einer Nachbarschaft nicht mehr verbessert werden kann in einer anderen Nachbarschaft sehr wohl noch Optimierungspotential besitzt. Diese neue Lösung kann dann vielleicht wieder mit einem Zug aus der ursprünglichen Nachbarschaft verbessert werden und so weiter.

Durchsucht man alle definierten Nachbarschaften systematisch (z.B. mittels *Best Improvement* Strategie, d.h. von allen möglichen Zügen innerhalb einer Nachbarschaft wird jener gewählt, der die Lösung maximal verbessert) hintereinander und wiederholt nach besseren Lösungen, dann stoppt dieser Prozess erst dann, wenn eine Lösung gefunden wurde, die ein lokales Optimum bezüglich sämtlicher Nachbarschaften darstellt.

Um diesem Zustand des vollkommenen Stillstandes zu entkommen, wird, nachdem dieses lokale Optimum für alle definierten Nachbarschaften erreicht wurde, mit immer größer werdenden Kraft die Lösung „geschüttelt“. Gemeint ist damit, dass die lokal optimale Lösung durch verschiedene Algorithmen verändert wird, welche nicht darauf abzielen, eine Lösung besser zu machen, sondern sie zunächst ein wenig, mit dem Fortschreiten der Optimierung aber auch möglichst stark zu verfremden, um einen neuen Startpunkt für das VND zu besitzen.

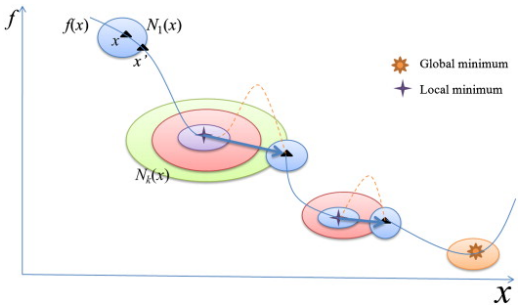


Abbildung 2.1.: Veranschaulichung der Variablen Nachbarschaftssuche. Chen u. a., 2013

Veranschaulichen kann man sich das Verfahren, wenn man sich alle gültigen Lösungen als eine große Fläche vorstellt. Je nach Güte der Lösung sind einzelne Teile dieser Fläche unterschiedlich hoch, gute Lösungen bilden Senken (wenige Farben werden für die Färbung benötigt), während schlechte Lösungen Berge bilden. Das VND tendiert nun dazu, sich in einer dieser Senken festzufahren, es wurde ein lokales Optimum bezüglich sämtlicher Nachbarschaften erreicht.

Mit Hilfe des „Schüttelns“ wird nun versucht, aus eben jenen Senken zu entkommen, um eine andere, hoffentlich tiefere Senke zu finden, also eine bessere Lösung. Nach diesem Schritt beginnt der Algorithmus nämlich wieder von vorne und sucht mit Hilfe der Nachbarschaften erneut ein lokales Optimum.

Obwohl die Variable Nachbarschaftssuche in der Theorie immer weiter laufen könnte und dabei hoffentlich immer wieder neue, bessere Lösungen erreicht, wird meist eine fixe Zeit oder eine Anzahl an Iterationen bzw. „Schüttelungen“ ohne Verbesserung als Abbruchkriterium gewählt.

Algorithmus 1 Pseudocode der Variablen Nachbarschaftssuche

```
1: Berechne Initiallösung mit onestepCD
2: Solange Terminationsbedingung nicht erfüllt
3:     Führe Variable Neighborhood Descent mit Best Improvement Strategie durch:
4:     Nachbarschaft l ← 1
5:     Solange l ≤ Anzahl der Nachbarschaften und Zeitlimit nicht erreicht
6:         Führe Nachbarschaft n_l aus
7:         Falls Lösung verbessert und l ≠ 1 dann
8:             l ← 1
9:         Sonst
10:            l ← l + 1
11:        Ende Falls
12:    Ende Solange
13:    Falls Beste Lösung verbessert oder k ≥ k_max dann
14:        k ← k_start
15:    Sonst
16:        k ← k + 1
17:    Ende Falls
18:    Wähle zufällig Nachbarschaft aus und „schüttle“ Lösung mit k zufälligen Zügen
19: Ende Solange
20: Returniere Beste gefundene Lösung
```

2.2. Konstruktionsheuristiken

Li, Simha und Williamsburg (2000) beschreiben mehrere schnelle Greedy Algorithmen (es wird beim Aufbau der Lösung die lokal beste Entscheidung getroffen ohne auf das große Ganze zu achten) zur näherungsweisen Lösung des Problems. Im Vergleich der benötigten Farben schneidet dabei der Algorithmus *onestepCD* am besten ab:

2.2.1. onestepCD

Zunächst werden in einem Vorverarbeitungsschritt alle Kanten aus dem Konfliktgraphen entfernt, die Knoten innerhalb eines Clusters miteinander verbinden (da für jede Kommunikation

nur genau ein Kommunikationsweg gewählt wird, können sich Knoten eines Clusters nie gegenseitig stören, die entsprechenden Kanten sind daher für die Lösung des Problems irrelevant und können gelöscht werden). Danach wird für jeden Cluster der Knoten mit der geringsten Anzahl an bereits eingefärbten Nachbarknoten bestimmt, der Knoten mit der insgesamt kleinsten Anzahl wird ausgewählt. Gibt es hier mehrere gleichwertige Kandidaten, wird aus diesen der Knoten mit der größten Anzahl an noch nicht eingefärbten Nachbarknoten gewählt bzw. – falls dies immer noch nicht eindeutig möglich ist – der erste Knoten aus diesen Kandidaten. Nachdem nun ein Knoten als Repräsentant für seinen Cluster bestimmt und ihm die kleinstmögliche Farbe zugewiesen wurde, werden alle anderen Knoten, die sich im selben Cluster befinden, gelöscht. Der Prozess wird solange fortgeführt, bis die Repräsentanten aller Cluster gewählt und eingefärbt sind.

2.2.2. PILOT

Bessere Ergebnisse als die einer naiven Konstruktionsheuristik wie *onestepCD* liefern häufig Varianten der gleichen Verfahren nach dem PILOT-Schema. Hier wird grundsätzlich nach den Entscheidungskriterien von *onestepCD* verfahren um jedoch nicht nur einen, sondern mehrere Knoten auszuwählen. Diese werden dann jeweils fixiert und durch Rekursion vollständige Lösungen berechnet. Von den verschiedenen Lösungen, die auf höherer Rekursionsebene zur Verfügung stehen, kann dann die vielversprechendste, also jene mit den wenigsten Farben, herangezogen werden, um eine lokal bessere Entscheidung zu treffen.

Es entsteht also eine beschränkte Enumeration der Lösungen, was deutlich zeitintensiver als die einmalige Berechnung von *onestepCD* ist, aber möglicherweise deutlich bessere Ergebnisse liefert.

2.3. Nachbarschaften

Wie bereits beschrieben handelt es sich bei Nachbarschaften um einzelne Algorithmen die versuchen, eine gültige Lösung in eine andere, bessere Lösung durch eine relativ kleine Änderung umzuwandeln. Wir haben uns zunächst auf die drei Nachbarschaften *ChangeColor*, *ChangeNode*

Abbildung 2.2.: Anschauliche Darstellung des PILOT-Verfahrens

und *DSATUR* konzentriert, wobei weitere Nachbarschaften noch zur Diskussion stehen.

Jede Nachbarschaft bietet auch eine Methode an, eine Lösung zu „schütteln“, welche ähnlich abläuft wie die Methode zur Suche des jeweiligen lokalen Optimums, allerdings ohne die Beschränkung ausschließlich bessere Ergebnisse zurückliefern zu müssen.

2.3.1. ChangeColor

Bei dieser sehr einfachen Nachbarschaft, die von der Tabu-Suche aus Noronha und Ribeiro (2006) inspiriert ist, wird versucht, alle Knoten umzufärben, welche mit der höchsten Farbe markiert wurden.

Erster Ansatz

Der erste Ansatz für diese Nachbarschaft arbeitete wie folgt. Alle Knoten, welche die höchste Farbe besitzen, werden mit einer neuen zufälligen, aber natürlich kleineren, Farbe markiert. Dies führt meist unweigerlich zu Konflikten zwischen verbundenen Knoten im Konfliktgraph, d.h. zu benachbarten Knoten mit gleicher Farbe.

Um diese Konflikte zu lösen wird ein zufälliger, im Konflikt stehender Knoten ausgewählt, um ihn mit einer passenden Farbe zu füllen. Zu beachten ist hierbei die Tatsache, dass diese passende Farbe kleiner sein muss als die frühere höchste Farbe, da ja eine Verbesserung (eine Reduktion der Farbenanzahl um eins) erzielt werden soll.

Sollte sich keine passende Farbe finden, wird auch dieser im Konflikt stehende Knoten mit einer zufälligen Farbe neu eingefärbt, was im Normalfall zu weiteren Konflikten führt. Es wird nun versucht, diesen gesamten Vorgang so lange zu wiederholen bis keine Konflikte mehr bestehen.

Sollte nach einer gewissen Anzahl an Iterationen keine konfliktfreie Lösung gefunden werden, bricht *ChangeColor* automatisch ab und liefert die letzte valide Lösung zurück.

Zweiter Ansatz

Da sich in Tests dieser erste Ansatz nicht bewährte, vor allem da er bei großen Probleminstanzen zu häufig zu keinem brauchbaren Ergebnis kam, wurde der Ansatz überarbeitet. Er basiert weiterhin auf der Umfärbung der Knoten mit der höchsten Farbe, diesmal allerdings ohne eine willkürliche Auswahl an in Konflikt stehenden Knoten.

Die veränderte Heuristik ist unter Algorihtmus 2 zu finden. Jeder Knoten, welcher die höchste Farbe besitzt, wird umgefärbt. Es werden alle Farben ausprobiert, welche kleiner als die ursprünglich höchste Farbe sind. Dann wird versucht, die adjazenten Knoten umzufärben und zwar so, dass auch hier nicht mehr die höchste Farbe verwendet wird. Wenn keine der ausprobierten Farben eine Verbesserung bringt, wird die Nachbarschaft abgebrochen. Konnten für alle Knoten, welche ursprünglich die höchste Farbe besaßen, eine Ersatzfarbe gefunden werden, wird die neue Lösung zurückgegeben, welches eine Verbesserung zum Eingabeobjekt darstellt.

2.3.2. ChangeNode

Diese Nachbarschaft baut auf der Tatsache auf, dass man aus einem Cluster nur jeweils einen Knoten auswählen muss.

Algorithmus 2 Pseudocode der *ChangeColor*-Nachbarschaft

Eingabe Solution s, Solution $full$
Ausgabe Solution l

```
1: l ← Kopie von s
2: c ← maximaleFarbe(l)
3: Für alle Knoten v ∈ l.graph für die gilt v.farbe = c
4:     Für Farbe i = 0 → c − 1
5:         v.farbe ← i
6:         Färbe alle adjazenten Knoten mit gleicher Farbe neu ein
7:         Falls maxFarbe(v.adjazenzen) < c dann
8:             Beende innere Schleife
9:         Ende Falls
10:    Ende Für
11:    Falls maxFarbe(v.adjazenzen) ≥ c dann
12:        Stoppe Nachbarschaft ohne bessere Lösung
13:    Ende Falls
14: Ende Für
15: retourniere l
```

Erster Ansatz

In einem ersten Ansatz werden wie bei *ChangeColor* alle Knoten höchster Farbe ausgewählt und umgefärbt. Nun wird wieder ein zufälliger Knoten aus den entstandenen Konflikten ausgewählt, welcher dann aber durch einen zufälligen Knoten aus dem selben Cluster ersetzt wird. An diesem Knoten wird dann ein neuer Einfärbeversuch unternommen, wobei die maximal zulässige Farbe natürlich wieder kleiner ist als die ursprünglich höchste Farbe.

Dieser Vorgang wird solange wiederholt bis alle Konflikte gelöst wurden oder zu viele Iterationen abgelaufen sind. Wurden alle Konflikte gelöst, wurde eine neue, verbesserte Lösung gefunden, ansonsten wird die Nachbarschaft abgebrochen.

Zweiter Ansatz

Bei dem ersten Ansatz dieser Nachbarschaft ergaben sich ähnliche Probleme wie beim ersten Ansatz von *ChangeColor*. Durch den höheren Aufwand, der durch das Tauschen von Knoten entsteht, erwies sich dieser erste Verusch der Nachbarschaft sogar als noch unzuverlässiger als *ChangeColor*. Mit Hilfe dieses zweiten Ansatzes, welcher es auch in die schlussendliche Implementierung schaffte, konnten verlässlicher und vor allem schneller Ergebnisse erzielt werden.

Die schlussendliche Variante der Nachbarschaft ist unter Algorithmus 3 zu finden. Ähnlich wie bei *ChangeColor* wird versucht, alle Knoten mit der maximalen Farbe umzufärben. Bei diesem zweiten Ansatz wird dies versucht, indem ein Knoten mit maximaler Farbe gegen einen anderen Knoten ausgetauscht wird. Dieser Austauschknoten muss natürlich auch Vertreter der selben Partition sein. Es wird dann versucht, diesen Knoten mit der kleinstmöglichen Farbe einzufärben. Sollte die Neueinfärbung mit einer Farbe niedriger als der bisher maximalen Farbe erfolgen, wird nach weiteren Knoten maximaler Farbe gesucht. Sollte der Algorithmus nicht in der Lage sein, einen passenden Ersatzknoten zu finden, welcher die Anzahl an verwendeten Farben verringert, bricht die Nachbarschaft ohne Verbesserung ab.

Algorithmus 3 Pseudocode der *ChangeNode*-Nachbarschaft

Eingabe Solution s, Solution $full$
Ausgabe Solution l

```
1: l ← Kopie von s
2: c ← maximaleFarbe(l)
3: Für alle Knoten v ∈ l.graph für die gilt v.farbe = c
4:     Für alle Knoten a ∈ full.graph für die gilt a.partition = v.partition ∧ a ≠ v
5:         Tausche Knoten v mit Knoten a in Solution l
6:         Färbe Knoten a mit minimal möglicher Farbe ein
7:         Falls a.farbe < c dann
8:             Beende innere Schleife
9:         Ende Falls
10:    Ende Für
11:    Falls a ≥ c dann
12:        Stoppe Nachbarschaft ohne bessere Lösung
13:    Ende Falls
14: Ende Für
15: retourniere l
```

2.3.3. DSATUR

Diese Nachbarschaft reduziert das Graph Partition Coloring Problem auf das normale Graph Coloring Problem ohne Clusterung der Knoten, indem die aktuell gewählten Repräsentanten fixiert werden und dieses so vereinfachte Problem (das allerdings immer noch NP-hart ist) wird nun mit folgendem Greedy Ansatz näherungsweise gelöst:

1. Finde den (noch nicht eingefärbten) Repräsentanten mit der größten Anzahl an eingefärbten Nachbarn (engl. *Degree Saturation*).

2. Bei Gleichstand: Wähle den Knoten mit der größten Anzahl an nicht eingefärbten Nachbarn.
3. Bei Gleichstand wähle den ersten Knoten in lexikographischer Abfolge.
4. Weise dem gewählten Knoten die kleinste mögliche Farbe zu.
5. Wiederhole, bis die Repräsentanten aller Cluster eingefärbt wurden.

Algorithmus 4 Pseudocode der Nachbarschaft *DSATUR*

```
1: Lösche die aktuelle Färbung
2: Sei t der ausgewählte Knoten
3: Sei D die maximale Saturation
4: Sei B die maximale Anzahl an nicht eingefärbten Nachbarn
5: Sei C die beste Färbung
6: Für i = 0, ..., k
7:     D ← −1
8:     B ← −1
9:     C ← −1
10:    t ← undefiniert
11:    Für alle v ∈ V'
12:        Falls Partition von v ist eingefärbt dann
13:            Überspringe diesen Knoten
14:        Sonst
15:            c ← kleinstmögliche Färbung von v
16:            d ← Anzahl an eingefärbten Nachbarn von v (Degree Saturation)
17:            Falls d < D dann
18:                Überspringe diesen Knoten
19:            Ende Falls
20:            b ← Anzahl an nicht eingefärbten Nachbarn von v (deg(v) − d)
21:            Falls d = D ∧ b ≤ B dann
22:                Überspringe diesen Knoten
23:            Ende Falls
24:            Wähle v als besten Kandidaten, zum jetzt Einfärben:
25:            D ← d
26:            B ← b
27:            C ← c
28:            t ← v
29:        Ende Falls
30:    Ende Für
31:    Färbe t mit C.
32: Ende Für
```

2.3.4. ChangeAll

Bei der Nachbarschaft *ChangeAll* handelt es sich um eine Kombination aus den beiden Nachbarschaften *ChangeColor* und *ChangeNode*. Beide Ansätze werden kombiniert, um ihre einzelnen Stärken gemeinsam auspielen zu können. In Tests mit großen und kleinen Probleminstanzen zeigte sich, das *ChangeAll* zu Verbesserungen führt, welche weder von *ChangeNode* noch von *ChangeColor* vorgenommen wurden.

Der Pseudocode ist unter Algorithmus 5 zu finden. Zuerst wird die bei *ChangeNode* nach einem neuen Knoten gesucht, welcher einen anderen Knoten mit höchster Farbe ersetzen soll. Dann wird allerdings wie bei *ChangeColor* dieser neue Knoten in verschiedenen Einfärbungen ausprobiert. Sollte eine dieser Kombinationen aus Knotenauswahl und Farben eine Verbesserung darstellen werden weitere Knoten mit maximaler Farbe gesucht. Sollte keine Verbesserung gefunden werden, bricht die Nachbarschaft ohne Ergebnis ab.

Algorithmus 5 Pseudocode der *ChangeAll*-Nachbarschaft

Eingabe Solution s, Solution $full$
Ausgabe Solution l
1: $l \leftarrow$ Kopie von s
2: $c \leftarrow maximaleFarbe(l)$
3: **Für alle** Knoten $v \in l.graph$ für die gilt $v.farbe = c$
4: **Für alle** Knoten $a \in full.graph$ für die gilt $a.partition = v.partition \wedge a \neq v$
5: Tausche Knoten v mit Knoten a in Solution l
6: **Für** Farbe $i = 0 \rightarrow c - 1$
7: $a.farbe \leftarrow i$
8: Färbe alle adjazenten Knoten mit gleicher Farbe neu ein
9: **Falls** $maxFarbe(a.adjazenzen) < c$ **dann**
10: Beende innerste Schleife
11: **Ende Falls**
12: **Ende Für**
13: **Falls** $maxFarbe(a.adjazenzen) < c$ **dann**
14: Beende mittlere Schleife
15: **Ende Falls**
16: **Ende Für**
17: **Falls** $maxFarbe(a.adjazenzen) \geq c$ **dann**
18: Stoppe Nachbarschaft ohne bessere Lösung
19: **Ende Falls**
20: **Ende Für**
21: retourniere l

3. Implementierung

Für die Implementierung des in Abschnitt 2 vorgeschlagenen Lösungsansatzes wurde eine möglichst flexible Struktur des Programms verwendet. Wie in Abschnitt 4.1 begründet, wurde als Implementierungssprache C++ gewählt. Durch den Einsatz von C++ kann ein einfach erweiterbares Programmgerüst geschrieben werden, welches es erlaubt, schnell und ohne großen Aufwand mehrere Nachbarschaften für die VNS zu implementieren.

Um diese flexible Struktur zu ermöglichen wurde das Programm in vier unabhängige Bereiche aufgeteilt, welche durch eine einzelne Startroutine zusammengehalten werden. Diese Bereiche können grob als Parser, Konstruktionsheuristik, Variable Nachbarschaftssuche und Nachbarschaften zusammengefasst werden. Hinzu kommt die sogenannte Solution, eine eigene Klasse, um eine Lösung sowohl zu speichern, aber auch um Möglichkeiten anzubieten, eine bestehende Lösung zu verändern. Diese einzelnen Bereiche sollen im folgenden Abschnitt ausführlich besprochen werden.

3.1. Solution

Bei Solution handelt es sich um die Basisklasse für alle weiteren Operationen. Sie speichert sowohl den Graphen, von dem bei der Lösung ausgegangen wird, als auch die Lösung selbst mit den für die Partition gewählten Knoten und den konkret verteilten Farben für diese Knoten. Außerdem stellt Solution Methoden bereit, um Veränderungen an Lösungen vorzunehmen, wie sie etwa von den Nachbarschaften gebraucht werden, um einzelne Knoten umzufärben oder innerhalb einer Partition auszutauschen. Außerdem besitzt Solution mehrere Methoden, einen Problemgraphen aus einer Datei einzulesen und diesen wiederum in einem Solution-Objekt zu speichern. Diese Methoden zum Einlesen von Dateien werden genauer in Abschnitt 3.2

besprochen.

Listing 3.1: Ein Ausschnitt aus der Signatur der Solutionklasse

```
1  class Solution {
2      public:
3          Solution();
4          Solution(Solution *s);
5          ~Solution();
6          void requestDeepCopy();
7
8          Graph *g;
9          int numParts;
10         int *partition;
11         std::vector<Vertex> *partNodes;
12         int colorsUsed;
13         int *representatives;
14         int *copyCounter;
15 };
```

Gespeichert wird eine Referenz auf den Lösungsgraphen. Dieser Graph enthält ausschließlich jene Knoten, welche als Repräsentanten für ihre Partition ausgewählt wurden. Der Graph muss daher verändert werden, wenn eine Partition ihren Repräsentanten wechselt. Des Weiteren wird die Anzahl an Partitionen gespeichert sowie die Anzahl an verwendeten Farben für die konkrete Lösung.

Im `partition`-Array werden für jede Partition genau eine ausgewählte Farbe gespeichert, wobei sich diese Farben natürlich von Lösung zu Lösung unterscheiden können, als Index dient die Partitions-ID. Das Array `representatives` bietet eine Zuordnung von Partition zu Knoten. Ein Wert an der Position n zeigt auf jenen Knoten, der die Partition n vertritt.

Einen Sonderfall stellt das Vector-Array `partNodes` dar. Es wird ausschließlich bei der eingelesenen Probleminstanz eingesetzt und speichert dort für jede Partition alle zugeordneten Knoten ab. Dies dient der schnellen Findung von alternativen Knoten für eine bestimmte Partition und findet in mehreren Nachbarschaften Verwendung.

3.1.1. Graph-Datenstruktur

Für die Speicherung von sowohl Lösungs- als auch Problemgraphen dient die `adjacency_list`-Datenstruktur, welche von Boost bereitgestellt wird. Der Einsatz von Boost, einer C++ Bib-

liothek, wird in Abschnitt 4.6 beschrieben. Es wurde bewusst eine Liste als Grundlage der Speicherung verwendet, um ein schnelles Iterieren über Knoten und Kanten zu ermöglichen. Die ebenfalls von Boost bereitgestellte `adjacency_matrix` (zwei Knoten sind adjazent, wenn sie über eine Kante miteinander verbunden sind) hatte sich in einem Test als ungeeignet erwiesen, da nicht alle Operationen wie das Hinzufügen und Löschen von Knoten unterstützt werden. Des Weiteren wurde die Option für ungerichtete Graphen gesetzt, da das PCP keine gerichteten Beziehungen zwischen Knoten vorsieht. Dies führt dazu, dass, sobald eine Kante vom Knoten A zu Knoten B führt, automatisch auch eine Beziehung von Knoten B zu Knoten A führt.

Boost bietet außerdem die Wahl zwischen verschiedenen Datenstrukturen für die Speicherung des Graphobjektes. Selbst wenn die Auswahl auf eine Liste gefallen ist, gibt es noch mehrere Möglichkeiten, die Datenstruktur zu beeinflussen. Zum einen kann bei der Speicherung der Knoten sowohl eine normale Liste als auch ein Vector ausgewählt werden. Die selbe Möglichkeit präsentiert sich dann auch für die Speicherung der Kanten. Ein Vector ist eine Datenstruktur ähnlich einem Array, bei dem alle Datenelemente im Speicher hintereinander angeordnet sind, welche aber vergrößert oder verkleinert werden kann.

Für die Speicherung der Knoten wurde ein Vector herangezogen, zum einen, da nun ein Knoten direkt über einen zahlenbasierten Index ansprechbar ist, zum anderen, weil während der Laufzeit des Programms nur zwei Mal Knoten hinzugefügt oder entfernt werden: Einmal beim Einlesen der Probleminstanz aus einer Datei und einmal bei der Berechnung der Ausgangslösung. Bei allen anderen Veränderungen der Lösung erfolgt der Austausch von Knoten direkt an der Stelle des zu ersetzenden Knoten. Es ist daher anzunehmen, dass während der Laufzeit des Programms keine größeren Reallokationsoperationen vorgenommen werden müssen.

Schwieriger ist die Wahl schon bei der Frage, ob für die Speicherung der Kanten eine Liste nicht besser geeignet wäre. Da in verschiedenen, weiter unten beschriebenen Nachbarschaften Knoten ausgetauscht werden, ein Vorgang der darauf basiert, die alten Adjazenzen eines Knotens zu kappen und an deren Stelle neue Adjazenzen zu schaffen, welche dem ausgetauschten Knoten entsprechen, handelt es sich bei der Kantenspeicherung um eine Datenstruktur, die häufigen Änderungen unterliegt. Da die meisten Knoten aber eine ähnliche Anzahl an Adjazenzen haben, würde auch hier ein Vektor keine allzu häufigen Reallokationen durchführen

müssen. In einem Test ergab sich unter Verwendung eines Vectors als Speicherstruktur für die Adjazenzinformationen eine bis zu 20% höhere Geschwindigkeit im Vergleich zur Verwendung einer Liste. Dies ist wohl vor allem auf die höhere Datenlokalität zurückzuführen, durch die der Prozessor bereits ganze Speicherbereiche in den Cache laden kann, wenn über die Adjazenzen iteriert wird.

3.1.2. Kopierkonstruktor

Besonders interessant bei der Solution-Klasse ist die automatisch bereitgestellte Kopiermechanik. Einer der besonderen Eigenschaften von C++ ist es, für jede Klasse automatisch einen sogenannten Kopierkonstruktor bereitzustellen, welcher als Parameter ein Objekt des selben Typs entgegen nimmt und dieses kopiert. Da im Falle einer Lösung aber nicht immer alle Daten kopiert werden müssen, wurde in dieser Arbeit eine eigene Kopiermechanik implementiert. Wie in Listing 3.2 zu sehen ist wird ein Großteil der Referenzdatentypen direkt übernommen, da diese häufig gar nicht verändert werden.

Das Kopieren einer Lösung kommt vor allem bei den verschiedenen Nachbarschaften zum Einsatz. Eine bestehende Lösung soll verändert werden, ohne dass die Information der Ausgangslösung verloren geht. Also wird die bestehende Lösung kopiert und die Veränderungen nur auf der neuen Lösung ausgeführt. Da viele der Nachbarschaften darauf basieren, dass ausschließlich die Lösungsfarben verändert werden, kann zum Beispiel, anstatt den gesamten Graphen zu kopieren, eine Referenz auf den alten Graphen aufrecht erhalten werden. Selbiges gilt natürlich auch für alle direkt mit dem Graph in Verbindung stehenden gespeicherten Daten.

Listing 3.2: Der Kopierkonstruktor der Solutionklasse

```
1  // Copy an existing solution, this will use the same graph object as the
2  // original, to cut these ties, use requestDeepCopy()
3  Solution::Solution(Solution *toCopy) {
4      this->partNodes = NULL;
5      this->g = toCopy->g;
6      this->copyCounter = toCopy->copyCounter;
7      // increment copycounter, to keep track of referencing solution
8      *this->copyCounter += 1;
9      this->numParts = toCopy->numParts;
10     this->partition = new int[this->numParts];
11     this->representatives = toCopy->representatives;
12     this->colorsUsed = toCopy->colorsUsed;
```

```
13      this->partitionMap = get(vertex_index1_t(), *this->g);
14      this->idMap = get(vertex_index2_t(), *this->g);
15
16      for (int i = 0; i < this->numParts; i++) {
17          this->partition[i] = toCopy->partition[i];
18      }
19  }
```

Nun könnten aber Probleme entstehen. Zum einen, wenn eine Nachbarschaft beispielsweise Knoten aus dem zu Grunde liegenden Graphen austauscht um eine bessere Lösung zu erhalten. Da durch das unechte Kopieren der in einem Solutionobjekt referenzierte Graph auch bei anderen Lösungen zum Einsatz kommt, würde eine Veränderung auf einem Graphen zu Veränderungen und aller Wahrscheinlichkeit nach zu Fehlern in alternativen Lösungen führen, welche dann unweigerlich unzulässige Ergebnisse zur Folge hätten. Daher kann mit der Methode `requestDeepCopy` eine vollständige Kopie des Solutionobjektes angefordert werden. In dieser Methode werden dann alle verbleibenden Fremdreferenzen noch einmal kopiert und in eigens für das anfragende Solutionobjekt reservierte Speicherbereiche geschrieben.

Listing 3.3: Die Methode `requestDeepCopy` der Solution Klasse

```
1   // Cut ties between a copy and the underlying graph by
2   // making a copy of the graph
3   void Solution::requestDeepCopy() {
4       // Copy graph
5       Graph *cp = g;
6       g = new Graph(*g);
7       // Reset property maps
8       this->partitionMap = get(vertex_index1_t(), *g);
9       this->idMap = get(vertex_index2_t(), *g);
10
11      // Copy representatives array
12      int *rep = representatives;
13      representatives = new int[numParts];
14      for (int i = 0; i < numParts; i++) {
15          representatives[i] = rep[i];
16      }
17
18      // delete the old copy if counter reached zero
19      *copyCounter -= 1;
20      if (*copyCounter <= 0) {
21          delete copyCounter;
22          delete cp;
23          delete[] rep;
24      }
25      copyCounter = new int;
26      *copyCounter = 1;
27  }
```

Ein anderes Problem entsteht beim Aufräumen von Objekten. Sollte durch das Löschen eines Solutionobjektes auch dessen referenzierter Graph verschwinden, würde das zu Speicherfehlern bei anderen Objekten führen, welche auf den selben Graphen zeigen. Daher wurde der Destruktor, welcher zum Aufräumen eines Objektes aufgerufen wird, überschrieben und löscht nun nur noch referenzierte Objekte, wenn kein anderes Objekt mehr Referenzen auf das selbe Objekt hält. Um dies zu ermöglichen wird von allen Solutionobjekten, welche auf den selben Graphen referenzieren ein Counter referenziert, welcher die Anzahl der Verweise auf den selben Graphen mitzählt. So lange dieser Zähler über eins ist wird kein Objekt gelöscht. Wird nun der Destruktor eines Solutionobjektes aufgerufen, überprüft dieser, ob er die letzte Referenz auf diese Objekte hält. Wenn ja, löscht er diese und seine eigenen Referenzen, wenn nicht, löscht er wiederum nur seine eigenen Referenzen und vermindert den Zähler um eins.

Listing 3.4: Der Destruktor der Solutionklasse mit Rücksichtnahme auf eventuell verbleibende Referenzen

```
1  // Destructor
2  Solution::~Solution() {
3      // Decrement copyCounter, if <= 0 delete all information about the
4      // referenced graph and corresponding arrays
5      *copyCounter -= 1;
6      if (*copyCounter <= 0) {
7          delete g;
8          delete copyCounter;
9          delete[] representatives;
10     }
11     delete[] partition;
12     delete[] partNodes;
13 }
```

3.1.3. StoredSolution

Um eine einzelne Lösung von einer anderen zu unterscheiden sind zwei Informationen notwendig: die Auswahl der Repräsentanten und die Auswahl an Farben. Mit diesen beiden Informationen und dem Ausgangsgraphen, aus dem die Lösung berechnet wurde, ist eine Lösung eindeutig rekonstruierbar. Um zu verhindern, dass die VNS-Schleife immer wieder auf die selben Lösungen kommt ohne echte Verbesserungen zu erzielen, werden alle neuen, verbesserten Lösungen komprimiert gespeichert.

Zur komprimierten Speicherung wurde eine eigene Datenstruktur definiert, welche eben nur jene oben erwähnten Informationen abspeichert. Dazu werden zwei Arrays verwendet, welche einerseits die für einzelne Partitionen verwendeten Farben, andererseits aber auch die eindeutige Identifikationsnummern der Repräsentanten abspeichern. Um eine Solution einfach in solch eine StoredSolution umzuwandeln wurde ein Konstruktor definiert, welcher ein klassisches Solutionobjekt übernimmt.

Listing 3.5: Die Signatur von StoredSolution

```
1  struct StoredSolution {
2      StoredSolution(Solution& toStore);
3      ~StoredSolution();
4
5      int n;
6
7      int *colors;
8      int *representatives;
9
10     std::string toString();
11 };
```

Für die schnelle Zuordnung einer neuen Lösung zu den bestehenden Lösungen wird eine Hashtable-Datenstruktur verwendet. Durch die Verwendung einer Hashtable ist eine quasi sofortige Zuordnung einer Lösung zu dem entsprechenden Platz in der Hashtable möglich. Sollte eine Lösung schon einmal aufgekommen sein, so wird dieser Lösungsstrang verworfen, da er nicht zu neuen Lösungen führen würde. Die Implementierung der Hashtable wird von Boost, beschrieben im Abschnitt 4.6, bereitgestellt. Um die Hashtable zu betreiben sind außerdem noch eine Methode zur Hashberechnung sowie eine Methode zum Vergleich zweier gespeicherten Solutions notwendig. Der Hash für ein StoredSolutionobjekt wird aus den gesammelten Werten der beiden internen Arrays (Repräsentanten und Farben) berechnet, während für die Vergleichsmethode die beiden Arrays Wert für Wert verglichen werden. Die von Boost bereitgestellte Hashtable verwendet bei gleichen Hashwerten für unterschiedliche Lösungen eine einfache Liste, um die Ergebnisse zu speichern.

3.1.4. Plausibilitätsprüfung zu Testzwecken

Um eine manuelle Überprüfung der Korrektheit einer Lösung zu umgehen, wird innerhalb der VNS, siehe Abschnitt 3.4, eine Plausibilitätsprüfung durchgeführt. Diese Überprüfung stellt anhand der innerhalb einer Solution gespeicherten Eigenschaften und anhand des ursprünglichen Problemgraphen fest, ob es sich um eine valide Lösung handelt oder nicht. Diese Überprüfung erfolgt standardmäßig am Ende der VNS und überprüft das beste erzielte Ergebnis, welches dann auch als Endergebnis präsentiert wird. Mit Hilfe eines Programmparameters, welcher zum Ausführungsstart angegeben werden muss, ist es auch möglich, eine solche Überprüfung für jede Lösung einzuschalten, welche eine Verbesserung zur vorherigen darstellt.

Innerhalb dieses Checks werden verschiedenste Dinge überprüft. Zum einen wird getestet, ob zwei verbundenen Knoten die selbe Farbe zugeteilt wurde. Dann wird überprüft, ob die gespeicherte Anzahl an Farben mit den tatsächlich verwendeten Farben übereinstimmt. Dies ist besonders wichtig, da ja die Anzahl an Farben das Hauptkriterium für eine bessere Lösung darstellt. Weiters wird überprüft, ob alle Partitionen vertreten sind und ob die jeweiligen Repräsentanten richtig abgespeichert wurden.

Zuletzt wird noch überprüft, ob der Lösungsgraph mit dem Ursprungsgraphen übereinstimmt. Dazu werden verschiedene Fakten überprüft. Zum einen werden fehlende oder überschüssige Kanten in der Lösung gesucht, zum anderen wird überprüft, ob die ausgewählten Repräsentanten einer Partition wirklich dieser Partition zugehörig sind.

Sollte keine dieser Überprüfungen auf Fehler stoßen, wird angenommen, es handle sich um eine valide Lösung für das PCP. Durch diese Überprüfung konnten auch schon Fehler innerhalb einzelner Nachbarschaften nachgewiesen und schließlich behoben werden.

3.2. Parser

Damit erzielte Ergebnisse mit älteren Programmversionen, aber auch mit anderen Lösungsansätzen für das PCP verglichen werden können, ist es von Vorteil, immer wieder die selbe Eingabe zu verwenden. Außerdem bietet sich eine solche Möglichkeit an, wenn, wie in dieser Arbeit ein Zufallsfaktor zu einer Verbesserung der Lösung führen kann. Durch immer wieder

gleiche Eingaben kann ein direkter Vergleich auch zwischen unterschiedlich parametrisierten Programminstanzen angestellt werden.

Es gibt mehrere Dateiformate, mit denen Probleminstanzen des PCP abgespeichert werden können. In dieser Arbeit wurde vor allem auf jenes Format Wert gelegt, in dem auch die meisten Testinstanzen der einschlägigen Literatur vorliegen. Allerdings wurde im Zuge von ausgeweiteten Testungen auch ein zweites Format zum Einlesen eines Problemgraphen implementiert.

Eine `.pcp`-Datei ist eine einfache Textdatei, welche allerdings nach einer bestimmten Struktur aufgebaut ist. Die allererste Zeile einer solchen Datei besteht aus drei Zahlen, welche jeweils durch ein Leerzeichen getrennt sind. Diese drei Zahlen stehen für die Anzahl der Knoten, die Anzahl der Kanten und die Anzahl der Partitionen im Problemgraph, genau in dieser Reihenfolge. Nun folgen n Zeilen an Zahlen, mit genau einer Zahl pro Zeile, wobei n die Anzahl an Knoten ist. Diese Zahlen stehen für die Partitionen, denen die jweiligen Knoten zugeordnet werden. Die erste Zeile dieses Abschnitts beinhaltet also die Partition des ersten Knotens, die zweite Zeile die des zweiten Knotens und so weiter. Zuletzt folgen noch m Zeilen mit jeweils einem Zahlenpaar, welche die Kanten zwischen den Knoten definieren, wobei m der Anzahl an Kanten insgesamt entspricht.

Listing 3.6: Eine einfache *.pcp*-Beispieldatei

```
1  4 5 2
2  0
3  1
4  1
5  0
6  0 1
7  0 2
8  1 2
9  1 3
10 2 3
```

Der Parser, welcher eine solche Datei einliest, ist in die Klasse Solution eingearbeitet. Die statische Methode `fromPcpStream` liest eine Datei von einem Eingabestream ein und wandelt sie in ein Solutionobjekt mit entsprechendem Problemgraphen um. Dieses Solutionbjekt wird während der gesamten Laufzeit des Programmes nicht verändert oder gelöscht und dient als Ausgangspunkt der Konstruktionsheuristik, aber auch als Referenz für Nachbarschaften, welche einzelne Knoten austauschen. Außerdem speichert dieses Solutionobjekt als einziges eine direkte

Zuordnung von einer Partitions-ID zu den verfügbaren Knoten in der Partition ab.

3.3. Konstruktionsheuristik

Als Konstruktionsheuristik wurde der in Abschnitt 2.2 vorgestellte Algorithmus *onestepCD* gewählt. Die Implementierung des Algorithmus erfolgt mit Hilfe der Methode `onestepCD` und. erzeugt ein neues Solutionobjekt, als einziger Übergabeparameter wird das aus der Datei eingelesene Solutionobjekt benötigt, welches ja den vollständigen Problemgraphen enthält. Dieses Solutionobjekt bildet dann die Ausgangsbasis für die Verbesserung in der Variablen Nachbarschaftssuche. Der Graph, welcher in dem erzeugten Solutionobjekt referenziert wird, ist ein anderer Graph als jener, der in der Gesamtlösung verwendet wird. In dem referenzierten Graphen befindet sich nur mehr ein Knoten per Partition, welcher auch den ausgewählten Repräsentanten der jeweiligen Partition darstellt.

3.3.1. PILOT

Alternativ zu `onestepCD` wurde auch eine PILOT-inspirierte Variante der Heuristik implementiert. Zuerst werden hier wieder Kanten zwischen Knoten der selben Partition entfernt, um dann schrittweise rekursiv eine Lösung zu berechnen. Im Unterschied zu `onestepCD` werden allerdings Knoten nicht trivial gewählt sondern eine Menge an Knoten gesucht, für die dann eine Lösung durchberechnet wird. So können auch andere, womöglich bessere Lösungen, gefunden werden.

Wie `onestepCD` operiert auch `pilot` auf einem Solutionobjekt, dass dann nahtlos an die VNS weitergereicht werden kann.

3.4. Nachbarschaftssuche

Die Variable Nachbarschaftssuche, wie sie in Abschnitt 2.1 beschrieben wurde, wurde als eine einzelne Funktion implementiert, welche den Ablauf der Nachbarschaften steuert. Um den Ablauf des Programmes möglichst einfach koordinieren zu können ist eine Vielzahl an Optionen beim Programmstart spezifizierbar. Vieler dieser Optionen beeinflussen direkt oder indirekt den

Ablauf der VNS.

Die wichtigsten Parameter der Methode `vnsRun`, deren Signatur in Listing 3.7 zu sehen ist, sind die Übergabe der derzeit besten Lösung `best`, welche von der Konstruktionsheuristik erzeugt wurde und die Übergabe der vollständigen, unveränderten Probleminstanz `orig`. Auch von essentieller Bedeutung ist der String `units`, welcher eine Zeichenkette enthält, der die Reihenfolge der verwendeten Nachbarschaften bestimmt.

Die nächsten Parameter bestimmen vor allem die Ausführungszeit der VNS. Der übergebene Parameter `unsuccessfulShake` bestimmt die maximale Anzahl an Schüttelvorgängen, welche ohne Verbesserungen ablaufen dürfen, bevor die VNS beendet wird. Der Parameter `shakeStart` bestimmt die initiale Stärke des ersten Schüttelvorganges und steht damit direkt im Zusammenhang mit der Stärke der Randomisation, welche von diesen Schüttelvorgängen ausgeführt wird. Mit Hilfe des Parameters `shakeIncrement` wird bestimmt, wie schnell der Stärkeparameter des Schüttelvorgangs ansteigt, sollten keine Verbesserung durch das Schütteln entstehen. Schließlich kann mit dem Parameter `maxTime` direkt die maximale Anzahl an Sekunden angegeben werden, die die VNS läuft, bevor sie sich selbst beendet.

Die letzten beiden Parameter dienen der Einstellung, ob alle verbesserten Zwischenlösungen (`checkIntermediate`) oder die endgültige Lösung (`checkFinal`) durch die Plausibilitätsprüfung überprüft werden sollen. Da die Überprüfung durchaus aufwendig ist und daher etwas Zeit in Anspruch nimmt, ist es meist eine gute Lösung, nur die endgültige Lösung überprüfen zu lassen, außer man will eine einzelne Nachbarschaft auf Fehler hin testen. Die Plausibilitätsprüfung selber ist in Abschnitt 3.1.4 beschrieben und testet verschiedenste Informationen, um die Korrektheit einer Lösung zu garantieren.

Listing 3.7: Signatur der Funktion, welche die Variable Nachbarschaftssuche ausführt und steuert

```
1 /// Run the VNS with a maximum of k iterations.
2 /// Returns after either there were a number of unsuccessful shakes
3 /// or if 'time' seconds have elapsed.
4 /// Shaking starts with shakeStart steps and is then incremented by
5 /// shakeIncrement.
6 Solution *vnsRun( Solution& best, Solution& orig, std::string units, int
    ↪ unsuccessfulShake, int shakeStart, int shakeIncrement, int maxTime,
    ↪ bool checkIntermediate, bool checkFinal);
```

3.5. Nachbarschaften

Die Nachbarschaften bilden das Herzstück der VNS. Eine einzelne Nachbarschaft versucht kleine Veränderungen vorzunehmen, welche die Lösung verbessern und gibt diese Verbesserungen dann an weitere Nachbarschaften weiter. Jede Nachbarschaft verfolgt ihren eigenen Ansatz zur Verbesserung und muss nicht immer bessere Ergebnisse liefern.

Bei der Implementierung der Nachbarschaften wurde versucht, ein möglichst einfach erweiterbares Gerüst zu schaffen, um schnell und einfach neue Ideen für Nachbarschaften ausprobieren zu können. Jede Nachbarschaft wurde daher als eigene Klasse implementiert, welche aber alle von einer einzelnen Basisklasse erben. Diese Basisklasse definiert ein Grundgerüst, welches von jeder Nachbarschaft ausgefüllt werden muss.

Zu diesem Grundgerüst zählt nicht nur die Implementierung der eigentlichen Nachbarschaft, sondern auch eine Funktion, um den Namen der Nachbarschaft auszugeben, eine Funktion, um ein Kurzzeichen für die Nachbarschaft zu definieren und eine Funktion, um eine bestehende Lösung zu schütteln, wie dies in Abschnitt 2.1 beschrieben wurde. Jede dieser Funktionen muss für eine Nachbarschaft implementiert werden und jede Nachbarschaft sollte nur gültige Lösungen zurückliefern oder zumindet einen Indikator liefern, ob die zurückgelieferte Lösung ein unbrauchbares Ergebnis darstellt.

Die Basisklasse, von der jede Nachbarschaft erbt, heißt `VNS_Unit`. Alle Nachbarschaften müssen alle definierten Methoden und Prozeduren implementieren. Jeder Nachbarschaft wird dabei beim Aufrufen der Verbesserungsmethode die zu verbessernde Lösung und auch jenes Solutionobjekt, welches die unveränderte Aufgabenstellung beeinhaltet, übergeben. Das selbe gilt für die Methode zum Schütteln der Lösung mit dem zusätzlichen Parameter der Stärke des Schüttelns. Somit kann von der VNS bestimmt werden, wie stark die Lösungen randomisiert werden.

Listing 3.8: Signatur der Basisklasse `VNS_Unit`, von welcher alle Nachbarschaften erben

```
1 class VNS_Unit {
2     public:
3         /// Compute the new improved solution of this neighborhood
4         virtual Solution *findLocalMin(Solution& curBest, Solution& full)
              ↪  = 0;
5
```

```
6           /// Shuffle a solution using the neighborhood as a base
7           virtual Solution *shuffleSolution(Solution& cur, Solution& full,
                ↪ int numSteps) = 0;
8
9           virtual ~VNS_Unit();
10
11          /// Returns a given name for the neighborhood
12          virtual const std::string getName() = 0;
13
14          /// Returns a given (unique) character used to quickly reference
15          /// an unit via command line argument.
16          static const char getAbbreviation();
17  };
```

3.5.1. ChangeColor

Bei dieser Nachbarschaft wurde versucht, die in Abschnitt 2.3.1 vorgeschlagenen Ansätze in eine konkrete Nachbarschaft zu übersetzen. Da sich bei Tests herrausstellte, dass der erste beschriebene Ansatz mit zufälliger Auswahl an Farbe und zufälliger Bestimmung eines in konfliktstehenden Knotens bei großen Instanzen zu schlechten Ergebnissen führte, wurde ein etwas modifizierter Ansatz gewählt.

Die Nachbarschaft wurde wie der in Algorithmus 2 beschriebene Pseudocode implementiert. Da er ausschließlich die Farben bearbeitet und sonst keine Veränderungen am Graphen durchführt, braucht die Nachbarschaft nur eine flache Kopie des vorherigen Solutionobjektes und muss daher nicht auf den in Abschnitt 3.1.2 beschriebenen Kopiermechanismus zurückgreifen.

3.5.2. ChangeNode

Ähnlich wie bei *ChangeColor* erwies sich der in Abschnitt 2.3.2 beschriebene Ansatz in der Realität bei großen Instanzen als nicht verlässlich. Häufig lieferte die Nachbarschaft keine gültigen Ergebnisse, da die Anzahl der Konflike zu stark angestiegen war. Daher wurde auch hier ein modifizierter Ansatz gewählt, welcher in Einzelfällen zwar schlechtere Ergebnisse liefert, dafür aber deutlich öfter bessere Lösungen erzeugen kann.

Die Implementierung dieser Nachbarschaft erfolgte unter Verwendung der in Abschnitt 3.1.2 beschriebenen Mechanismen zur Abkopplung einzelner Solutionobjekte von anderen, älteren Lösungen. Da diese Nachbarschaft auch den zu Grunde liegenden Graphen verändert, ist es

notwendig, ihn auf einem eigenen, abgetrennten Graphenobjekt arbeiten zu lassen.

3.5.3. DSATUR

DSATUR steht im Kontrast zu den *Change**-Heuristiken: Das PCP wird auf das Graph Coloring Problem ohne Einteilung der Knoten in Partitionen reduziert. Konkret belässt die Heuristik also die Auswahl der Knoten, färbt sie aber alle erneut ein. Dabei wird nach bestimmten Regeln verfahren, welche in 2.3.3 genauer beschrieben sind.

Ein wesentlicher Vorteil dieser Heuristik ist, dass sie nur Änderungen an der Färbung durchführt, die Strukur des Graphs aber nicht verändert und somit ohne das Kopieren des Graphens auskommt, was sie zur schnellsten Methode macht.

3.5.4. ChangeAll

Obwohl *ChangeAll* eine große Ähnlichkeit mit sowohl *ChangeColor* als auch *ChangeNode* aufweist, entstand diese Nachbarschaft erst wesentlich später. Der Ansatz dieser Nachbarschaft ist im Abschnitt 2.3.4 zu finden. Es zeigte sich, dass durch die Verwendung von *ChangeAll* weitere Verbesserungen durchgeführt werden konnten, welche nicht mit Hilfe von anderen Nachbarschaften erzielt werden konnten.

Ähnlich wie bei der *ChangeNode*-Nachbarschaft werden auch bei dieser Heuristik einzelne Knoten der Lösung ausgetauscht. Da dies zu einer Veränderung im Lösungsgraphen führt, ist es notwendig, das erzeugte Solutionobjekt mit samt dem Graphenobjekt vollständig abzutrennen. Dazu werden die in Abschnitt 3.1.2 beschriebenen Methoden verwendet.

4. Entwicklung

Im folgenden Abschnitt sollen verschiedene Programmiersprachen, Programme und Code-Bibliotheken besprochen werden, welche bei der Verwirklichung dieses Projektes zum Einsatz kamen. Viele der erwähnten Programme und Bibliotheken sind Standardwerkzeuge der Entwicklung unter Linux und sind unter Open-Source-Lizenzen frei verfügbar.

Um sowohl ein funktionierendes als auch bequem zu bedienendes Programm zu entwickeln, kamen auch mehrere Programmiersprachen zum Einsatz. Jede dieser Programmiersprachen hat ihre Stärken in einem bestimmten Bereich und es wurde versucht, jede Sprache nach ihrer individuellen Stärke einzusetzen.

4.1. C++

Als primäre Programmiersprache wurde C++ gewählt. Verschiedene Gründe sprachen für C++, unter anderem die an C angelehnte Syntax, aber auch die gute Performance des generierten Maschinencodes im Vergleich zu Sprachen, welche üblicherweise in einer virtuellen Maschine laufen oder gar interpretiert werden. Zu all diesen Gründen kam noch hinzu, dass für C++ bereits ausgezeichnete Codebibliotheken zur Verfügung stehen, welche ein schnelles und effizientes Arbeiten ermöglichen.

Bei C++ handelt es sich um eine im Jahr 1979 von Bjarne Stroustrup entwickelte Programmiersprache, welche anfangs als *C mit Klassen* ausgelegt wurde. Über mehrere Entwicklungsschritte entwickelte sich eine Sprache, dessen Wurzeln in der Sprache C immer noch zu erkennen sind, dessen Möglichkeiten jene von C aber bei weitem übersteigen. Trotzdem wird C++ häufig in Verbindung mit C gebracht, unter anderem wohl deswegen, weil beide Sprachen häufig gemeinsam eingesetzt werden.

Wie C handelt es sich bei C++ um eine statisch-typisierte Programmiersprache, das heißt der Compiler kann schon während der Übersetzung die syntaktisch korrekte Verwendung von Datentypen sicherstellen. Dadurch können solche Überprüfungen während der Ausführung des Programms entfallen, was zu einer enormen Beschleunigung der Ausführung des Programms führt. Außerdem ermöglicht es dem Entwickler Fehler mit inkorrekt verwendeten Datentypen einfacher zu erkennen, da eine Fehlermeldung des Übersetzers eine genaue Zeile angeben kann, in der der Fehler verursacht wird.

Obwohl sich C++ wesentlich von C unterscheidet, ist es immer noch kompatibel mit C. Eine Stück C Code kann immer noch in C++ verwendet und übersetzt werden. Dadurch können nicht nur Code-Bibliotheken aus dem C++-Umfeld verwendet werden, sondern auch ursprünglich für C gedachte Bibliotheken eingebunden und verwendet werden. Da C immer noch eine der beliebtesten Programmiersprachen weltweit und vor allem **die** Sprache für Systemprogrammierung und performante Software ist, bedeutet diese Mitnahme der Vorteile aus der C-Welt in die Welt von C++ einen großen Vorteil bei der Programmierung.

Des Weiteren wurde C++ möglichst plattformunabhängig designt. Während manche Funktionen, welche vom Betriebssystem bereit gestellt werden, natürlich nicht wirklich plattformunabhängig sein können, wurde in C++ versucht, keine Funktionen oder Eigenschaften zu verwenden, welche nur von einem bestimmten System ausgeführt werden können. Aus diesem Grund sind auch die meisten Code-Bibliotheken für C++ für die meisten Plattformen erhältlich.

Obwohl C++ wesentlich mehr Funktionen als C bietet, ist trotzdem noch die Anlehnung an C zu erkennen. Daher ist es auch nicht verwunderlich, dass C++-Programme meist nur geringfügig langsamer sind als vergleichbare C-Programme. In manchen Fällen ist C++ sogar schneller als C, da der Compiler andere, unter Umständen bessere, Optimierungen vornehmen kann.

4.1.1. STL

Zusätzlich zu erwähnen ist die umfangreiche *Standard Template Library* (STL), welche es inzwischen sogar fast vollständig in den C++-Standard geschafft hat. Bei der STL handelt es sich um eine von den meisten Compilern bereitgestellte Code-Bibliothek für häufig verwen-

dete Datenstrukturen und Hilfsroutinen, welche eine wesentliche Erleichterung für jeden C++-Programmierer darstellen. Unter anderem in der STL enthaltene Datenstrukturen sind Listen, Stacks, aber auch Iteratoren für die meisten Datenstrukturen, sowie häufig verwendete Algorithmen wie Quicksort oder Binarysearch. Hinzu kommen vereinheitlichte Ein- und Ausgabemöglichkeiten für verschiedenste Plattformen, sowie die Möglichkeit der parallelen Ausführung von Software mit Hilfe von Threads.

Die STL wurde inzwischen in den Sprachstandard von C++ aufgenommen, so dass jeder standardkonforme Compiler eine Implementierung der STL inkludiert. Mit der STL hebt sich C++ stark von C ab, für welches es keine solche allgemein einsetzbare Codebibliothek gibt. Der Preis für diese Einfachheit ist die Zeit, die ein einzelner Übersetzungsdurchgang benötigt. Da die STL stark auf die namensgebenden *Templates* setzt, welche es ermöglichen eine einzelne Datenstruktur für eine Vielzahl an Datentypen verfügbar zu machen, muss der Compiler diese *Templates* zum Übersetzungszeitpunkt auflösen, ein Vorgang, welcher einen stark negativen Einfluss auf die Übersetzungszeit hat.

4.1.2. Einsatz

Auf Grund der hohen Performance und der umfangreichen Code-Bibliotheken wurde für die Implementierung des Herzstückes dieser Arbeit C++ gewählt. Das bedeutet, dass alle Softwareteile, welche zur Lösung einer einzelnen Probleminstanz gebraucht werden, als ein einzelnes Programm in C++-Quellcode vorliegt. Viele Hilfsroutinen, welche unter anderem dazu geschrieben wurden, neue Instanzen des PCP zu generieren, mehrere VNS-Instanzen gleichzeitig auszuführen und auch Auswertungen zu fertig berechneten Instanzen zu bieten, wurden hingegen in anderen Programmiersprachen geschrieben, welche zwar wesentlich langsamer in ihrer Ausführung sind, dafür aber durch ihre Besonderheiten eine schnelle Programmentwicklung ermöglichen und auch einfach an neue Bedürfnisse angepasst werden können.

4.2. Python

Neben der eigentlichen Applikation mit algorithmischem Kern entstanden im Laufe der Arbeit am Projekt zusätzliche Skripte, um repetitive Tätigkeiten zu beschleunigen. Unter anderem wurde Python zur massenhaften Ausführung des Algorithmus auf verschiedenen Testinstanzen und die anschließende Aufbereitung der Ergebnisse eingesetzt. Außerdem wurde diese Skriptsprache zur Konversion verschiedener Eingabeformate und zur Generierung von Eingabedaten verwendet.

Im Kontrast zur statisch typisierten Sprache C++ steht Python mit seinen dynamischen Eigenschaften, das heißt, dass Variablen grundsätzlich keinen statischen Typ aufweisen. Die so erhöhte Lesbarkeit wurde noch bewusst durch Whitespace-Konventionen und leichtgewichtige Syntax für Kontrollstrukturen und Schleifen erhöht. Außerdem ist die Python-Plattform (inklusive Interpreter) auf unixbasierten Systemen weit verbreitet, bei vielen Distributionen sogar Teil des Grundsystems. Hohe Popularität und gute Lesbarkeit als Design Pattern machen Python zu einer hervorragenden Skriptsprache, weshalb sie auch im Rahmen dieses Projektes in dieser Rolle Einsatz findet.

Während bei der Implementierung der Variablen Nachbarschaftssuche besonders die Performance zur Laufzeit berücksichtigt wurde, stand bei Skripten die Lesbarkeit und einfache Wartbartkeit im Vordergrund. Python passt auch hier ins Bild, da die Sprache üblicherweise interpretiert oder Just-In-Time übersetzt wird, was in der Regel eine höhere Laufzeit mit sich bringt. Dieser Nachteil relativiert sich dadurch, dass Scripts sowieso nicht als laufzeitkritische Teile des Projekts betrachtet werden.

4.3. Make

Bei GNU make handelt es sich um ein Standardwerkzeug der Linux-Software-Entwicklung. Mit Hilfe von make kann die Übersetzung des Quellcodes in den Programmcode automatisiert werden. Dabei stellt make viele verschiedene Konfigurationsmöglichkeiten bereit. Unter anderem können verschiedene Compiler aber auch verschiedene Übersetzungsmodi eingestellt werden. Um ein schnelleres Übersetzen zu ermöglichen werden, bei richtiger Konfiguration, außerdem

nur jene Quellcode-Dateien neu übersetzt, welche seit dem letzten Übersetzungsvorgang verändert haben.

In der Konfigurationsdatei für make, dem so genannten Makefile, können unter anderem mehrere *Targets* angegeben werden, also quasi verschiedene Endprogramme oder verschiedene Übersetzerkonfigurationen für das selbe Programm. Zu diesen *Targets* werden dann Abhängigkeiten definiert welche bestimmen, in welcher Reihenfolge die einzelnen Quelltextdateien, aus denen das gesamte Projekt besteht, übersetzt werden müssen. Impliziert wird durch diese Abhängigkeitsdefinition auch, wann ein Programmteil neu übersetzt werden muss. Bei diesen Abhängigkeiten handelt es sich meist selbst um Dateien, oft Quelltextdateien, welche den den Code für einen *Target* enthalten. GNU make erkennt nun automatisch wenn eine dieser Abhängigkeiten seit dem letzten Programmablauf verändert wurde und wird genau für jenen *Target* einen neuen Übersetzungsvorgang starten.

Durch verschiedene Variablen, welche ebenfalls im Makefile spezifiziert sind, kann außerdem ohne großen Aufwand der Übersetzer, welcher die eigentliche Hauptaufgabe übernimmt, ausgetauscht werden. Die für diese Arbeit verwendeten Compiler werden in Abschnitt 4.4 besprochen. Vereinfacht wird dies durch die beinahe gleiche Befehlssyntax, die von beiden Compilern verwendet wird. Hinzu kommt die Tatsache, dass die von den beiden Compilern erzeugten Dateiformate miteinander kompatibel sind, so dass auch bei uneinheitlicher Übersetzung mit immer wieder wechselnden Übersetzern keine Fehler entstehen.

Außerdem hilfreich ist, dass make auch Skripte oder kleinere Kommandos automatisch ausführen kann. Häufig verwendet wird etwa die Routine zur Bereinigung des Programmverzeichnisses von bei der Übersetzung entstandenen temporären Dateien, die später nicht mehr benötigt werden. Auch nützlich ist etwa die vollständige Entfernung jeder Spur einer Kompilierung, um eine gänzlich neue Übersetzung einzuleiten. Weiters kann mit Hilfe von solchen Skripten das entstandene Programm nach der Übersetzung automatisch auf eine Testmaschine geschickt werden, um die korrekte Funktion des Programmes zu verifizieren.

4.4. Compiler

Der Compiler oder Übersetzer ist dafür verantwortlich, ein Programm von einer Quellsprache in eine Zielsprache zu übersetzen. Im Falle dieser Arbeit war die primäre Quellsprache C++ und die Zielsprache die Maschinensprache des Prozessors, also ein x86-Assembly. Übersetzer sind außerordentlich komplexe Stücke an Software, welche nicht einfach die Befehle eines Programmierers eins zu eins übernehmen, sondern zusätzlich noch auf eventuelle Programmierfehler aufmerksam machen und Optimierungen vornehmen, um eine beschleunigte Programmausführung zu ermöglichen. Da dieser Vorgang sehr zeitaufwendig sein kann ist es wichtig, Werkzeuge wie das in Abschnitt 4.3 besprochene make einzusetzen, um eine vollständige Neuübersetzung bei nur marginalen Änderungen im Quellcode zu verhindern.

Da C++ eine besonders komplexe Sprache mit vielen verschiedenen, aufwendigen Funktionen und Sprachbesonderheiten ist, ist die Übersetzung eines C++-Quellcodes besonders aufwendig. Selbst kleinere C++-Programme können schon mehrere Minuten zum vollständigen Übersetzen brauchen. Dies gilt insbesondere für Quellcode, welche die C++-Funktion der so genannten *Templates* benützt. *Templates* erlauben es, neben anderen Dingen, eine allgemeine Datenstruktur für viele verschiedene Datentypen zu programmieren ohne eine konkrete Implementation für einen bestimmten Datentyp notwendig zu machen. Da auch mehrere *Templates* geschachtelt werden können und dies alles zum Zeitpunkt der Übersetzung expandiert werden muss, um schließlich übersetzt werden zu können, kann bei starker Verwendung von solchen Funktionen die Geschwindigkeit der Übersetzung stark beeinträchtigt werden.

Um diese Probleme mit *Templates* elegant zu umgehen, existiert das Konzept der *Precompiled Headers*. Normalerweise werden Header-Dateien, welche allgemeine Deklarationen von Klassen, verwendeten Datentypen und ähnliches enthalten, zum Übersetzungszeitpunkt in den C++-Quellcode, welcher übersetzt werden soll, komplett kopiert und diese Gesamteinheit als eine Übersetzungseinheit angesehen. Dadurch muss, sollte sich ein Teil der Definitionen innerhalb des C++-Codes ändern, auch die Definition neu übersetzt werden. Mit *Precompiled Headers* wird dieses Problem gelöst, indem schon der Header als zu übersetzende und vom eigentlichen Quelltext abgetrennte Einheit angesehen wird. Der Header, welcher ja unter anderem auch die Deklarationen für *Templates* enthält, muss nur noch übersetzt werden, wenn sich in seinem eigenen Code etwas ändert. Dadurch können die eigentlichen Programmabschnitte, welche in

normalen C++-Quelldateien zu finden sind, wesentlich schneller compiliert werden ohne auf die Vorteile von *Templates* zu verzichten.

4.4.1. GCC

GCC, die GNU Compiler Collection, ist eine Sammlung an Übersetzern, welche alle unter der Open-Source-Lizenz GPL veröffentlicht wurden. GCC wird häufig als der Standard-Compiler unter Linux angesehen und erfreut sich immer noch großer Beliebtheit. Die Sammlung unterstützt viele verschiedene Sprachen, unter ihnen auch C++, sowie mehrere verschiedene Standards einzelner Sprachen. Auf Grund seiner ausgiebigen Testung und Erprobung im alltäglichen Einsatz und nicht zuletzt seiner freien Verfügbarkeit bleibt GCC immer noch einer beliebtesten Compiler weltweit. Dabei ist die Sammlung verschiedener Compiler vor allem im Umfeld der Softwareentwicklung für Linux beliebt, er kann aber auch für die Entwicklung unter Windows oder anderen Betriebssystemen eingesetzt werden.

Ein wichtiges Merkmal von *g++*, dem C++-Compiler in der Sammlung der GCC, ist sein evolutionäres Wachstum seit seiner Entstehung. Da C++ mit den Jahren immer wieder leicht angepasst und um Funktionen erweitert wurde, musste sich auch GCC anpassen, um mit diesen Änderungen Schritt zu halten. Daher wurden im Laufe der Jahre immer wieder Teile von *g++* neu geschrieben, umgebaut oder durch neue Funktionen ergänzt. Daraus ergeben sich aber Konsequenzen, welche unter anderem die Code-Qualität negativ beeinträchtigen. Zwar Implementiert GCC auch den neuesten Standard von C++, allerdings ist die Qualität der Implementierung sehr stark abhängig von dem Alter des entsprechenden Standards. Während ältere Standards bereits ausgiebig getestet wurden, ist bei neuen Standards eine korrekte Implementierung nicht immer gegeben.

Auf Grund dieser gewachsenen Struktur von GCC ist der *g++*-Compiler nicht der schnellste Vertreter seiner Art. Vor allem die Übersetzung von Headern nimmt mit der GCC mehr Zeit in Anspruch als bei der Konkurrenz. Wie viele andere Compiler unterstützt auch *g++* die Übersetzung von Header-Dateien in *Precompiled Headers*. Dazu wird ein GCC-eigenes Format verwendet und *g++* sucht bei einer späteren Einbindung eines Headers in den Quellcode nach einer gleichnamigen Datei mit der Endung „.gch“. Die Verwendung von solchen *Precompiled Headers* beschleunigt die Übersetzung ungemein, weil die Übersetzung des Headers selber einige

Zeit in Anspruch nimmt. Die für diese Arbeit verwendeten *Precompiled Headers* bewegten sich in der Größenordnung von knapp über 100 Megabyte, eine erstaunliche Datenmenge für eine einzige Übersetzungseinheit. Diese Datenmengen sind vor allem den *Templates* geschuldet, welche während ihrer Expansion wesentlich an Speicherbedarf zunehmen.

4.4.2. clang

Bei clang handelt es sich um einen neuen Konkurrenten für GCC am Markt der Open-Source-Compiler. Wie bei GCC handelt es sich bei clang nicht um einen einzelnen Übersetzer für eine einzelne Programmiersprache, sondern viel mehr um eine Sammlung an Übersetzern für vor allem mit C verwandten Sprachen wie C++ und Objective-C. Anders als GCC ist clang aber nicht über mehrere Jahrzehnte angewachsen, das Projekt startete erst vor wenigen Jahren. Auch baut clang nicht von Grund auf auf einen neuen Compiler sondern benützt das Rahmenwerk der LLVM, der *Low Level Virtual Machine*. LLVM ist ein Projekt, welches mit dem Ziel gestartet wurde, Techniken des Just-in-Time-Übersetzens zu den Sprachen C und anderen Low-Level-Sprachen zu bringen. Dazu wird eine eindeutige Zwischensprache zwischen Quell- und Zielsprache definiert, welche dann von LLVM optimiert werden kann. Diese Technik, welche ursprünglich zur Laufzeit eines Programmes zum Einsatz kommen sollte, wurde schließlich so adaptiert, um einen Compiler von eben jener Zwischensprache zur Zielsprache zu bilden.

Clang wurde nun mit dem Ziel geschaffen, einen sauber programmierten, standard-konformen Compiler für den produktiven Einsatz zu schaffen. Dazu wurde zunächst vor allem auf die Entwicklung eines C-Frontends Wert gelegt. Als das C-Frontend einen arbeitsfähigen Zustand erreichte beschloss man, auch C++ als Quellsprache mit einzuschließen. Da C++ eine sehr umfangreiche Sprache mit vielen verschiedenen Spracheigenschaften ist, schreitet die C++-Implementierung wesentlich langsamer voran als jene des C-Zweiges. Inzwischen hat aber auch dieser Entwicklungszweig einen produktiven Zustand erreicht.

Im Vergleich mit GCC bietet clang vor allem den Vorteil der Geschwindigkeit. Ein einzelner Übersetzerlauf dieser Arbeit ist um durchschnittlich 20% schneller als mit GCC und die erzeugten Object-Dateien sind um durchschnittlich 30% kleiner. Dieser Vorteil ist wohl der neueren Programmierung ohne Mitnahme von Altlasten geschuldet. Der Preis für diesen Vorteil ist eine etwas langsamere Programmausführung des fertig übersetzten Programmes, da unter

anderem GCC einige Optimierungsmöglichkeiten nutzt, welche LLVM noch unbekannt sind. Im Allgemeinen sind die Unterschiede zwischen den beiden Compilern aber vernachlässigbar gering.

4.5. DDD

DDD steht für Data Display Debugger und ist ein weiteres Werkzeug zur Softwareentwicklung von GNU. DDD stellt Werkzeuge zum Debuggen, also zur Fehlersuche in Software, zur Verfügung. Um produktiv eingesetzt werden zu können muss eine spezielle Option während der Übersetzung mit GCC gesetzt werden, damit DDD einzelnen Programmschritten die richtigen Quellcodezeilen zuordnen kann. DDD ist eine sehr vielseitige Anwendung mit graphischer Benutzeroberfläche, welche sehr intuitiv und einfach zu bedienen ist und trotzdem viele Optionen bietet. In Abbildung 4.1 wird beispielhaft die Verwendung von DDD im Einsatz als Debugger dargestellt.

Eines der wichtigesten Werkzeuge zur Fehlersuche innerhalb eines laufenden Programmes sind Unterbrechungspunkte, also eine bestimmte Stelle innerhalb des Quellcodes, an dem die Ausführung angehalten werden soll, um den genauen Zustand des Programmes zu untersuchen. DDD stellt sowohl normale Unterbrechungspunkte als auch spezielle Punkte, welche nur unter vom Benutzer definierten Bedingungen auslösen, bereit. Sobald ein solcher Unterbrechungspunkt erreicht ist, kann DDD die Belegung sämtlicher Variablen anzeigen und vergleichen. Dies kann besonders nützlich sein, um ungeplantes Verhalten innerhalb des Progammablaufes zu untersuchen und zu beheben.

Außerdem bietet DDD die Möglichkeit, im Falle eines Programmabsturzes die auslösende Zeile des Quellcodes zu ermitteln und von dort aus auf die Ursachen des Absturzes zu schließen. Dies ist besonders praktisch, da eine genaue Eingrenzung der Absturzursache ansonsten äußerst schwierig und zeitaufwendig ist. Sollte ein Programm schadhaftes oder fehlerhaftes Verhalten an den Tag legen, zum Beispiel einen unberechtigten Speicherzugriff, wird normalerweise das Programm sofort vom Betriebssystem beendet. DDD protokolliert aber jede aufgerufenen Zeile Code mit und kann daher genaue Auskunft geben, in welchem Programmabschnitt der Fehler aufgetreten ist.

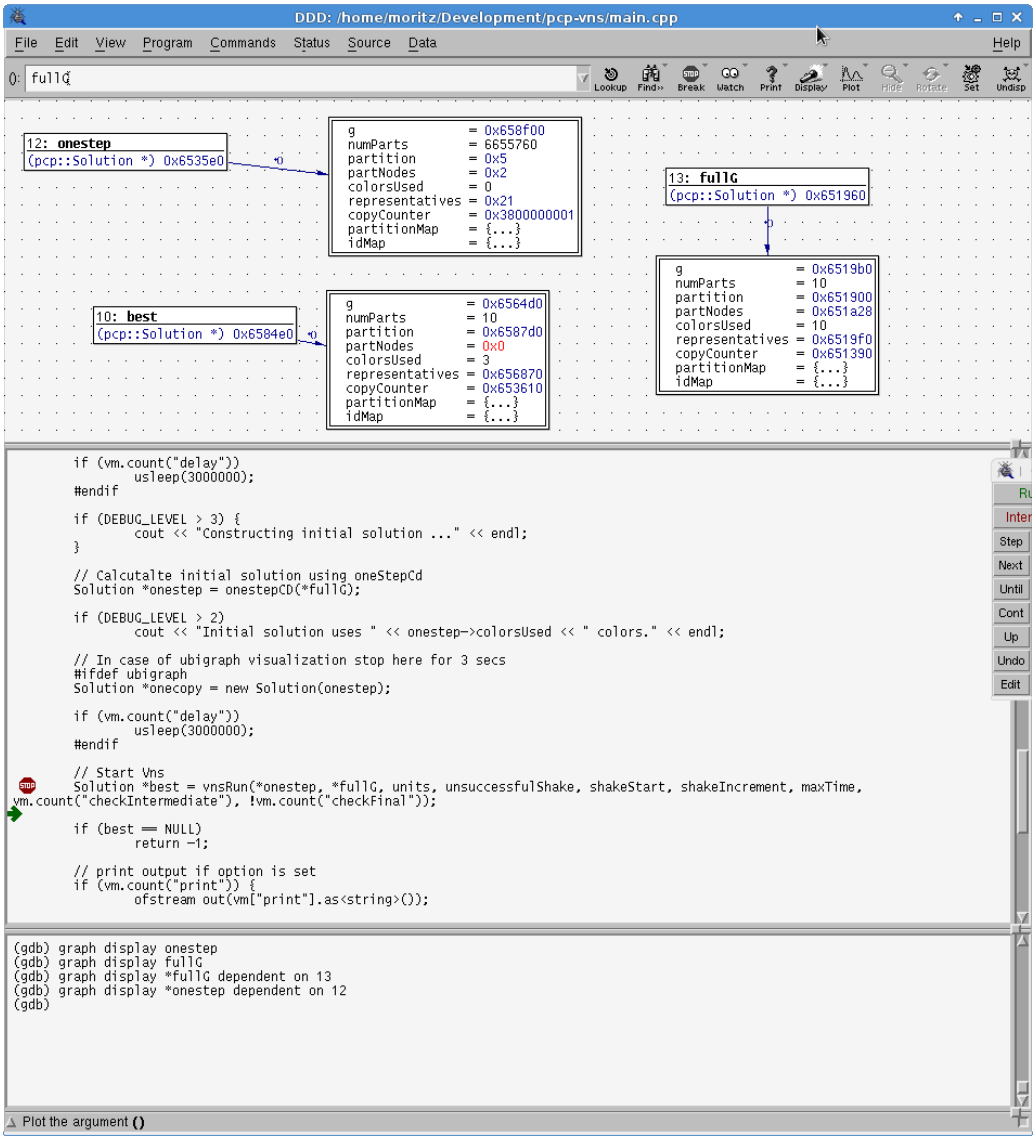


Abbildung 4.1.: Beispiel eines Debuggingvorganges mit Hilfe des Werkzeuges DDD. Klar zu erkennen ist die Dreiteilung der Benutzeroberfläche. Zuoberst befindet sich das Fenster für die Darstellung von Variablen, in der zur Zeit drei Referenzvariablen sowie ihr Inhalt dargestellt sind. Darunter folgt die Fläche für die Darstellung des Quellcodes. Zu sehen ist ein Unterbrechnungspunkt (markiert durch das Stoppschild) sowie die derzeitige Position im Programmablauf (dargestellt durch den Pfeil). Am unteren Ende des Programmfensters ist die Befehlskonsole zu finden, durch die einzelne Befehle, wie zum Beispiel das Starten und Anhalten des Programmes, gesteuert werden können.

Ein weiteres praktisches Werkzeug von DDD ist die Möglichkeit, Pointer, welche auf vom Programm reservierte Speicherbereiche zeigen, genauenstens zu verfolgen. Durch eine übersichtliche graphische Darstellung ist es einfach zu vergleichen, welche Pointer auf ein und den selben Speicherbereich zeigen, was unter Umständen gewollt sein kann oder aber auch einen fehlerhaften Programmablauf hervorruft.

4.6. Boost

Boost, oder auch Boost C++-Libraries, ist eine Codebibliothek welche versucht, Unzulänglichkeiten im Sprachstandard von C++ geschickt zu umgehen und Benutzern ein einfacheres Programmieren zu ermöglichen. Boost besteht aus mehreren unabhängigen Unterbibliotheken, welche zum größten Teil unter der Boost Software Lizenz stehen, welche sowohl eine Open-Source als auch eine kommerzielle Nutzung zulässt. Alle Unterbibliotheken sind portabel designt, um eine Nutzung auf allen Betriebssystemen mit C++-Compiler zu ermöglichen.

Die Entwicklung von Boost begann im Jahr 2000, als Mitglieder des C++-Standardisierungskomitees zusammentraten um Erweiterungen des C++-Standards auf einer breiteren öffentlichen Plattform zu präsentieren. Im Laufe der Entwicklung wurden immer wieder auch ehemals kommerzielle Produkte von Firmen in Boost eingegliedert, wie etwa die Graphikbibliothek GIL, welche ursprünglich von Adobe stammt, inzwischen aber in den Boost-Stamm aufgenommen wurde.

Boost erweitert den Umfang an vorprogrammierten Datenstrukturen und auch Algorithmen und fügt bestehende Strukturen nahtlos ein. Dabei versucht Boost immer eine möglichst einfache und klar verständliche Schnittstelle zu bieten, welche transparent mit Datenstrukturen etwa aus der STL umgeht, welche in Abschnitt 4.1.1 besprochen wurden. Von Boost geschaffene Datenstrukturen sind sehr ähnlich zu jenen aus der STL zu verwenden und bieten wie jene auch einfache Möglichkeiten des Zugriffes auf Datenelemente über Iteratoren.

Des Weiteren bietet Boost neue Möglichkeiten im Umgang mit Strings, also Zeichenketten, sowie weitere nützliche Hilfsfunktionen, welche von den meisten Programmen häufig gebraucht werden. Außerdem bietet Boost Algorithmen und Datenstrukturen für mathematische Berechnungen, die über triviale Aufgaben wie addieren oder subtrahieren von einfachen Zahlen hin-

ausgehen. So bietet Boost eine Unterbibliothek für Statistik und auch für Graphen, welche in dieser Arbeit zum Einsatz kam.

Außerdem wurden in dieser Arbeit viele Hilfsroutinen, welche von Boost bereitgestellt werden, verwendet: Auch kamen mehrere Datenstrukturen aus Boost-Unterbibliotheken zum Einsatz.

4.6.1. graph

Da es sich bei dem PCP um ein Optimierungsproblem aus dem Umfeld der Graphentheorie handelt, ist natürlich eine Graphenstruktur innerhalb des Programmes unerlässlich. Boost bietet eine ganze Unterbibliothek, welche rein graphenbezogene Datenstrukturen und Algorithmen bereitstellt. Diese Graphenbibliothek ist gut getestet, vielseitig, stark anpassbar, schnell und einfach einzusetzen und war daher die ideale Wahl für die Lösung des PCP.

Boost bietet zwei grundsätzlich unterschiedliche Implementierungen eines Graphen an. Bei der ersten Variante werden alle Kanten in einer Liste oder einer ähnlichen Datenstruktur gespeichert, bei der zweiten Variante wird eine Matrix verwendet, um die Beziehungen zwischen zwei Knoten darzustellen. Während die zweite Variante für sehr dichte Graphen einige Vorteile bietet, ist die gesamte Datenstruktur bei weitem nicht so flexibel wie die Variante mit einer Listenstruktur. In der von Boost verwendeten Implementierung dieser Adjazenzmatrix werden viele praktischen Funktionen, um einen Graphen zu manipulieren, nicht unterstützt. So ist es zum Beispiel nicht möglich, einen Knoten nachträglich wieder zu entfernen.

Daher fiel die Wahl auf eine Adjazenzliste, welche den Graphen für diese Arbeit bereitstellte. Auch hier gibt es mehrere verschiedene Konfigurationsmöglichkeiten, etwa die genaue Art der Speicherung von Knoten und Kanten. Für die Speicherung der Knoten wurde ein Vektor gewählt, ein zusammenhängender Speicherbereich, welcher bei Bedarf vergößert oder verkleinert werden kann. Dies bietet mehrere Vorteile, etwa einen direkten Zugriff auf einen Knoten über einen Index, welcher in Form einer einfachen ganzen Zahl leicht zu benützen war. Dies bedeutet, dass zu jedem Zeitpunkt direkt auf einen bestimmten Knoten zugegriffen werden kann. Da die Anzahl der Knoten quasi immer konstant bleibt mit Ausnahme des erstmaligen Aufbaus des Problemgraphens und dem Errechnen der Initiallösung, werden die Nachteile einer vektorbasierten Speicherung der Knoten bei weitem von den Vorteilen aufgewogen.

Da im Rahmen der Variablen Nachbarschaftssuche mehrere Nachbarschaften immer wieder eine Veränderung der Adjazenzen vornehmen, fiel die Wahl der Datenstruktur für die Speicherung der Kanten zunächst auf eine Liste. Zu jedem Knoten wird eine Liste der Nachbarknoten gespeichert, woraus dann die Kanten gebildet werden. Da das Löschen aus und das Einfügen in eine Liste in konstanter Laufzeit möglich ist, wurden mit der Listenstruktur die besten Ergebnisse erwartet. Ausgiebige Tests ergaben jedoch, dass auch für die Speicherung der Kanten ein Vektor besser geeignet war. Da beim Löschen eines Knotens zumeist gleich alle adjazenten Kanten gelöscht werden, reduzierte sich der Vorteil einer Listenstruktur und das Einfügen ist auch in einen Vektor in zumeist konstanter Zeit möglich.

Eine weitere Konfigurationsmöglichkeit betrifft das Verhalten des Graphen beim Hinzufügen von Kanten. Es gibt gerichtete und ungerichtete Graphen, wobei gerichtete Graphen Kanten besitzen, welche quasi als Einbahn fungieren, sie haben einen Anfangsknoten und einen Zielknoten und stellen nicht automatisch auch eine Verbindung vom Zielknoten zum Anfangsknoten her. Im Gegensatz dazu stehen die ungerichtete Graphen, bei denen eine Kante automatisch eine Beziehung in beide Richtungen darstellt. Auf Grund der Eigenschaften des PCP fiel die Wahl auf einen ungerichteten Graphen, da kein Bedarf an gerichteten Kanten bestand.

Eine weitere von Boost bereitgestellte Funktion des Graphen ist die Speicherung von Eigenschaften im direkten Zusammenhang mit Kanten und Knoten. Da jeder Knoten im PCP zu einer Partition zugeteilt wird, kann über diesen Mechanismus einfach und schnell von einem Knoten auf die Partition geschlossen werden. Diese Eigenschaften können außerdem dazu eingesetzt werden, den originalen Index eines Knotens zu speichern. Da Boost bei der Entfernung eines Knotens automatisch den Index der nachfolgenden Knoten um eins dekrementiert, kann normalerweise nicht mehr auf den Index des Ursprungsknotens geschlossen werden. Durch die Speicherung des Index in einer separat von Boost bereitgestellten, mitschrumpfenden Datenstruktur kann aber immernoch eindeutig auf den Originalknoten geschlossen werden.

Eine einfache Möglichkeit der Ausgabe wird ebenfalls von Boost gestellt. Boost bietet eine einzelne Funktion, welche einen Boost-Graphen im Graphviz-Format ausgibt. Dadurch ergibt sich eine schnelle Möglichkeit, die Richtigkeit einer Lösung zu kontrollieren. Gerade bei frühen Versuchen der Nachbarschaftssuche wurde viel mit diesem Werkzeug gearbeitet, um die Ergebnisse zu verifizieren.

4.6.2. Hilfsroutinen

Neben Graphen wurden auch einige von Boosts Hilfsfunktionen benutzt. Um bei der Lösung des PCP möglichst flexibel zu sein wurde eine Vielzahl an Variablen verwendet, um bestimmte Parameter der Nachbarschaftssuche einfacher ändern zu können. Dazu wurde angedacht, die Terminalparameter, als jene Parameter, welche beim Aufruf des Programmes über die Kommandozeile mitgegeben werden, zu verwenden. In einem ersten Versuch wurde ein einfaches Stringparsing betrieben, welches sich aber schnell als fehleranfällig herausstellte und zusätzlich noch zu unschönem Code führte.

Genau für den Zweck der Programmargumente bietet Boost eine eigene Unterbibliothek, welche einem alle Aufgaben aus dem Bereich des Parsings übernimmt und dies außerdem noch in einer eleganten Art und Weise für den Programmierer anbietet. Durch einen Bibliotheksaufruf können einfach neue Befehle angegeben werden und diesen Befehlen gleich ein bestimmter Datentyp zugeordnet werden. Es besteht aber nicht nur die Möglichkeit, einen bestimmten Datentypen zuzuordnen, sondern ein bestimmtes Argument gleich an eine Variable zu binden, sodass diese Variable automatisch von Boost gesetzt wird. Des Weiteren kann einem Argument automatisch ein Standardwert zugewiesen werden welcher automatisch angenommen wird, sollte der Benutzer bei der Ausführung des Programmes dieses Argument nicht explizit angeben.

Um eine möglichst genaue Zeitmessung für die statistische Auswertung der Ergebnisse zu erlangen, wurde die Zeitnehmungsbibliothek von Boost herangezogen. Boost stellt einfache Zeitnehmer bereit, um möglichst genaue Messergebnisse zu bekommen, welche im Millisekundenbereich liegen. Dies ist manchmal gar nicht so einfach, da etwa die normalen Zeitfunktionen nur auf Sekundenbasis arbeiten und eine genauere Zeitmessung zumeist über die abgelaufenen CPU-Zyklen erfolgt.

4.7. Valgrind

Valgrind ist, wie schon DDD, ein Werkzeug zur Fehlersuche in laufenden Programmen. Dabei spezialisiert sich Valgrind auf die korrekte Verwendung und Freigabe von Speicher. Mit Hilfe von Valgrind ist es möglich, Speicherlecks aufzuspüren und zu schließen. Dafür überwacht Valgrind

den Programmablauf und sämtliche Speicherzugriffe und gibt nach Beendigung des Programmes eine genaue Aufzählung an vergessenen Speicherbereichen und nicht sauber gelöschten Objekten (siehe Listing 4.1) aus.

Zum Zweck der Informationssammlung und zur besseren Darstellung der Ergebnisse sollte das eigene Programm wieder mit der Debug-Option des Compilers übersetzt werden. Dadurch kann Valgrind unter anderem die Zeile anzeigen, in der das nicht gelöschte Objekt erzeugt wird und Informationen geben, ab wann ein Speicherbereich verloren gegangen ist. Außerdem ist zu beachten, dass durch diese Mitprotokollierung des Programmablaufes die Geschwindigkeit desselben stark beeinflusst wird. Daher ist es notwendig, bei Vorgängen, welche innerhalb des Programmes durch eine Ausführungszeit beschränkt werden, diese Zeit anzupassen, um eine korrekte und ausreichend lange Ausführung des Programmes zu gewährleisten.

Wenn nun innerhalb des Programmes ein Stück Arbeitsspeicher reserviert wird, merkt sich Valgrind diesen Vorgang und protokolliert mit, wieviele Zeiger auf diesen Speicherbereich verweisen. Sollte nun irgendwann kein einziger Zeiger mehr diesen Speicherbereich referenzieren, es wurden also quasi alle Variablen, die diese Information enthielten, verworfen oder überschrieben und der Speicherbereich wurde nicht wieder freigegeben, so weiß Valgrind, dass dieser Speicher für das Programm verloren gegangen ist. Sollte sich dieser Vorgang wiederholen entsteht ein immer größer werdender Pool aus nicht freigegebenen Speicherbereichen, welche nicht gelöscht wurden. Nachdem eine Low-Level-Programmiersprache wie C++ keine richtige automatisierte Speicherverwaltung wie andere Sprachen, z. B. Java und C# hat muss der Programmierer selbst für die Freigabe von reservierten Speicherbereichen sorgen. Daher ist es essentiell für gute Programmierung, alle erzeugten Objekte nach ihrer Verwendung wieder freizugeben, um so den exzessiven und sinnlosen Gebrauch von Hauptspeicher zu verhindern.

Listing 4.1: Ausgabe von `valgrind -v -leak-check=full ...` bei vorhandenem Speicherleck.

```
1 ==3504== HEAP SUMMARY:
2 ==3504==     in use at exit: 72 bytes in 3 blocks
3 ==3504==   total heap usage: 3,436 allocs, 3,433 frees, 169,780 bytes
     ↪ allocated
4 ==3504==
5 ==3504== Searching for pointers to 3 not-freed blocks
6 ==3504== Checked 221,528 bytes
7 ==3504==
8 ==3504== 72 bytes in 3 blocks are definitely lost in loss record 1 of 1
```

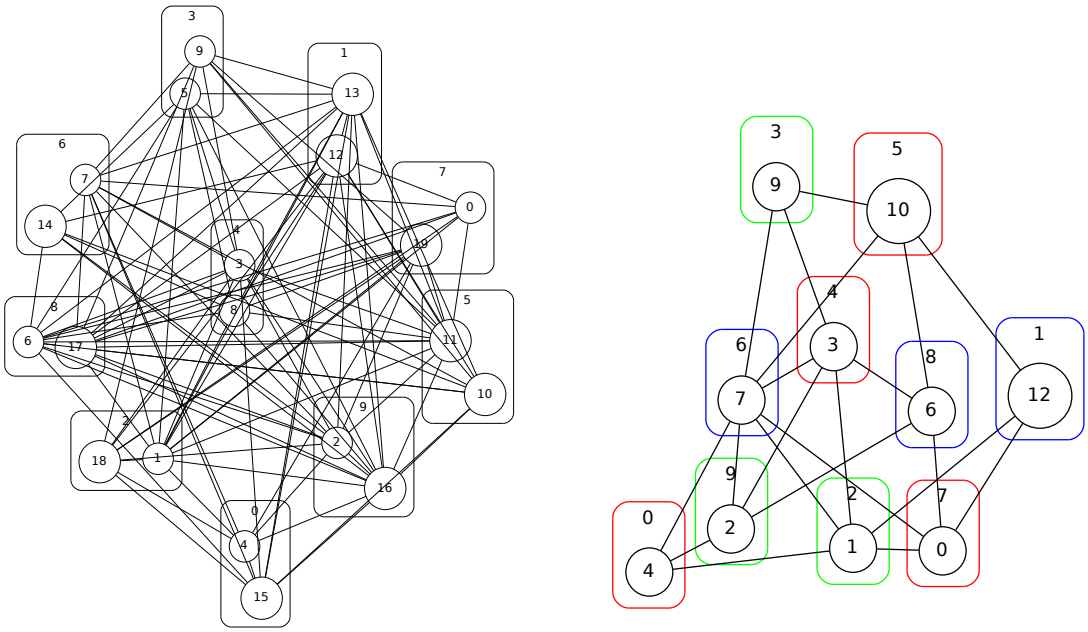
```
==3504==    at 0x4C2C7A7: operator new(unsigned long) (in
   ↪ /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3504==    by 0x421D4C: pcp::vnsRun(pcp::Solution&, pcp::Solution&,
   ↪ std::string, int, int, int, int, bool, bool) (vns.cpp:111)
==3504==    by 0x40925E: main (main.cpp:132)
==3504==
==3504== LEAK SUMMARY:
==3504==    definitely lost: 72 bytes in 3 blocks
==3504==    indirectly lost: 0 bytes in 0 blocks
==3504==      possibly lost: 0 bytes in 0 blocks
==3504==    still reachable: 0 bytes in 0 blocks
==3504==         suppressed: 0 bytes in 0 blocks
==3504==
==3504== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
--3504--
--3504-- used_suppression:      2 dl-hack3-cond-1
==3504==
==3504== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

Da die VNS in der Theorie eine endlose Schleife der Verbesserung darstellt und daher auch das Programm eine prinzipiell unbegrenzte Laufzeit besitzen sollte, ist die Frage der Speicherverwaltung essentiell. Selbst kleinste Speicherlecks könnten, über einen langen Zeitraum betrachtet, zum Absturz des Programmes führen. Daher bietet Valgrind eine einfach zu nutzende und übersichtliche Möglichkeit, die Korrektheit der Speicherverwaltung innerhalb des laufenden Programmes zu testen.

4.8. Graphviz

Graphviz ist eine Sammlung an Software-Werkzeugen zur Visualisierung von Graphen. Jedes Werkzeug aus der Softwaresammlung liest dabei eine im sogenannten Graphviz-Format geschriebene Datei ein und transformiert sie auf bestimmte Art und Weise. Die Funktionen umfassen von der Entfernung von doppelten Kanten bis zur visuellen Ausgabe in Form einer PDF oder PostScript-Datei viele nützliche Bearbeitungsmethoden für Graphen.

Da die im Abschnitt 4.6.1 besprochene Implementation eines Graphen-Datentypes durch die Boost-Bibliothek eine automatische Ausgabe eines Graphen in das Graphviz-Dateiformat ermöglicht, war eine Verwendung dieser Werkzeuge sehr naheliegend. Durch die vielen verschiedenen Programme, welche von Graphviz bereitgestellt werden, war es ein Leichtes, eine einfache Möglichkeit der visuellen Kontrolle der Arbeitsergebnisse zu implementieren. Im Gegensatz



(a) Konfliktgraph der Instanz selbst

(b) Eine mögliche Lösung (eingefärbter und vereinfachter Konfliktgraph)

Abbildung 4.2.: Beispiel der Darstellung von Probleminstanzen und -lösungen durch Graphviz, anhand von `Table2_random_instances/n20p5t2s5.pcp`.

zur im Abschnitt 4.9 eingesetzten Echtzeitdarstellung der Vorgänge während des Ablaufes des Programmes wurde die Werkzeugsammlung dazu eingesetzt, die Eingabe mit der Ausgabe zu vergleichen und zu verifizieren, dass eventuelle Fehler in der internen Prüfungsroutine des Programmes nicht dazu führten, dass falsche Lösungen plötzlich als zulässig angesehen würden. Konkret trägt die bildliche Darstellung der Instanz (siehe Abbildung 4.2a) dazu bei, eine bessere Vorstellung des Konfliktgraphen zu gewinnen, um die Schritte zur Lösung (Abbildung 4.2b) leichter nachvollziehen zu können.

Eingeschränkt wurde der Einsatz von Graphviz durch die Tatsache, dass bei großen Probleminstanzen natürlich auch die Lösungsgröße entsprechend anwächst und daher ein händischer Vergleich nur noch schwer bis gar nicht möglich ist. Daher wurde Graphviz vor allem dazu eingesetzt, die Selbsttestung des Programmes zu überprüfen und mehrere Szenarien zu kreieren, welche zu Problemen im Programmablauf führen könnten. Mit diesen Szenarien konnte dann verifiziert werden, dass das Programm in der Lage ist, ungültige Lösungen von selbst auszuschließen.

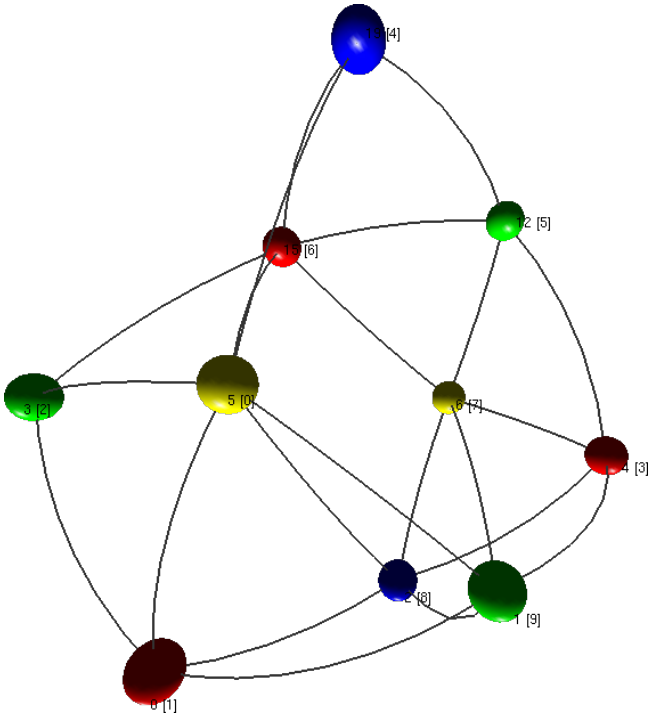


Abbildung 4.3.: Ein Beispiel der graphischen Ausgabe der Ubigraph-Software. Zu sehen ist die Ausgabe der finalen Lösung der VNS, welche bereits mehrere Nachbarschaften durchlaufen hat.

4.9. Ubigraph

Bei *Ubigraph*[1] handelt es sich um eine Plattform zur visuellen Darstellung von Graphen. Die Plattform besteht einerseits aus einem Server, welcher Befehle entgegen nimmt und aus diesen Befehlen eine dreidimensionale Darstellung des Graphen ausgibt und andererseits aus einer Vielzahl an Schnittstellen für alle nur erdenklichen Programmiersprachen. Mit Hilfe von Ubigraph ist es möglich, die Vorgänge innerhalb der VNS, sowie beim Aufbau der Initiallösung zeitnah mitzuverfolgen und gleichzeitig eine ansehnliche dreidimensionale Darstellung des Graphen zu erlangen.

Bei der Serversoftware handelt es sich um ein Stück proprietären Code, welcher per XML-RPCs angesprochen werden kann. Der Server nutzt OpenGL, um die gezeichneten Graphen per Grafikkarte darzustellen. Als Schnittstelle dienen *Remote Procedure Calls* (RPC), welche per *Hyper Text Transfer Protocol* (HTTP) an den Server übermittelt werden. Der Name XML-RPC stammt von der Auszeichnungssprache *Extensible Markup Language* (XML), was soviel heißt

[1] `http://ubietylab.net/ubigraph/`

wie Erweiterbare Auszeichnungssprache, welche als Anfrage- und Antwortsprache verwendet wird. Die Daten einer solchen Anfrage werden von der ausgeführten Software per HTTP-POST-Methode an den Server übermittelt, welcher auch wieder eine XML-Meldung über Erfolg oder Misserfolg der Operation zurückliefert.

Da es sich bei XML-RPC um einen textbasierten Standard handelt, ist vor allem die Integration von Ubigraph in Skriptsprachen besonders einfach. Für viele Sprachen werden bereits Schnittstellen mitgeliefert, welche eine noch einfachere Verwendung von Ubigraph ermöglichen. Der Quellcode dieser Schnittstellen ist als Open-Source-Software zur Verfügung gestellt und kann daher ebenso eingesehen werden wie die Spezifikation für die Server-Schnittstelle per RPC. Da für C++ selbst keine eigene Schnittstelle vorhanden war, wurde auf die Schnittstelle für C zurückgegriffen, welche sich wiederum einiger anderen Bibliotheken aus dem XML und RPC-Umfeld bedient. Da C++ die Möglichkeit bietet, nahtlos mit C-Code umzugehen, war die Integration von Ubigraph keine große Herausforderung. Die Ubigraphschnittstelle für C bietet einige einfache Optionen zum Zeichnen von Knoten, Kanten beziehungsweise der Manipulation der Eigenschaften derselben, wie z. B. Farbe, Form und Beschriftung.

5. Projektmanagement

Im Folgenden soll erläutert werden, wie diese Arbeit als Projekt organisiert und durchgeführt wurde. Dabei soll ein Überblick über die Projektstruktur sowie den Ablauf der einzelnen Arbeitsschritte gegeben werden sowie im speziellen auf die Implementierung der beschriebenen Lösungsansätze und Algorithmen eingegangen werden, da genau dieser Aspekt wohl klar die meiste Arbeitszeit in Anspruch nahm.

Es folgt eine übersichtsartige verbale Beschreibung der Arbeitsphasen und -schritte.

5.1. Projektstruktur

Um das Management während dem Projektverlauf zu erleichtern, wurde das Projekt in vier Teilbereiche, welche wiederum aus Arbeitspaketen bestehen, gegliedert, die in Gesamtheit alle Aufgaben, die im Rahmen des Projekts erledigt wurden, abdecken.

Die Darstellung des Projektstrukurplans (Abbildung 5.1) und das Gantt-Diagramm (Abbildung 5.2) dienen zur besseren Veranschaulichung.

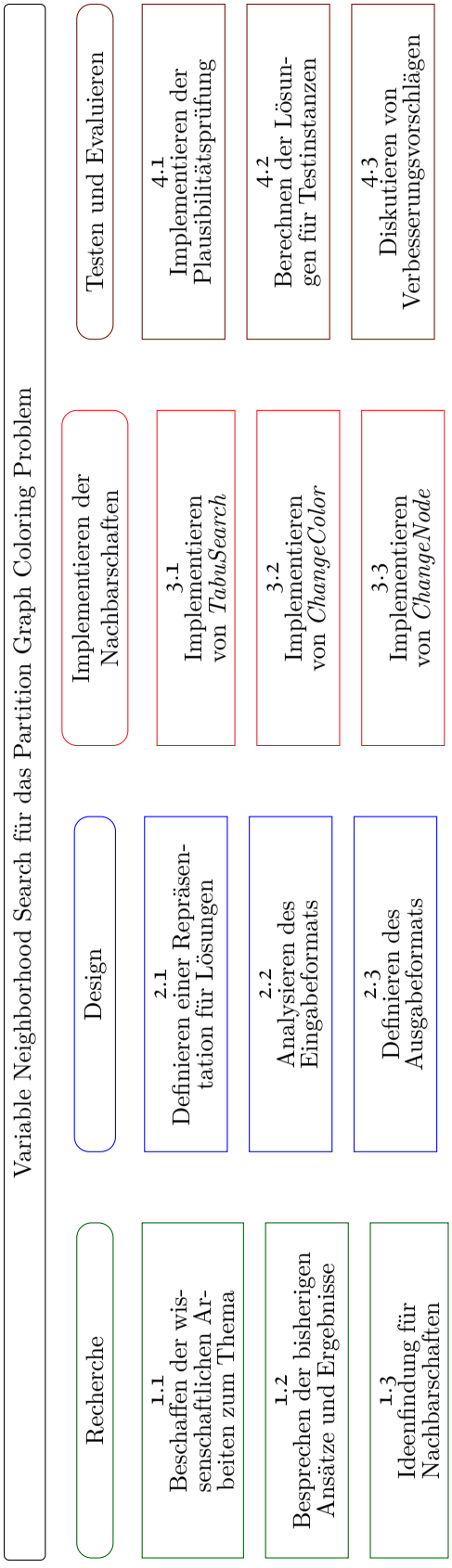


Abbildung 5.1.: Projektstrukturplan zur Übersicht über die Teilaufgaben innerhalb des Projektablaufs.

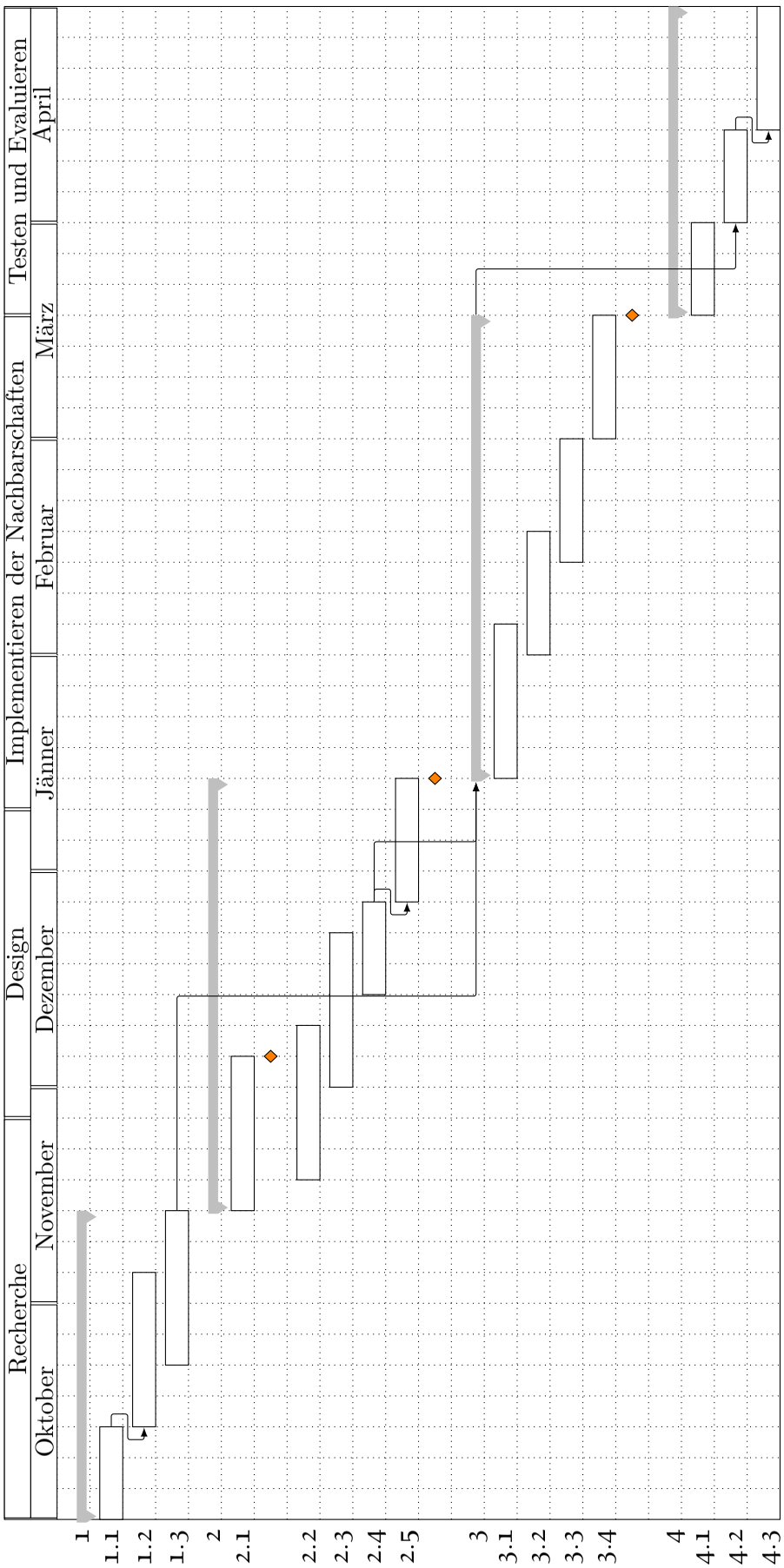


Abbildung 5.2.: Gantt-Diagramm mit Abhängigkeiten zwischen Arbeitspaketen und Meilensteinen

5.1.1. Recherche

Um nicht bereits zu Projektbeginn die Fehler anderer erneut zu begehen, erschien eine ausgedehnte Recherchephase sinnvoll. Rückblickend zeigte sich diese Phase als besonders hilfreich, um mit der komplexen Materie vertraut zu werden.

1.1 Beschaffung der wissenschaftlichen Arbeiten zum Thema

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Lorenz Leutgeb Moritz Wanzenböck	Moritz Wanzenböck	ja

Die wissenschaftliche Komponente des Projekts erforderte die umfangreiche Auseinandersetzung mit bereits abgeschlossenen Arbeiten zu ähnlichen Lösungsansätzen der gleichen Problemstellung. Am wichtigsten hierbei ist das Identifizieren von wichtigen Erkenntnissen, die sich vorteilhaft für die eigene Arbeit nutzen lassen. So konnten wir beispielsweise schnell die Konstruktionsheuristik *onestepCD* (siehe 2.2) anderen Varianten vorziehen, da diese in früheren Evaluierungen die besten Ergebnisse berechnete.

1.2 Besprechung der bisherigen Ansätze und Ergebnisse

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Lorenz Leutgeb Moritz Wanzenböck	Moritz Wanzenböck	ja

Abgesehen von der Wahl von *onestepCD* (siehe Abschnitt 2.2.1) aufgrund der offensichtlichen Vorteile wurde vor allem die Tabu-Suche von Noronha und Ribeiro (2006) genauer analysiert, um daraus eventuell Schlüsse für leistungsfähige Nachbarschaften ziehen zu können.

1.3 Ideenfindung für Nachbarschaften

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Lorenz Leutgeb Moritz Wanzenböck	Moritz Wanzenböck	ja

Zum Abschluss der Recherchephase entwickelte sich bereits ein Gespür für die Struktur des Problems, sodass Ideen für eigene Nachbarschaften aufkamen.

5.1.2. Design

Nach einer Umfangreichen Recherche und Auseinandersetzung mit der Problemstellung ging das Projekt in die Designphase über. Hier galt es sicherzustellen, wie die Implementierung der Nachbarschaften erfolgen soll und außerdem Schnittstellen des Programms zur Außenwelt klar zu definieren.

2.1 Definieren einer Repräsentation für Lösungen

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Lorenz Leutgeb	Moritz Wanzenböck	nein
Moritz Wanzenböck		

2.2 Analyse des Eingabeformats

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Lorenz Leutgeb	Lorenz Leutgeb	nein
Moritz Wanzenböck		

Eine wichtige Eingabeschnittstelle des Programms ist das Einlesen von Probleminstanzen über die Standardeingabe. Hier wurde das Parsing zweier verschiedener Quellformate implementiert. Einerseits das `.pcp`-Format, in welchem ein Großteil der Instanzen vorlagen.

2.3 Festlegung des Ausgabeformats

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Lorenz Leutgeb	Lorenz Leutgeb	nein
Moritz Wanzenböck		

Um die Ergebnisse der Berechnungen leicht auswerten zu können, wurde die Ausgabe aller Daten im JSON-Format beschlossen.

2.4 Wahl von Codebibliotheken, Compiler, etc.

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Lorenz Leutgeb Moritz Wanzenböck	Moritz Wanzenböck	nein

Bevor die Entwicklung des Programms tatsächlich beginnen konnte wurde eine Codebibliothek gewählt, welche zusätzlich Zeit sparen und Fehlerquellen eindämmen sollte. Weitere Details siehe 4.6

2.5 Implementieren einer Visualisierung

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Lorenz Leutgeb Moritz Wanzenböck	Lorenz Leutgeb	ja

Um die Arbeitsweise des Programms möglichst anschaulich darstellen zu können, wurde eine Visualisierung mit Ubigraph implementiert. Dies ermöglicht eine Überwachung der Arbeitsschritte einzelner Nachbarschaften, sodass auch Dritte Einsicht in den Ablauf gewinnen.

5.1.3. Implementieren der Nachbarschaften

3.1 Implementierung von TabuSearch

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Moritz Wanzenböck	Moritz Wanzenböck	nein

Als erstes wurde die Tabu-Suche implementiert, da eine ausführliche Beschreibung des Verfahrens (zu finden in Abschnitt 1.3.2) vor lag.

3.2 Implementieren von ChangeColor

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Moritz Wanzenböck	Moritz Wanzenböck	nein

Die Nachbarschaft *ChangeColor* (ausführlich beschrieben in Abschnitt 2.3.1) ließ sich einfach implementieren.

3.3 Implementieren von ChangeNode

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Moritz Wanzenböck	Moritz Wanzenböck	nein

Da es sich bei *ChangeNode*, erläutert in Abschnitt 2.3.2, um eine kompliziertere Nachbarschaft handelt, wurde sie als eine der letzten implementiert.

3.4 Implementieren von DSATUR

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Lorenz Leutgeb	Lorenz Leutgeb	nein

Die Implementierung von *DSATUR* konnte aufgrund der Erfahrungen mit anderen Nachbarschaften schnell umgesetzt werden.

5.1.4. Testen und Evaluieren

4.1 Implementieren einer Plausibilitätsprüfung

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Moritz Wanzenböck	Moritz Wanzenböck	nein

Um sicher zu gehen, dass die von den Heuristiken berechneten Lösungen plausibel sind, also keine Konflikte oder unerwünschte Verbindungen enthalten, müssen diese überprüft werden. Da dies mit steigender Instanzgröße nicht von Hand möglich ist, wurde ein simpler Plausibilitätscheck implementiert.

4.2 Berechnung der Lösungen für Testinstanzen

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Lorenz Leutgeb	Lorenz Leutgeb	ja

Risiko	Eintreten p	Ausmaß s	Priorität P
Untergang der Welt mit Ende des 13. Baktun	0.5	1	0.5
Fehlende Vergleichsmöglichkeit der Ergebnisse	0.25	0.75	0.1875
Fehlende Kenntnisse in C++	0.2	0.8	0.16
Datenverlust	0.1	0.9	0.09

Tabelle 5.1.: Übersicht über identifizierte Risiken

Um die Ergebnisse des in dieser Arbeit beschriebenen Ansatzes vergleichen zu können, wurden Testinstanzen herangezogen, welche als Richtwert dienen können.

4.3 **Diskutieren von Verbesserungsvorschlägen**

Mitarbeit	Verantwortlichkeit	Betreuer informiert
Lorenz Leutgeb	Lorenz Leutgeb	ja
Moritz Wanzenböck		

Nach Abschluss des Projekts ist davon auszugehen, dass eine weitere Verbesserung der Ergebnisse möglich ist. In einer abschließenden Diskussion ging es darum, wie dieser Weg weiter verfolgt werden könnte.

5.2. Risikoanalyse

Wie in Tabelle 5.1 ersichtlich ging von dem befürchteten Weltuntergang mit dem Beginn des nächsten *Baktun*-Zyklus im Kalendersystem der Maya-Hochkultur eine große Gefahr für die Erfüllung der Projektziele aus, doch die präventive Verbarrikadierung des Projektteams am 21.12.2012 konnte durchgehende Arbeit (zu diesem Zeitpunkt an der Implementierung) gewährleisten und erwies sich im Nachhinein als nicht zwingend nötig, da beim Verlassen des Hochsicherheitssystems am 22.12. keine dem Ereignis zuordenbaren größeren Infrastrukturschäden im Großraum Wien bemerkt wurden.

Das Absichern gegen weitere Risikoszenarien konnte aufgrund der weit höheren Priorität im Laufe der Vorbereitungen auf den Untergang der Welt nicht gewährleistet werden.

5.3. Versionskontrolle

Eine der ersten Entscheidungen im Laufe der Arbeit am Projekt, wahrscheinlich sogar schon vor der Wahl des Arbeitstitels, war die Auswahl eines geeigneten Versionskontrollsystems. Wie es die Teammitglieder für kleinere Projekte gewohnt waren, begann die Entwicklung in einem von einem Cloud Service bereitgestellten, synchronisierten Ordner. Doch schon nach wenigen Tagen war klar, dass eine für das Projektteam derart wichtige Arbeit ein voll ausgebautes und dezidiertes Versionskontrollsystem benötigt. Schnell fiel die Entscheidung auf *Git*[1], ein System, dass mitunter von den Entwicklern der größten Projekte der *Free und/oder Open Source Software* eingesetzt wird.

Dabei beeindruckt Git vor allem durch den geringen Mehraufwand, um die Versionskontrolle zu pflegen. Weiters wurde es so möglich, komplett unabhängig an Dateien zu arbeiten und Änderungen im Nachhinein einfach zusammenzuführen.

5.4. Continuous Integration

Als zusätzliches Feedback-Element wurde in diesem Projekt auf *Continuous Integration* gesetzt. Das bedeutet, dass sobald Änderungen am Code in der Versionskontrolle eingecheckt und hochgeladen wurden, ein Webservice zur Verfügung stand, um die neue Version zu kompilieren und zu testen. Bei Fehlschlagen eines solchen Builds wurde das Projektteam instantan informiert, um möglichst rasch eine Fehlerbehebung zu beginnen.

Dies ermöglicht insbesondere auch die Überprüfung, ob der Code sich mit GCC genauso wie mit clang kompilieren lässt und bietet außerdem einen guten Richtwert der im Projektverlauf aufgetretenen Fehler.

[1] `https://git-scm.org/`

Literaturverzeichnis

Chen, Yin-Yann u. a. (2013). „A hybrid approach based on the variable neighborhood search and particle swarm optimization for parallel machine scheduling problems—A case study for solar cell industry“. In: *International Journal of Production Economics* 141.1. Meta-heuristics for manufacturing scheduling and logistics problems, S. 66–78. ISSN: 0925-5273. DOI: 10.1016/j.ijpe.2012.06.013. URL: http://www.sciencedirect.com/science/article/pii/S0925527312002411.

Frota, Yuri u. a. (2010). „A branch-and-cut algorithm for partition coloring“. In: *Networks* 55.3, S. 194–204. ISSN: 00283045. DOI: 10.1002/net.20365. URL: http://doi.wiley.com/10.1002/net.20365.

Hoshino, Edna A., Yuri A. Frota und Cid C. de Souza (2011). „A branch-and-price approach for the partition coloring problem“. In: *Operations Research Letters* 39.2, S. 132–137. ISSN: 0167-6377. DOI: 10.1016/j.orl.2011.02.006. URL: http://www.lix.polytechnique.fr/ctw09/ctw09-proceedings.pdf#page=199.

Li, Guangzhi, Rahul Simha und Mary Williamsburg (2000). „The Partition Coloring Problem and its Application to Wavelength“. In: *Proceedings of the First Workshop on Optical Networks*.

Noronha, Thiago F. und Celso C. Ribeiro (2006). „Routing and wavelength assignment by partition colouring“. In: *European Journal of Operational Research* 171.3, S. 797–810. ISSN: 03772217. DOI: 10.1016/j.ejor.2004.09.007. URL: http://linkinghub.elsevier.com/retrieve/pii/S0377221704005867.

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Codeverzeichnis

Appendix

A. Ergebnisse

Mit Ende der Arbeit an dem Projekt im Rahmen der Diplomarbeit wurden in Tabelle A.1 ersichtliche Ergebnisse auf dem Berechnungsnetzwerk der Arbeitsgruppe für Algorithmen und Datenstrukturen des Instituts für Computergraphik und Algorithmen an der Technischen Universität Wien erzielt.

Die Instanzen stammen von der genannten Arbeitsgruppe und sind mit jenen von Noronha und Ribeiro, 2006 vergleichbar.

Das Abbruchkriterium der VNS für diese Ergebnisse war ein Zeitlimit von 600s oder 5000 fehlgeschlagene „Schüttelversuche“.

Legende

Instanz Name der berechneten Instanz.

$|V|$ Anzahl der Knoten der Instanz.

$|E|$ Anzahl der Kanten der Instanz.

$|C|$ Anzahl der Clustern der Instanz.

S_i Die Anzahl der Farben, die von der durch *onestepCD* berechneten Ausgangslösung benötigt werden.

N Die verwendeten Nachbarschaften in Reihenfolge der Iteration:

- `c` *ChangeColor*
- `n` *ChangeNode*
- `a` *ChangeAll*

`d` *DSATUR*

S_{avg} Die im Durchschnitt benötigte Anzahl an Farben der von unserer Implementierung in 40 Testläufen berechneten Lösungen.

S_σ Die Standardabweichung der Ergebnisse aller 40 Testläufe (verursacht durch Randomisierung in den einzelnen Verfahren).

S_Δ Prozentuelle Verbesserung der Ausgangslösung durch den beschriebenen Ansatz.

t Die mittlere Laufzeit in Sekunden.

A.1. Kurzinterpretation

Wie man erkennen kann ist die Reihenfolge, in der das VND die einzelnen Nachbarschaften durchsuchen muss, um die jeweils besten Resultate zu erzielen, nicht wirklich eindeutig. Die erzielten Verbesserung gegenüber der von der Konstruktionsheuristik *onestepCD* ermittelten Ausgangslösungen sind mitunter beachtlich und bewegen sich großteils im Bereich zwischen 11 und 15%.

Instanz	$\|V\|$	$\|E\|$	$\|C\|$	S_i	N	S_{min}	S_{avg}	S_σ	S_Δ	t
dsjc500.5-1.in	500	62624	500	69	cnad	62	63.67	0.52	11.29	494109.3
					cand		63.55	0.55		494685.3
					acnd		63.67	0.52		492537.0
dsjc500.5-2.in	1000	249671	500		cnda	60	61.30	0.51	15.00	482239.0
					cadn		61.27	0.50		472219.3
					dcan		61.55	0.55		488494.5
					danc		61.40	0.54		497711.0
					ncda		61.13	0.40		496069.8
					ndca		61.27	0.50		499308.0
					ndac		61.17	0.49		508483.5
					nacd		61.45	0.55		448795.8
					nadc		60.95	0.31		499959.3
					acdn		60.88	0.40		489919.3
					adcn		60.80	0.40		502662.8
					adnc		60.95	0.31		509733.8
					ancd		61.70	0.51		443322.5
					andc		61.00	0.45		499170.8
dsjc500.5-3.in	1500	562401	500		cdna	60	61.20	0.46	15.00	477578.0
					cadn		61.20	0.51		465780.5
					dcna		61.38	0.53		479848.8
					dnac		61.23	0.52		498028.5
					danc		61.23	0.47		489702.3
					ncda		60.95	0.22		475189.8
					ndca		60.88	0.33		484062.0
					ndac		60.98	0.16		488796.3
					nacd		61.55	0.59		432441.0
					nadc		60.92	0.35		491085.8
					acdn		60.95	0.22		483350.0
					adcn		60.98	0.35		493467.8
					adnc		60.95	0.22		501021.3
					andc		60.98	0.27		491790.3
dsjc500.5-4.in	2000	999508	500		adnc	59	60.65	0.53	16.95	497422.8

Tabelle A.1.: Ergebnisse der VNS