Zusammenfassung

Die vorliegende Arbeit behandelt das Problem der Erweiterung eines gegebenen Graphen auf zweifachen Zusammenhang durch die Auswahl einer möglichst kostenminimalen Menge an zusätzlichen Kanten. Ein praxisbezogenes Beispiel für die Problemstellung ist die Erweiterung eines bestehenden Kommunikationsnetzwerkes durch zusätzliche Verbindungen, so dass bei einem zeitweiligen Ausfall eines beliebigen (Verteiler-)Knotens noch immer die Erreichbarkeit aller anderen Knoten untereinander gewährleistet ist. Der hier vorgestellte Memetische Algorithmus umfasst eine effektive, deterministische Vorverarbeitung der Probleminstanzen zur Reduktion des zu betrachtenden Suchraums, effiziente, speziell auf die Problemstellung zugeschnittene Variations-Operatoren und eine schnelle lokale Optimierungsstrategie, die auf jede generierte Lösung angewandt wird, bevor diese der Population hinzugefügt wird. Hierdurch wird gewährleistet, dass die Population des Algorithmus immer eine Menge von gültigen und lokal optimalen Lösungskandidaten ist. Ein über zwei verschiedene Probleminstanzklassen durchgeführter empirischer Vergleich zwischen dem vorgestellten Algorithmus und zwei bisherigen heuristischen Ansätzen und einem genetischen Algorithmus gibt Auskunft über die Leistungsfähigkeit des Verfahrens und zeigt die Überlegenheit des neuen Ansatzes.

Abstract

This diploma thesis considers the vertex biconnectivity augmentation problem, where a given graph needs to be augmented by a cheapest possible set of additional edges to become vertex biconnected. A real world instance of this problem is the enhancement of an already established communication network by additional connections to ensure the connectivity of all other nodes in case of a temporary breakdown of any single node. The presented memetic algorithm includes an effective deterministic preprocessing of problem data, efficient problem dependent variation operators and a fast local improvement strategy which is applied before a solution is added to the population. In this way the population processed by the algorithm consists always of only feasible and locally optimal solution candidates. Empirical results on two sets of test instances indicate the superiority of the new approach over two previous heuristics and an earlier evolutionary algorithm.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Zuverlässigkeit ist einer der wichtigsten Punkte beim Aufbau eines Kommunikationsnetzes. Unabhängig von der spezifischen Art der Verbindung und der Teilnehmer, seien es jetzt Menschen via Telefon oder Computer in einem großen Firmennetz, ist es in einem kommerziell betriebenen Netzwerk zumeist nicht akzeptabel, dass durch den Ausfall eines einzelnen Knotenpunktes das gesamte Netz in Mitleidenschaft gezogen wird, weil mitunter größere Teilbereiche nicht mehr erreichbar sind. Um dem entgegenzuwirken, werden redundante Verbindungen errichtet, die alternative Routen im Fall eines temporären Ausfalls eines beliebigen Knotens schaffen. Diese zusätzlichen Verbindungen sollen im Allgemeinen möglichst wirtschaftlich gewählt werden, und genau das ist die Problemstellung zu dieser Arbeit. Abbildung 1.1 auf Seite 2 zeigt hierzu ein Beispiel.

1.1 Über die folgenden Kapitel

Kapitel 2: Es wird zunächst die Problemstellung genauer erläutert und auch formal beschrieben. Weiters werden hier die im Folgenden verwendeten Definitionen und Bezeichnungen erklärt.

Kapitel 3: Bisherige Ansätze zur Lösung dieses Problems werden vorgestellt, insbesondere auch jene, mit denen in Kapitel 7 die empirischen Resultate verglichen

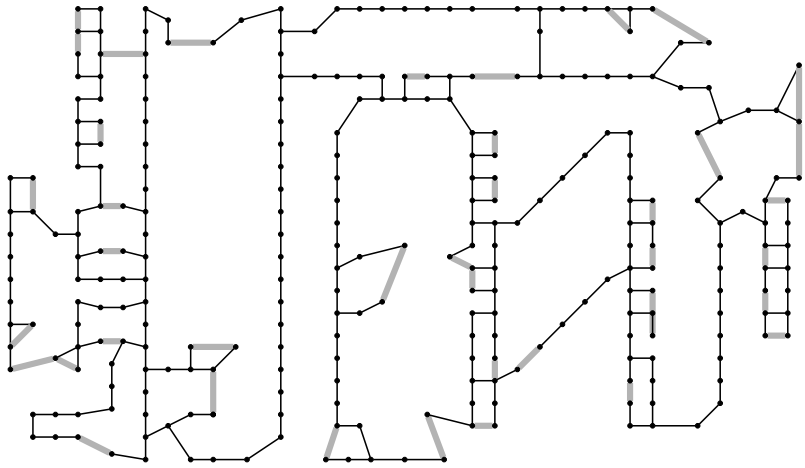


Abbildung 1.1: Das hier dargestellte Netzwerk besteht aus 280 Knoten und wird durch die grau eingezeichneten Kanten auf zweifachen Zusammenhang erweitert.

werden.

Kapitel 4: Dieses Kapitel stellt eine kurze Einführung zu Genetischen und Memetischen Algorithmen dar und dient als Vorbereitung und zum leichteren Verständnis für Kapitel 6, teilweise auch für Kapitel 3.

Kapitel 5: Die Vorverarbeitungsschritte für das nachfolgende Optimierungsverfahren werden hier vorgestellt. Ziel ist es, die Probleminstanz soweit wie möglich zu vereinfachen, um den Suchraum für den Memetischen Algorithmus einzuengen. Für einfache Instanzen besteht die Möglichkeit, dass hierbei bereits eine optimale Lösung gefunden wird.

Kapitel 6: Die Funktionsweise des Memetischen Algorithmus wird in diesem Kapitel erläutert. Die verwendeten Operatoren zusammen mit möglichen alternativen Varianten werden vorgestellt.

Kapitel 7: Nach einem Überblick über die verwendeten Testinstanzen werden die Resultate des Memetischen Algorithmus mit denen anderer aktueller Verfahren ver-

glichen.

Kapitel 8: Hier finden sich Hinweise zu den für diese Arbeit erstellten Programmen und Hilfsprogrammen, die auf der beiliegenden CD zu finden sind.

Kapitel 9: Dieses Kapitel enthält abschließende Bemerkungen zu dem vorgestellten Algorithmus und resümiert die Ergebnisse dieser Arbeit.

Anhang A: Die Probleminstanzen werden in diesem Abschnitt grafisch dargestellt. Zu jeder Abbildung werden die wichtigsten Eckdaten aufgeführt.

Anhang B: Eine Kurzfassung dieser Arbeit wurde in [23] zusammen mit G. Raidl und I. Ljubić in Form eines Konferenzbeitrages veröffentlicht. Der Anhang beinhaltet den Beitrag entsprechend der für die Veröffentlichung vorgeschriebenen Layout-Vorgaben des Springer-Verlages.

Kapitel 2

Die Problemstellung

Für ein kommerziell betriebenes Kommunikationsnetzwerk ist neben der Leistungsfähigkeit vor allem dessen Verfügbarkeit von Bedeutung. Die Verfügbarkeit eines Systems definiert sich durch die Zeit, die das System im Mittel spezifikationsgemäß funktioniert dividiert durch dieselbe Zeitspanne zuzüglich der mittleren Zeit zur Wiederinstandsetzung (u.a. [25]). Um eine hohe Verfügbarkeit zu gewährleisten ist es extrem wichtig, dass das System auch bei Auftreten von Fehlern die gestellten Anforderungen so weit wie möglich weiterhin erfüllt. Dies nennt man *Survivability*. Eine der grundlegendsten Anforderungen an ein Kommunikationsnetzwerk ist die Erreichbarkeit der Teilnehmer. Es ist in vielen Anwendungsbereichen nicht akzeptabel, dass bereits bei Ausfall eines Serviceknotens – zum Beispiel ein Router oder ein weitervermittelnder Rechner – Teilbereiche des Netzwerks nicht mehr erreichbar sind, da das Netz in unzusammenhängende Komponenten zerfällt. Um ein Netzwerk robust gegenüber Störungen dieser Art zu gestalten, müssen redundante Verbindungen errichtet werden, um beim (temporären) Versagen eines beliebigen Knotens alternative Verbindungswege bereitzustellen. Weiterführende Literatur zu *Survivability* von Netzwerken findet man unter anderem in [9, 17, 36].

Zur Modellierung eines Netzwerkes bietet sich die Darstellung in Form eines einfachen, ungerichteten Graphen an. Die Knoten in diesem Graphen repräsentieren entweder weitervermittelnde Einheiten wie zum Beispiel einen Router oder Switch, oder nicht weitervermittelnde Einheiten wie zum Beispiel ein Terminal. Bestehende Verbindungen

werden durch die Kanten des Graphen modelliert, mögliche zusätzliche Verbindungen werden zusammen mit ihren errechneten oder geschätzten Kosten separat verwaltet. Die Entfernung eines weitervermittelnden Knotens teilt diesen Graphen im Allgemeinen in unzusammenhängende Komponenten.

In der Graphentheorie wird die Toleranz gegenüber Knotenausfällen durch den *k-fachen Zusammenhang* eines Graphen beschrieben. Ein Netzwerk wird als k-fach zusammenhängend bezeichnet, wenn mindestens k Knoten zusammen mit den ihnen benachbarten Kanten entfernt werden müssen, damit es in zwei oder mehr unzusammenhängende Komponenten zerfällt. Ein Netzwerk mit mindestens zweifachem Zusammenhang ($k = 2$) gilt als robust. Im Allgemeinen geht man davon aus, dass ein Fehler behoben werden kann, bevor ein weiterer auftritt. In der Praxis ist es üblich, dass die zusätzliche Robustheit eines dreifachen Zusammenhangs ($k = 3$) durch die damit verbundenen Mehrkosten nicht gerechtfertigt wird. Aus diesem Grund befasst sich diese Arbeit mit dem häufigsten Fall, einem zweifach zusammenhängenden Netzwerk ($k = 2$). In diesem Fall müssen mindestens zwei Knoten zusammen mit allen zu diesen Knoten führenden Kanten entfernt werden, damit der Graph in zwei oder mehr Teilgraphen zerfällt. Man nennt einen solchen Graphen *zweifach zusammenhängend*. Für das Netzwerk bedeutet das, dass erst unter diesen Voraussetzungen Teilbereiche nicht mehr erreichbar sind.

In der Problemstellung der Erweiterung eines Graphen auf zweifachen Zusammenhang, im Englischen *Vertex Biconnectivity Augmentation Problem* genannt, kurz V2AUG, ist ein zusammenhängender aber nicht zweifach zusammenhängender Graph gegeben. In einem solchen Graphen existieren demzufolge kritische Knoten, die, sobald auch nur einer von ihnen entfernt wird, einen Zerfall des Graphen in einzelne Teilgraphen zur Folge haben. Diese Knoten nennt man im Englischen *Cut Nodes* oder *Cut Points*, im Deutschen werden sie als Artikulationsknoten, Gelenksknoten oder Brückenknoten bezeichnet. Solche Knoten gilt es zu identifizieren und durch das Hinzufügen von weiteren Kanten abzusichern. Es müssen also zusätzliche Verbindungen ausgewählt werden, die beim Entfernen des entsprechenden Knotens verhindern, dass der Graph in Teilgraphen zerfällt. Diesen Vorgang bezeichnet man als das *Abdecken von Artikulationsknoten* oder im Englischen „Covering of Cut Nodes“. Das Ziel des hier präsentierten Algorithmus ist es, alle in dem vorhandenen Graphen existierenden Artikulationsknoten durch Hinzufügen neuer Kanten mit möglichst geringen Gesamtkosten abzudecken.

2.1 Eine formale Definition

2.1.1 Elementare Begriffe der Graphentheorie

Um das V2AUG-Problem formal zu beschreiben, werden ein paar grundlegende Definitionen aus dem Bereich der Graphentheorie benötigt, die auch in späteren Kapiteln immer wieder Verwendung finden werden. Es folgt eine kurze Wiederholung. Grundlegende Literatur zur Graphentheorie findet sich unter anderem in [3, 46, 38].

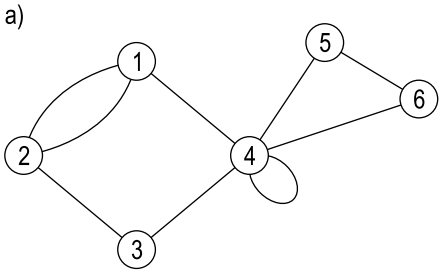
Ein Graph $G = (V, E)$ wird durch eine Knotenmenge V und eine Kantenmenge E beschrieben. Jede Kante $e \in E$ entspricht einem Knotenpaar $e = (v_1, v_2)$ mit $v_1, v_2 \in V$. Ist dieses Knotenpaar ungeordnet, so heißt die Kante e *ungerichtet*, und v_1 und v_2 sind ihre *Endpunkte*. Weiters bezeichnet man die Knoten v_1 und v_2 als *benachbart* oder *adjazent*. Sind alle Kanten des Graphen ungerichtet, so spricht man von einem *ungerichteten Graphen*. In einem ungerichteten Graphen gibt der *Grad* eines Knoten $v \in V$ an, wieviele Kanten ihn als Endpunkt beinhalten.

Gilt für zwei Kanten e_1 und e_2, dass ihnen dasselbe Knotenpaar zugeordnet ist, so heißen sie *Mehrfachkanten*. Eine Kante $e = (v, v)$ bezeichnet man als *Schlinge*. Besitzt ein Graph keine Mehrfachkanten oder Schlingen, so bezeichnet man ihn als *einfachen* oder *schlichten* Graphen. Ist jeder Knoten eines solchen Graphen zu jedem andern benachbart, so heißt der Graph *vollständig*.

Ein Graph $H = (V', E')$ heißt *Teilgraph* von $G = (V, E)$, wenn $V' \subseteq V$ und $E' \subseteq E$ gilt. Gilt $V' = V$, so bezeichnet man H als *spannenden Teilgraph*. Ist jeder Knoten $v \in V'$ über eine oder mehrere Kanten von jedem anderen Knoten aus erreichbar, so heißt H *zusammenhängend*, selbiges gilt natürlich auch für G. Ist dies nicht der Fall, so bilden die jeweils untereinander erreichbaren Knoten die *zusammenhängenden Komponenten* des (Teil-)Graphen.

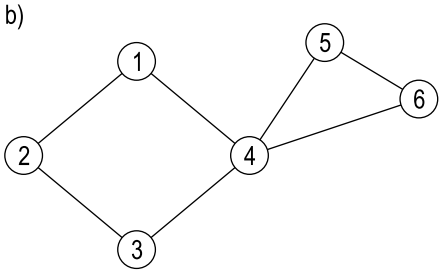
Die Beispiele der Abbildung 2.1 auf Seite 7 veranschaulichen die vorangegangenen Definitionen und Begriffe.

a)



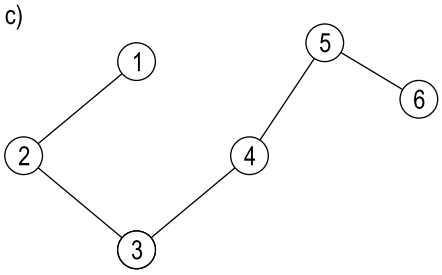
Darstellung eines ungerichteten, zusammenhängenden Graphen $G = (V, E)$, mit Mehrfachkanten zwischen Knoten 1 und 2, und einer Schlinge bei Knoten 4.

b)



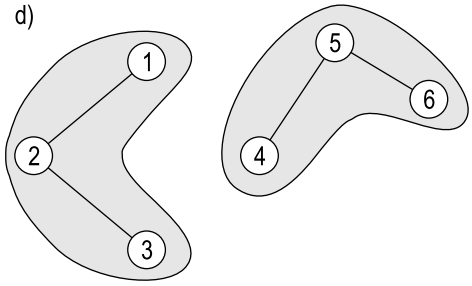
Entfernt man aus G eine der Mehrfachkanten und die Schlinge, so erhält man einen einfachen Graphen.

c)



Ein spannender Teilgraph $H = (V, E')$ des Graphen G.

d)



Durch das Entfernen der Kante zwischen Knoten 3 und 4 zerfällt der Teilgraph H in zwei zusammenhängende Komponenten. Diese sind grau markiert und umfassen die Knoten 1 bis 3, beziehungsweise 4 bis 6.

Abbildung 2.1: Beispiele für die wichtigsten Begriffe und Definitionen.

2.1.2 Die Problemstellung formal zusammengefasst

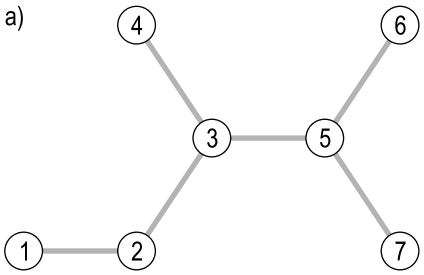
Das V2AUG-Problem lässt sich wie folgt formal charakterisieren: Sei $G = (V, E)$ ein ungerichteter, zweifach zusammenhängender Graph, mit Knotenmenge V, und E die Menge aller möglichen Kanten. Über die Funktion $Kosten(e) > 0$ werden allen Kanten $e \in E$ entsprechend Kosten zugeordnet. Ein zusammenhängender, spannender aber nicht zweifach zusammenhängender Teilgraph $G_0 = (V, E_0)$ mit $E_0 \subset E$ repräsentiert ein fixes, gegebenes Netzwerk, wobei $E_a = E \setminus E_0$ die Menge jener Kanten repräsentiert, die zur Erweiterung von G_0 verwendet werden können. Ziel ist es nun, eine Teilmenge $E_s \subseteq E_a$ zu bestimmen, sodass der erweiterte Graph $G_s = (V, E_0 \cup E_s)$ zweifach zusammenhängend ist, und

$$Kosten(E_s) = \sum_{e \in E_s} Kosten(e) \tag{2.1}$$

minimal ist.

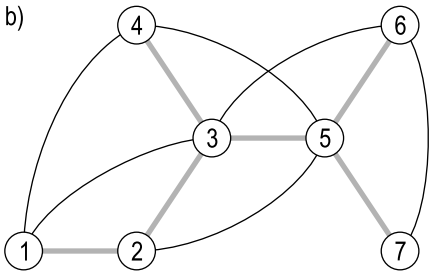
Die Darstellung in Abbildung 2.2 auf Seite 9 zeigt ein einfaches Beispiel der Erweiterung eines Graphen auf zweifachen Zusammenhang und veranschaulicht die Definition der Problemstellung. a) Der zu $G = (V, E)$ gehörige, spannende Teilgraph $G_0 = (V, E_0)$ soll auf zweifachen Zusammenhang erweitert werden. b) Hier wird zusätzlich die Menge der zur Erweiterung verfügbaren Kanten $E_a = E \setminus E_0$ durch dünne, gebogene Linien dargestellt. Durch eine geeignete Auswahl sollen die Artikulationsknoten 2, 3 und 5 abgedeckt werden. c) Durch das Hinzufügen der Kanten $(1, 4)$, $(4, 5)$ und $(6, 7)$ bleibt der Graph zwar zusammenhängend so lange man eine beliebige Kante entfernt, jedoch ist er noch nicht zweifach zusammenhängend. Die Problemstellung des zweifachen *Kanten*-Zusammenhanges wird in [42, 28, 29] genauer erläutert und diente als Grundlage für die hier vorgestellte Arbeit. d) Eine mögliche Lösung des Problems wird durch das Hinzufügen der Kanten (1,4), (2,5), (3,6) und (6,7) erreicht. Der dargestellte Graph $G_s = (V, E_s)$ ist zweifach zusammenhängend. Mindestens zwei seiner Knoten müssen entfernt werden, sodass G_s in untereinander nicht mehr verbundene Komponenten zerfällt.

a)



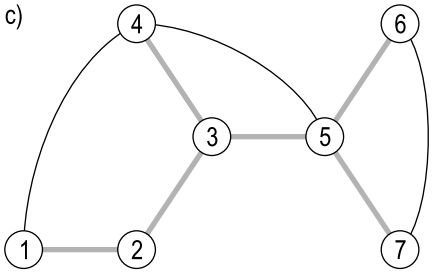
$G_0 = (V, E_0)$ soll auf zweifachen Zusammenhang erweitert werden.

b)



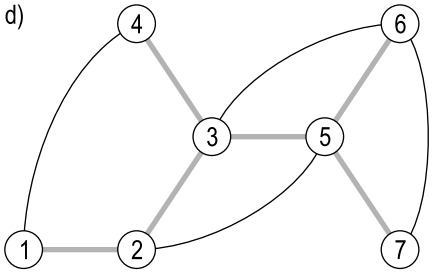
$E_a = E \setminus E_0$ ist die Menge der zur Erweiterung verfügbaren Kanten.

c)



Der Graph bleibt zusammenhängend, wenn eine beliebige *Kante* entfernt wird. Für das Entfernen eines beliebigen *Knotens* gilt dies jedoch nicht.

d)



Eine korrekte Lösung der Problemstellung.

Abbildung 2.2: Ein einfaches Beispiel für die Erweiterung eines gegebenen Graphen G auf zweifachen Zusammenhang.

Kapitel 3

Bisherige Lösungsansätze

Bereits 1976 untersuchten Eswaran und Tarjan [10] das V2AUG-Problem. Für den speziellen Fall, dass G ein vollständiger Graph ist und alle Kanten die gleichen Kosten haben, konnte ein Algorithmus gefunden werden, der das Problem in polynomialer Zeit löst [21]. Im allgemeinen Fall jedoch ist diese Problemstellung NP-schwierig. Selbst unter der Einschränkung, dass die Kosten der Kanten nur aus einer Menge mit 2 Elementen gewählt werden können, bleibt die NP-Schwierigkeit erhalten. Dies wurde von Eswaran und Tarjan [10] gezeigt.

Auf Grund der Komplexität des Problems sind exakte Lösungsmethoden für größere Problemstellungen im allgemeinen Fall nicht praktikabel, weshalb sich die Mehrheit der bisherigen Ansätze auf Näherungsverfahren beschränkt.

3.1 Deterministische Ansätze

Frederickson und Jájá [13] entwickelten ein Näherungsverfahren, das eine Lösung mit maximal doppelten optimalen Kosten garantiert. Der von ihnen entwickelte Algorithmus benutzt einen Vorverarbeitungsschritt, im Zuge dessen der zu erweiternde Graph G_0 in einen so genannten Block-Cut-Tree (BCT) umgewandelt wird. Hierbei werden kritische Knoten identifiziert und bereits existierende zweifach zusammenhängende

Komponenten zu Blockknoten zusammengefasst. Eine genauere Erläuterung der BCT-Darstellung folgt in Kapitel 5.1. In Frederickson und Jájás Ansatz wird der BCT zu einem willkürlich gewählten Endknoten hin gerichtet, und die Erweiterungskanten werden als gerichtete Kanten auf den BCT übertragen. Nach Möglichkeit werden hierbei für eine Lösung irrelevante Kanten identifiziert und ausgefiltert. Durch das Hinzufügen von zusätzlichen Leerknoten und das Verschieben der Endpunkte problematischer Erweiterungskanten erreichen die Autoren, dass der ursprüngliche Graph auch einen zweifachen Zusammenhang aufweist, sobald der BCT stark zusammenhängend ist. Ein Minimum-Outbranching-Algorithmus, wie er von Gabow et al. [15] beschrieben wird, sucht nun in dem ergänzten BCT nach einer möglichen Lösung. Der Rechenaufwand des Verfahrens ist $O(|V^2|)$.

Eine Verbesserung dieses Algorithmus wurde von Khuller und Thurimella [24] entwickelt und reduziert den Zeitaufwand auf $O(|E| + |V| \log |V|)$, wobei aber auch hier nur eine Lösung mit Aproximationsfaktor zwei zum Optimum garantiert werden kann.

Zhu et al. [48] entwickelten basierend auf Khuller und Thurimellas Ansatz eine iterative Variante. Mittels einer von ihnen als Drop-Heuristik bezeichneten Methode wird abgeschätzt, wie geeignet eine gewählte Kante für die Lösung der Probleminstanz ist. Um den Drop-Wert zu ermitteln, wird der verwendete Branching-Algorithmus für jede zu untersuchende Kante zweimal ausgeführt. Das erste Mal behält die Kante ihre Gewichtung, das zweite Mal werden ihre Kosten auf null gesetzt. Die Differenz zwischen beiden Resultaten im Verhältnis zum Gewicht der Kante ergibt den Drop-Wert. Die Kante mit dem höchsten Drop-Wert unterstützt eine mögliche Lösung voraussichtlich am besten und wird dieser hinzugefügt, wobei ihre Gewichtung auf null gesetzt wird. Diese Vorgangsweise wird so lange wiederholt, bis eine komplette Lösung mit Gewicht null erreicht ist. Da das Verfahren durch den gewählten Startknoten beeinflusst wird, wird der Algorithmus für jeden Blattknoten als Startknoten ausgeführt, und die beste gefundene Lösung wird als Resultat gewertet.

Obwohl der Algorithmus auch nur eine Lösung mit Aproximationsfaktor zwei zum Optimum garantieren kann, werden damit in der Praxis weit bessere Resultate erzielt als mit dem Ansatz von Khuller und Thurimella. Die bessere Qualität der Lösung wird jedoch mit einem signifikant höheren Zeitaufwand erkauft. Dies wird auch durch die Testergebnisse in Kapitel 7 bestätigt.

3.2 Stochastische Verfahren

Ein hybrider Genetischer Algorithmus wurde von Ljubić und Kratica [26] für das V2AUG-Problem entwickelt. Die Kodierung von Kandidatenlösungen erfolgt durch Bitstrings, wobei gewählte Erweiterungskanten jeweils durch ein gesetztes Bit im Chromosom repräsentiert werden. Individuen werden durch Tournament-Selection gewählt. Zur Rekombination wird eine Variante von Uniform-Crossover verwendet, bei der rund 30% der Gene ausgetauscht werden. Als Mutationsoperator wird eine Bit-Flip-Mutation verwendet. Das Ersetzungsschema tauscht pro Generation ein Drittel der Individuen aus. Duplikate innerhalb der Population werden bei jedem Generationswechsel entfernt.

Generierte Lösungen werden mittels Tarjans Algorithmus [47] zum Prüfen auf zweifachen Zusammenhang verifiziert. Bei ungültigen Lösungen kommt eine greedy Reparatur-Heuristik zum Einsatz. Diese eliminiert separat einen Artikulationsknoten nach dem anderen. Der abzudeckende Knoten wird vorerst aus dem Graphen entfernt. Die dadurch entstehenden Teilgraphen werden durch Anwendung von Kruskals Algorithmus für minimale Spannbäume wieder verbunden, und die dafür verwendeten Erweiterungskanten werden der Lösung hinzugefügt. Nach dem Wiedereinfügen des Artikulationsknotens ist dieser nun abgedeckt. Dieser Vorgang wird wiederholt, bis die Lösung keine Artikulationsknoten mehr enthält. Der Zeitaufwand für diese Heuristik ist mit $O(|V| \cdot |E| \cdot \log |V|)$ angegeben.

3.3 Eine verwandte Problemstellung

Eine dem V2AUG ähnliche Problemstellung stellt das *Edge-Biconnectivity-Augmentation-Problem*, kurz E2AUG, dar. Hierbei soll ein gegebener Graph derart erweitert werden, dass nach dem Entfernen einer beliebigen *Kante* der Graph noch immer zusammenhängend ist. Diese Problemstellung ist aus algorithmischer Sicht einfacher zu lösen, da jeweils nur der Ausfall einer Kante berücksichtigt werden muss und jede Kante durch eine einzelne Erweiterungskante gesichert werden kann. Ein Graph, der zweifach zusammenhängend ist, ist auch zweifach Kanten-zusammenhängend. Der umgekehrte Schluss gilt allerdings nicht.

Raidl und Ljubić [42, 28] beschreiben einen effizienten Evolutionären Algorithmus für dieses Problem. Basierend auf einer kompakten Kantenmengen-Kodierung wird zunächst der Suchraum durch ein deterministisches Preprocessing eingeengt. Hiernach folgt der eigentliche Algorithmus, der eine lokale Optimierung bei den Hauptoperatoren – Rekombination, Mutation und auch Initialisierung – miteinbezieht und somit nur die lokalen Optima durchsucht. Die Resultate und Erkenntnisse dieses Verfahrens lieferten die Grundlage für den hier vorgestellten Algorithmus.

Kapitel 4

Genetische und Memetische Algorithmen

4.1 Überblick

Genetische Algorithmen, im Folgenden auch kurz GAs, sind robuste, stochastische Suchalgorithmen, deren Funktionsweise lose jene Mechanismen modelliert, die in der Natur Anpassung und Evolution bewirken. Die Vorgangsweise ist dabei nicht deterministisch, aber zielgerichtet. Auf eine Menge von möglichen Lösungen, den so genannten Individuen, die in ihrer Gesamtheit die Population eines Genetischen Algorithmus darstellen, werden iterativ die Modellvarianten der natürlichen Selektion, der Rekombination und Mutation angewandt. Die Selektion simuliert, dass sich starke gegenüber schwachen Individuen durchsetzen, die Rekombination entspricht der Vermehrung, also Eigenschaften der Eltern vererben sich auf die Kinder der nächsten Generation, und die Mutation erzeugt zufällige Veränderungen am Genmaterial. Dies geschieht so lange, bis ein zuvor definiertes Abbruchkriterium, wie zum Beispiel eine gewisse Anzahl an Generationen oder die Unterschreitung einer relativen Verbesserung, erreicht ist. Gesteuert wird die Suche durch eine Bewertungsfunktion, bei Genetischen Algorithmen Fitnessfunktion genannt, die eine qualitative Bewertung der Individuen durchführt und somit Einfluss auf die Wahrscheinlichkeit der Fortpflanzung oder Überlebensfähigkeit

eines Individuums hat. Ausführliche Abhandlungen zu Genetischen Algorithmen finden sich unter anderem in [20, 16] und [33]. Weiterführende Literaturverweise finden sich in Abschnitt 4.3.

Memetische Algorithmen, im Folgenden kurz MAs, basieren auf Genetischen Algorithmen. Auch bei ihnen wird eine auf einer Population von Lösungen aufbauende Suche durchgeführt, Individuen werden selektiert, rekombiniert und mutiert, jedoch kommt ein neuer Aspekt hinzu: Man versucht, dem Algorithmus so viel Wissen wie möglich in Form von Heuristiken, integrierten Approximationsverfahren oder spezialisierten Operatoren mitzugeben. Weiters ist man bestrebt, nur einen Teilraum des eigentlichen Suchraums im MA zu betrachten, nämlich den lokaler Optima. Man kann Memetische Algorithmen durchaus auch als problemspezialisierte, hybride Genetische Algorithmen betrachten, die sich im Vergleich zu einem konventionellen GA weit weniger an Vorbilder der Natur anlehnen, sondern die darin enthaltenen Ideen generalisieren und an die zu lösende Problematik anpassen. Eine ausführliche Beschreibung zu Memetischen Algorithmen findet man in [35]. Weiterführende Literaturverweise finden sich in Abschnitt 4.3.

4.2 Der schematische Ablauf

Im Folgenden wird nun der Aufbau eines konventionellen Genetischen Algorithmus betrachtet, wobei Unterschiede zu Memetischen Algorithmen aufgezeigt werden. Es folgt zunächst die Pseudocode-Darstellung beider Varianten zur besseren Orientierung in Abbildung 4.1 auf Seite 17 und Abbildung 4.2 auf Seite 18. Italic-Schrift kennzeichnet hierbei *Variablen* oder wichtige, veränderliche *Datenstrukturen*, `Methoden` werden in Schreibmaschinenschrift wiedergegeben und BEFEHLE zur Programmsteuerung in Kapitälchen.

Der Ablauf eines Genetischen Algorithmus beginnt mit der Initialisierung (siehe 4.2.2) einer Startpopulation *pop*, die aus zumeist zufällig generierten Lösungen, genannt Individuen, besteht. Jedes Individuum wird entsprechend seiner Qualität bewertet. Das Ergebnis entscheidet im weiteren Verlauf über einen Verbleib in der Population und beeinflusst eine mögliche Selektion. Zunächst werden zwei so genannte Elternindivi-

duen S_1, S_2 aus der Population *pop* selektiert (siehe 4.2.3) und generieren durch die Rekombination (siehe 4.2.4) einen neuen Nachkommen S_c. Das neu generierte Individuum S_c wird mit einer zuvor festgelegten Wahrscheinlichkeit durch die Mutation (siehe 4.2.5) verändert und anschließend hinsichtlich seiner Qualität bewertet. Neu generierte Nachkommen werden durch ein zuvor festgelegtes Ersetzungsschema in die Population eingegliedert und ersetzen (teilweise) andere Individuen (siehe 4.2.6), wodurch die neue, modifizierte Population pop' gebildet wird. Die benötigten Schritte, um von einer Population *pop* zu einer neuen pop' zu gelangen, bezeichnet man als Erzeugen einer neuen Generation. Dieser Vorgang wiederholt sich, bis ein zuvor definiertes Abbruchkriterium (siehe 4.2.7) erfüllt ist.

Auch ein Memetischer Algorithmus beginnt mit der Initialisierung einer Startpopulation *pop*. Hier aber wird zur Verbesserung der Lösungen eine lokale Optimierung durchgeführt. Die Bewertung der Individuen erfolgt analog zum GA. Um einen Nachkommen zu erzeugen wird eine Menge S_{Eltern} an Individuen über die Selektion aus der Population *pop* für die Rekombination gewählt, wodurch mehr als zwei Elternteile an der Rekombination beteiligt sein können. Durch die Rekombination wird ein neues Individuum S_c erzeugt, das optimiert und mit der festgelegten Mutationswahrscheinlichkeit mutiert wird. Ein mutiertes Individuum wird erneut optimiert. Das generierte Individuum wird anschließend bewertet. Die neu generierten Nachkommen werden wie bei einem GA durch ein zuvor festgelegtes Ersetzungsschema in die Population eingegliedert. Dieser Vorgang wird wiederholt, bis ein zuvor definiertes Abbruchkriterium erfüllt ist.

4.2.1 Die Wahl einer geeigneten Kodierung

Ein wesentlicher Punkt bei der Entwicklung eines GA/MA ist die Wahl einer geeigneten Kodierung der Individuen, die die Lösungen darstellen. Schon in diesem Schritt kann die Effizienz deutlich positiv oder negativ beeinflusst werden. Eine geeignete Kodierung soll im Allgemeinen alle möglichen, zumindest jedoch die optimalen Lösungen darstellen können. Sie soll hinreichend kompakt sein, um bei steigender Problemgröße noch praktikabel zu bleiben, und sie soll gewährleisten, dass eine kleine Änderung am Chromosom nur eine kleine Änderung der Lösung bewirkt. Während Memetische

```
Genetischer_Algorithmus()
BEGINN
    pop ← Initialisiere die Startpopulation
    FÜR ALLE Individuen S ∈ pop
        Bewerte (S)
    WIEDERHOLE
        FÜR die gewünschte Anzahl an Rekombinationen
        BEGINN
            S_1, S_2 ← Selektion(pop) zweier Individuen für die Rekombination
            S_c ← Rekombination (S_1, S_2)
            S_c ← Mutation (S_c)
            Bewerte (S_c)
        ENDE
        pop' ← pop;  ein Ersetzungsschema bestimmt, wie die zuvor generierten
                     Nachkommen S_c in die Population übernommen werden
    BIS das Abbruchkriterium erfüllt ist
ENDE
```

Abbildung 4.1: Pseudocode für einen konventionellen Genetischen Algorithmus.

```
Memetischer_Algorithmus()
BEGINN
    pop ← Initialisiere die Startpopulation
    FÜR ALLE Individuen S ∈ pop
        S ← Lokale_Optimierung (S)
        Bewerte (S)
    WIEDERHOLE
        FÜR die gewünschte Anzahl an Rekombinationen
        BEGINN
            S_Eltern ← Selektion einer Menge S_Eltern ⊆ pop für die Rekombination
            S_c ← Rekombination (S_Eltern)
            S_c ← Lokale_Optimierung (S_c)
            S_c ← Mutation (S_c)
            S_c ← Lokale_Optimierung (S_c)
            Bewerte (S_c)
        ENDE
        pop' ← pop;  ein Ersetzungsschema bestimmt, wie die zuvor generierten
                     Nachkommen S_c in die Population übernommen werden
    BIS das Abbruchkriterium erfüllt ist
ENDE
```

Abbildung 4.2: Pseudocode für einen auf einer lokalen Suche basierenden Memetischen Algorithmus, Kombinationen mit anderen Optimierungsverfahren sind möglich. Man beachte, dass veränderte Individuen stets lokal optimiert werden, bevor sie in einem weiteren Bearbeitungsschritt verwendet werden. Weiters kann die Mutation direkt nach der Rekombination folgen, wie es hier der Fall ist, muss es aber nicht. Eine parallel laufende Schleife kann zum Beispiel Individuen direkt aus der Population selektieren.

Algorithmen ein breites Spektrum an unterschiedlichen Kodierungsverfahren nutzen, wird bei klassischen Genetischen Algorithmen oft eine Bitkodierung verwendet. Hierbei werden einzelne Parameter einer potentiellen Lösung in Form von Bitfolgen fixer Länge kodiert, den so genannten Genen eines Individuums. In einer fix gewählten Anordnung aneinander gereiht ergeben die Gene in ihrer Gesamtheit nun einen Bitstring, das Chromosom eines Individuums, und repräsentieren so einen Punkt im Suchraum. Ein Vorteil dieser Art der Kodierung ist, dass nachfolgend beschriebene Operatoren unabhängig von dem tatsächlich bearbeiteten Problem auf derart kodierte Lösungen angewandt werden können und somit universell einsetzbar sind. Zu berücksichtigen ist jedoch, dass durchaus Chromosomen durch die Anwendung von Mutation oder Rekombination entstehen können, die keiner definierten oder gültigen Lösung entsprechen. In diesem Fall muss nun entweder eine Reparaturfunktion eingesetzt werden, oder die mangelnde Entsprechung ist bei der Bewertung der Individuen zu berücksichtigen.

Hierzu ein kurzes Beispiel in Abbildung 4.3, in dem der Zustand zweier unterschiedlicher Produktionsmaschinen kodiert werden soll.

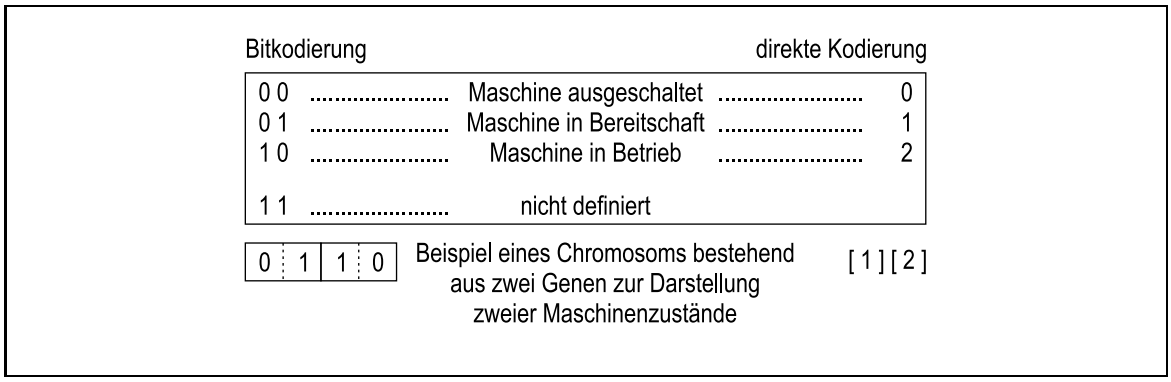


Abbildung 4.3: Gegenüberstellung von zwei möglichen Kodiervarianten.

4.2.2 Initialisierung

Um dem Algorithmus eine möglichst breit gefächerte Ausgangsbasis zu geben und eine vorzeitige Konvergenz zu vermeiden, werden die Individuen der Startpopulation zumeist zufällig initialisiert.

Bei Memetischen Algorithmen ist man bestrebt, auch in diesem Schritt schon vorhandenes Wissen zu integrieren. Dies kann zum Beispiel dadurch geschehen, dass man Bereiche des Suchraums begünstigt, in denen die Wahrscheinlichkeit einer optimalen Lösung höher ist. Weiters wird jedes erzeugte Individuum noch lokal optimiert, bevor es der Population hinzugefügt wird.

4.2.3 Selektion und Bewertung von Individuen

Jedem Individuum wird durch eine Bewertungsfunktion, bei Genetischen Algorithmen oft auch Fitnessfunktion genannt, eine qualitative Maßzahl zugeordnet. Diese Fitnessfunktion ist verständlicherweise problemspezifisch, aber als generelle Richtlinie gilt: Je höher die Fitness eines Individuums ist, desto besser ist die Lösung. Die Bewertung eines Individuums beeinflusst die Wahrscheinlichkeit, mit der ein Individuum zur Rekombination ausgewählt wird. Oft verwendete Selektionsvarianten sind zum Beispiel die *Fitness-Proportional-Selection*, oder auch *Roulette-Wheel-Selection* genannt, und die *k-Tournament-Selection*. Bei ersterer gibt das Verhältnis des Fitnesswertes eines Individuums zur Summe aller Fitnesswerte der Population die Wahrscheinlichkeit an, mit der es gewählt wird. Liegen die Fitnesswerte nahe beieinander, kann eine Skalierung sinnvoll sein, um die qualitativen Unterscheidungsmerkmale zu verdeutlichen. Bei der *k-Tournament-Selection* wird jeweils der Sieger aus einem Turnier von k zufällig ausgewählten Teilnehmern selektiert, wobei der Fitnesswert über den Ausgang des Turniers bestimmt. Um auch die Wahl des schlechtest bewerteten Individuums zu ermöglichen, sollten die Teilnehmer der Turniere mit Zurücklegen gewählt werden. In [5] werden verschiedene Varianten der Selektion für Genetische Algorithmen und Evolutionsstrategien – hierfür siehe [43] u. [45], oder z.B. [4] für eine kurze Übersicht – hinsichtlich des von ihnen erzeugten Selektionsdrucks analysiert.

Bei Memetischen Algorithmen bestimmt die Selektion ebenfalls die Menge der Individuen, die an der Rekombination beteiligt sein sollen, und folgt somit denselben Zielsetzungen wie die eines GA. Soll sich die Mutation nicht auf neu generierte Individuen beschränken, kann eine zweite Selektion integriert werden, um zu mutierende Individuen direkt aus der Gesamtpopulation zu wählen (siehe [35]). Dies kann durch eine zufällige Auswahl oder komplexere Strategien verwirklicht werden.

4.2.4 Rekombination

Die Rekombination, bei Genetischen Algorithmen auch *Crossover* genannt, wird in einem GA oft als der wichtigste Variations-Operator angesehen [16] und wird mit einer zuvor gewählten Wahrscheinlichkeit angewandt. Durch sie wird bestimmt, wie sich gute Teillösungen auf die nachfolgenden Generationen vererben können und somit zur Konvergenz der Population in Richtung Optimum beitragen. Abhängig von dem zu lösenden Problem kann mal die eine oder andere Rekombinations-Strategie besser sein. Mittels Selektion werden zwei Elternindividuen bestimmt, deren Chromosomen nun die Basis für ihren Nachkommen bilden. Zwei bekannte Strategien für binär kodierte Chromosomen sind das *Two-Point-Crossover* und das *Uniform-Crossover*. Bei beiden Verfahren ergeben sich zwei mögliche Nachkommen, von denen im Allgemeinen aber nur einer erzeugt wird.

Beim Two-Point-Crossover werden zwei zufällig gewählte Schnittpunkte, die für beide Chromosomen gleich sind, ermittelt. Die durch diese Schnittpunkte definierten Teilbereiche der Chromosomen übertragen sich nun alternierend auf zwei mögliche Nachkommen. Das Beispiel in Abbildung 4.4 verdeutlicht die Vorgangsweise.

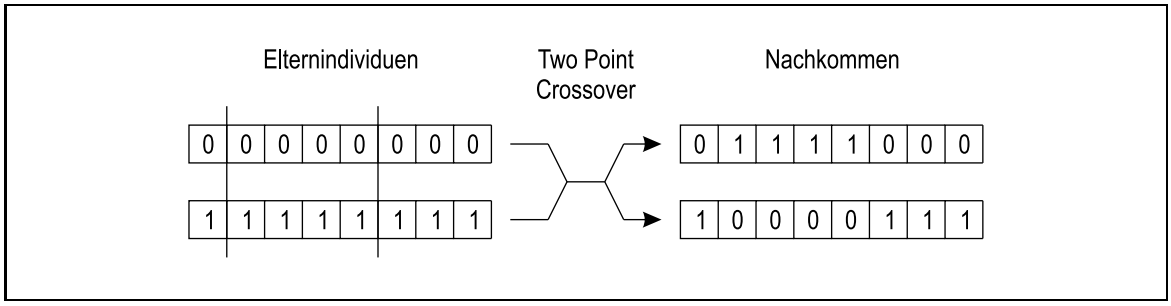


Abbildung 4.4: Two–Point–Crossover. Die Chromosomen der beiden Elternteile kombinieren sich abhängig von zwei zufällig gewählten Schnittpunkten zu zwei möglichen Nachkommen.

Beim Uniform-Crossover wird bitweise entschieden, von welchem Elternteil die Information übernommen wird. Wiederum ergeben sich zwei mögliche Nachkommen. Die Abbildung 4.5 zeigt hierzu ein Beispiel.

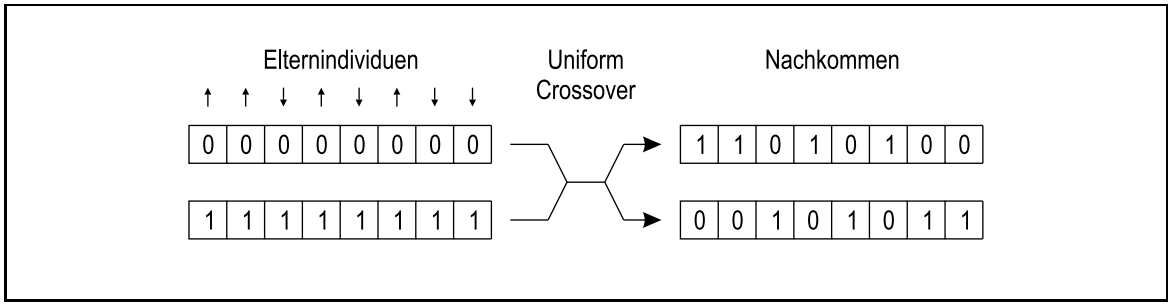


Abbildung 4.5: Uniform–Crossover. Die Elternteile vererben sich bitweise zufällig auf ihre möglichen Nachkommen.

Bei Memetischen Algorithmen spricht man im Zusammenhang mit der Rekombination von einer *k-Merger-Methode*. Wie der Name schon andeutet, limitiert man den Vermehrungsprozess nicht von vornherein auf zwei Elternteile, sondern wählt eine für das Problem günstige Anzahl an beteiligten Individuen, die zu gleichen oder unterschiedlichen Anteilen an der Erschaffung eines Nachkommen beteiligt sind. Die hierbei verwendeten Operatoren sind meist auf das Problem zugeschnitten, und erzeugte Nachkommen werden lokal optimiert, bevor sie der Population hinzugefügt werden.

Das folgende Beispiel in Abbildung 4.6 verwendet einen 3-Merger-Algorithmus, der die durch Mengen repräsentierten Lösungen zuerst vereinigt und anschließend reduziert, um einen Nachkommen zu erzeugen. Elemente, die in mehr als einem Elternteil vorkommen, werden bevorzugt behandelt und direkt auf den Nachkommen vererbt.

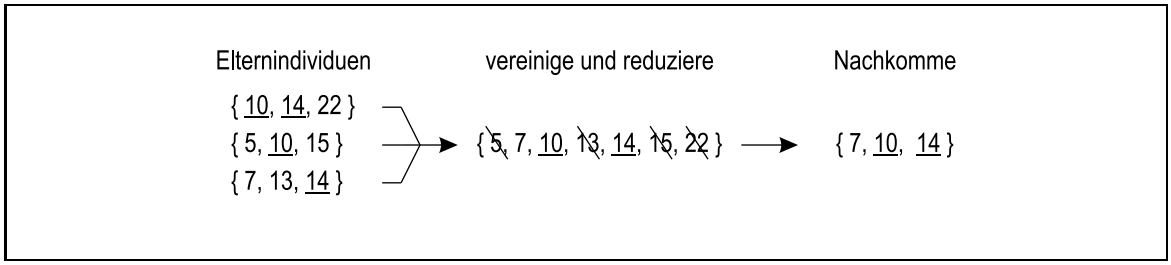


Abbildung 4.6: Beispiel einer 3–Merger–Methode zur Erzeugung eines Nachkommen.

4.2.5 Mutation

Der Mutationsoperator wird in einem Genetischen Algorithmus mit einer zuvor gewählten Wahrscheinlichkeit auf ein durch die Rekombination hervorgebrachtes, neu generiertes Individuum angewandt und bewirkt kleine, zufällige Änderungen im Chromosom. Die Wahrscheinlichkeit hierfür wird meist pro Gen angegeben. Es ist also möglich, dass ein Individuum in mehreren Genen mutiert wird. Durch diesen Vorgang wird neues oder verloren gegangenes Genmaterial wieder in die Population integriert und die Diversität erhöht. Die Mutation hilft somit dem Algorithmus, lokale Optima zu verlassen, und verhindert ein vorzeitiges Konvergieren.

Eine oft verwendete Mutationsvariante ist die Bit-Flip-Mutation. Hierbei wird ein zufällig gewähltes Bit innerhalb des Gens gewählt und invertiert, wie in Abbildung 4.7 zu sehen ist.

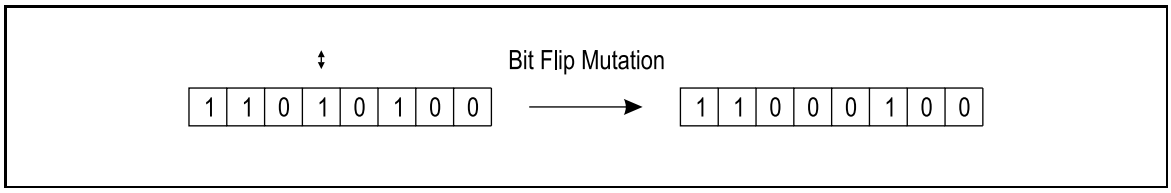


Abbildung 4.7: Bit–Flip–Mutation. Ein einzelnes Bit des Chromosoms wird zufällig bestimmt und invertiert.

Bei Memetischen Algorithmen ist die Mutation nicht notwendigerweise an die Rekombination gebunden. Abhängig von der Problemstellung kann es durchaus sinnvoll sein, bereits in der Population integrierte Individuen erneut für eine Veränderung zu selektieren, was zu einer aktiven Suche im Umfeld der vorhandenen Lösungen führt. Weiters bleibt es dem Designer überlassen, ob das mutierte Individuum das Original ersetzt, wie es bei Genetischen Algorithmen üblich ist, oder als Kopie der Population hinzugefügt wird und das Ersetzungsschema beim Generationswechsel über seinen Verbleib entscheidet.

Das Beispiel in Abbildung 4.8 zeigt eine punktuelle Mutation angewendet auf ein als Menge repräsentiertes Individuum.

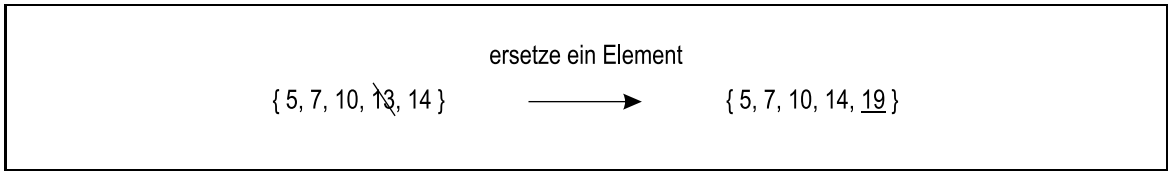


Abbildung 4.8: Durch das Ersetzen eines einzelnen Elementes des Individuums wird eine punktuelle Mutation erreicht.

4.2.6 Ersetzungsschema

Nach der Erzeugung neuer Individuen durch die Rekombination, oder bei Memetischen Algorithmen gegebenenfalls auch durch die Mutation, muss entschieden werden, wie diese in die Population eingefügt werden. Um ein stetiges Wachstum der Population zu verhindern, ist es erforderlich, bereits vorhandene Individuen zu ersetzen. Im Folgenden werden nun einige sehr bekannte Strategien besprochen.

Generational-Replacement: Die einfachste Strategie ist ein generelles Ersetzen der Vorgängerpopulation. Hierbei wird die Elternpopulation vollständig gelöscht und durch eine gleich große Menge an Nachkommen ersetzt. Ein Nachteil dieser Strategie ist, dass auch besonders gute Individuen verloren gehen und eine nachkommende Generation auch schlechter sein kann als die vorangegangene. Die Konvergenzgeschwindigkeit ist somit geringer als bei anderen Strategien. Abhängig von der Art der Problemstellung kann dieser Effekt auch gewollt sein, um aus lokalen Optima eher entkommen zu können.

Elitismus: Um eine Verschlechterung beim Generationswechsel zu verhindern und eine monoton steigende Bewertung des besten Individuums über die Generationen zu erhalten, werden die besten n Individuen (die Elite) in die nachfolgende Generation übernommen. Die Zahl n ist dabei im Allgemeinen klein, z.B. $n = 1$.

Overlapping-Populations: Bei dieser Strategie wird nur ein Teil der Vorgängerpopulation durch Nachkommen ersetzt. Die Vorteile hierbei sind zum einen technischer Natur – es wird weniger Speicherplatz benötigt, um die Nachfolgegeneration zu erzeugen, was insbesondere bei sehr großen Populationen sinnvoll ist – und zum anderen wird neues Genmaterial früher in die Population eingebracht, wodurch die Evolution beschleunigt

werden kann. Die Auswahl der zu löschenden Individuen kann nun in verschiedenen Variationen erfolgen. Die einfachste Variante ist eine gleichverteilte, zufällige Auswahl der Individuen, die gelöscht werden. Interessanter ist jedoch das Einbeziehen der Fitness in die Wahl der zu ersetzenden Individuen. So kann man guten Individuen eine höhere Chance zu überleben einräumen oder generell nur die schlechtesten ersetzen, um die Konvergenzzeit zu verkürzen.

Manche Algorithmen ziehen zusätzlich noch das *Alter* eines Individuums, also die Anzahl an Generationsschritten, die es bereits in der Population verbracht hat, in Betracht. Sie entfernen bevorzugt jene Individuen, die eine gewisse Altersstufe erreicht haben, um die Chance auf zu dominante Individuen zu reduzieren.

4.2.7 Abbruchkriterien

Die durch einen Genetischen oder Memetischen Algorithmus eingeleitete Suche nach einer optimalen Lösung muss durch ein klar definiertes Abbruchkriterium beendet werden, da beide Algorithmentypen in ihrer Suche nach einer Verbesserung der aktuell besten Lösung von sich aus nicht endende Prozesse sind.

Ein Abbruchkriterium kann sich auf die Zeit oder auf erzielte Ergebnisse beziehen. So kann sich das Abbruchkriterium zum Beispiel an der effektiv benötigten Laufzeit orientieren, an einer festgelegten Anzahl von Generationsschritten, daran, dass keine Verbesserung der aktuell besten Lösung innerhalb von n Generationen erreicht wurde, oder an einer Unterschreitung einer festgelegten Diversität innerhalb der Population.

4.3 Abschließende Bemerkungen und weiterführende Literatur

Dieser Abschnitt vermag nur einen kleinen Überblick über die vielen verschiedenen Spielarten beider Algorithmentypen zu geben. Weiters sei auch angemerkt, dass eine klare Grenze zwischen hybriden Genetischen Algorithmen und Memetischen Algorithmen schwierig zu ziehen ist. Wegen der deutlichen Abgrenzung wurde der hier beschrie-

bene Vergleich auf der Basis eines konventionellen GA durchgeführt. Eine ausführlichere Beschreibung zu Memetischen Algorithmen findet man in [35], Literatur zu Genetischen Algorithmen findet man unter anderem in [11] und [34], weiterführende zu hybriden GA in [19].

Kapitel 5

Die Vorverarbeitung

Im Preprocessing wird der Problemgraph auf eine für die Problemstellung vorteilhafte Repräsentationsform, einen so genannten *Block-Cut-Tree* (BCT), abgebildet. Hierbei werden Artikulationsknoten identifiziert und Bereiche, die bereits zweifach zusammenhängend sind, zu Blöcken zusammengefasst. Anschließend werden Erweiterungskanten, die als redundant für eine optimale Lösung erkannt werden, eliminiert und Kanten, die in jeder Lösung vorkommen müssen, fixiert. Dies erfolgt in drei iterativ ausgeführten Schritten, um mögliche Vereinfachungen des BCT berücksichtigen zu können. Im Anschluss werden die in der Vorverarbeitung generierten Datenstrukturen an den Memetischen Algorithmus weitergereicht, der nun zur Lösungsfindung eingesetzt wird. Bei einfachen Probleminstanzen ist es möglich, dass das Preprocessing selbst schon eine optimale Lösung findet.

5.1 Der Block-Cut-Tree (BCT)

Gegeben ist ein ungerichteter, zusammenhängender Graph $G_0 = (V, E_0)$ mit Knotenmenge V und Kantenmenge E_0. G_0 selbst ist nicht zweifach zusammenhängend, jedoch Teilgraph von $G = (V, E)$, für den diese Eigenschaft gilt. Die Kantenmenge E setzt sich aus der Menge der vorgegebenen und somit fixen Kanten E_0 und der Menge der möglichen Erweiterungskanten E_a zusammen. Es gilt $E = E_0 \cup E_a$.

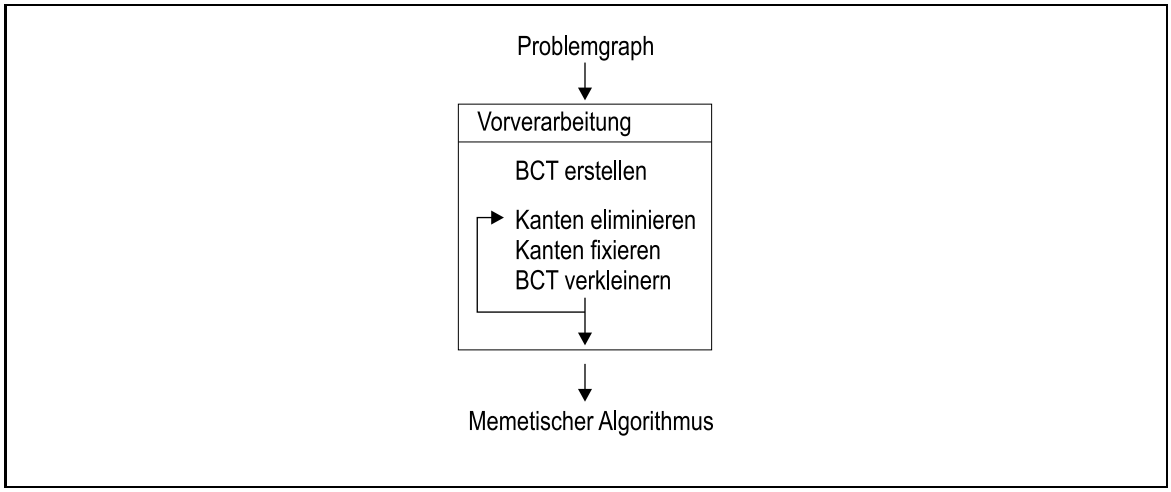


Abbildung 5.1: Überblick über den Aufbau und Ablauf der Vorverarbeitungsschritte.

Ein Block-Cut-Tree $T = (V_T, E_T)$ mit Knotenmenge V_T und Kantenmenge E_T ist ein ungerichteter Baum, der die Struktur von Artikulationsknoten und bereits zweifach zusammenhängenden Teilbereichen (Komponenten) des Graphen G_0 widerspiegelt [10]. Hierzu werden zwei verschiedene Arten von Knotentypen verwendet, nämlich Cut-Knoten und Block-Knoten. Es gilt $V_T = V_C \cup V_B$, und $V_C \cap V_B = \emptyset$, mit V_C als Menge aller Cut-Knoten und V_B als Menge aller Block-Knoten. Die Zuordnung der Knoten $v \in V$ des Graphen G_0 zu den Knoten $v' \in V_T$ erfolgt nun anhand der gefundenen zweifach zusammenhängenden Bereiche maximaler Größe in G_0. Alle Knoten $v \in V$ eines solchen Bereichs werden einem Block-Knoten $v' \in V_B$ im Block-Cut-Tree zugeordnet, wenn sie nicht Artikulationsknoten sind. Artikulationsknoten $v \in V$ werden durch je einen Cut-Knoten $v' \in V_C$ repräsentiert. Daraus ergeben sich folgende Eigenschaften für die Knoten $v' \in V_T$:

- Wenn $v' \in V_C$, dann ist ihm exakt ein Knoten $v \in V$ zugeordnet. Daraus folgt, dass die Anzahl an Artikulationsknoten in G_0 der Anzahl der Cut-Knoten des BCT entspricht.
- Wenn $v' \in V_B$, dann kann ihm eine beliebig große Menge an Knoten $v_i \in V$ zuge-

ordnet sein. Man beachte, dass die Menge der einem Block-Knoten zugeordneten Knoten aus G_0 auch leer sein kann, was genau dann der Fall ist, wenn der entsprechende zweifach zusammenhängende Bereich in G_0 nur aus Artikulationsknoten besteht.

Die Zuordnung von Knoten aus dem Graphen G_0 zu den entsprechenden Knoten des BCT wird später noch benötigt und daher bidirektional gespeichert.

Eine Kante im Block-Cut-Tree verbindet jeweils einen Block-Knoten mit einem Cut-Knoten. Ein Cut-Knoten $v' \in V_T$ und ein Block-Knoten $u' \in V_T$ werden in T dann und nur dann durch eine Kante verbunden, wenn der durch v' repräsentierte Artikulationsknoten $v \in V$ Teil desjenigen zweifach zusammenhängenden Bereiches in G_0 ist, den u' darstellt. Daraus folgt:

- In einem BCT alternieren Cut-Knoten und Block-Knoten auf jedem beliebig gewählten Pfad.
- Ein BCT ist immer ein Baum. Gäbe es noch zusätzliche Kanten, so könnte jeder daraus entstehende Zyklus in einem neuen, größeren Block-Knoten zusammengefasst werden.

Abbildung 5.2 auf Seite 31 zeigt ein Beispiel für den Aufbau eines BCT.

Nachdem der BCT vollständig aufgebaut ist, werden alle Erweiterungskanten $e \in E_a$ auf den BCT übertragen. Ein solcher Vorgang wird im Englischen mit "superimposition of edges" bezeichnet. Hierbei wird eine neue Kantenmenge E_A nach folgendem Schema aufgebaut: Für jede Kante $e \in E_a$ mit den Endknoten $(u, v) \in V$ wird eine korrespondierende Kante $e' \in E_A$ erzeugt und e zugeordnet. Die Endknoten $(u', v') \in V_T$ der neuen Kante e' werden so gewählt, dass sie den korrespondierenden Knoten von u und v entsprechen. Bei diesem Vorgang werden bereits manche redundanten Kanten identifiziert und entfernt. Dies ist in folgenden Fällen möglich: Kanten, die im BCT als Schlingen aufscheinen, Kanten, die zwei benachbarte Knoten im BCT miteinander verbinden und Kanten, die zwei zu demselben Block-Knoten benachbarte Cut-Knoten verbinden. Weiters werden von im BCT auftretenden Mehrfachkanten jeweils nur die kostengünstigen behalten. Eine ausführliche Betrachtung über das Eliminieren von

Kanten folgt in Abschnitt 5.3, nachdem im Abschnitt 5.2 erläutert wird, welche Anforderungen Kanten erfüllen müssen, um Cut-Knoten abdecken zu können. Die Zuordnung der Kanten E_a zu denen aus E_A wird gespeichert, um später generierte Lösungen wieder in Bezug zu dem ursprünglichen Problemgraphen setzen zu können. Abbildung 5.3 auf Seite 32 zeigt ein Beispiel für das Übertragen von Erweiterungskanten von E_a nach E_A.

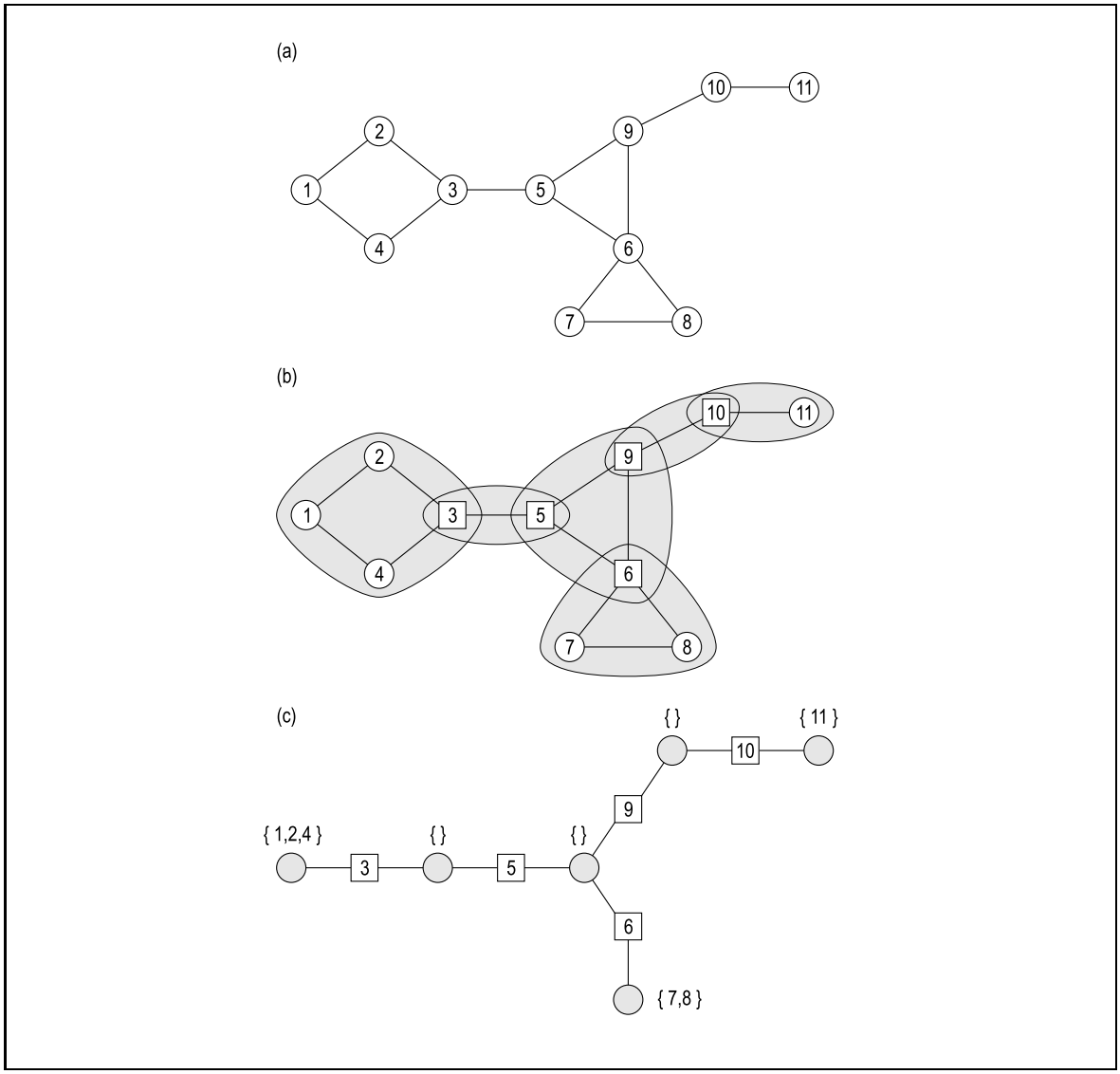


Abbildung 5.2: Der Aufbau eines Block-Cut-Trees. Abbildung a) zeigt den gegebenen Graphen G_0, zu dem ein BCT erstellt werden soll. Im darauf folgenden Bild b) sind die bereits zweifach zusammenhängenden Bereiche ermittelt worden und werden durch die graue Unterlegung der Knoten dargestellt. Weiters wurden die Artikulationsknoten des Graphen G_0 identifiziert, sie sind als quadratische Knoten hervorgehoben. Bild c) zeigt den daraus resultierenden Block-Cut-Tree mit der Zuordnung der einzelnen Knoten aus G_0.

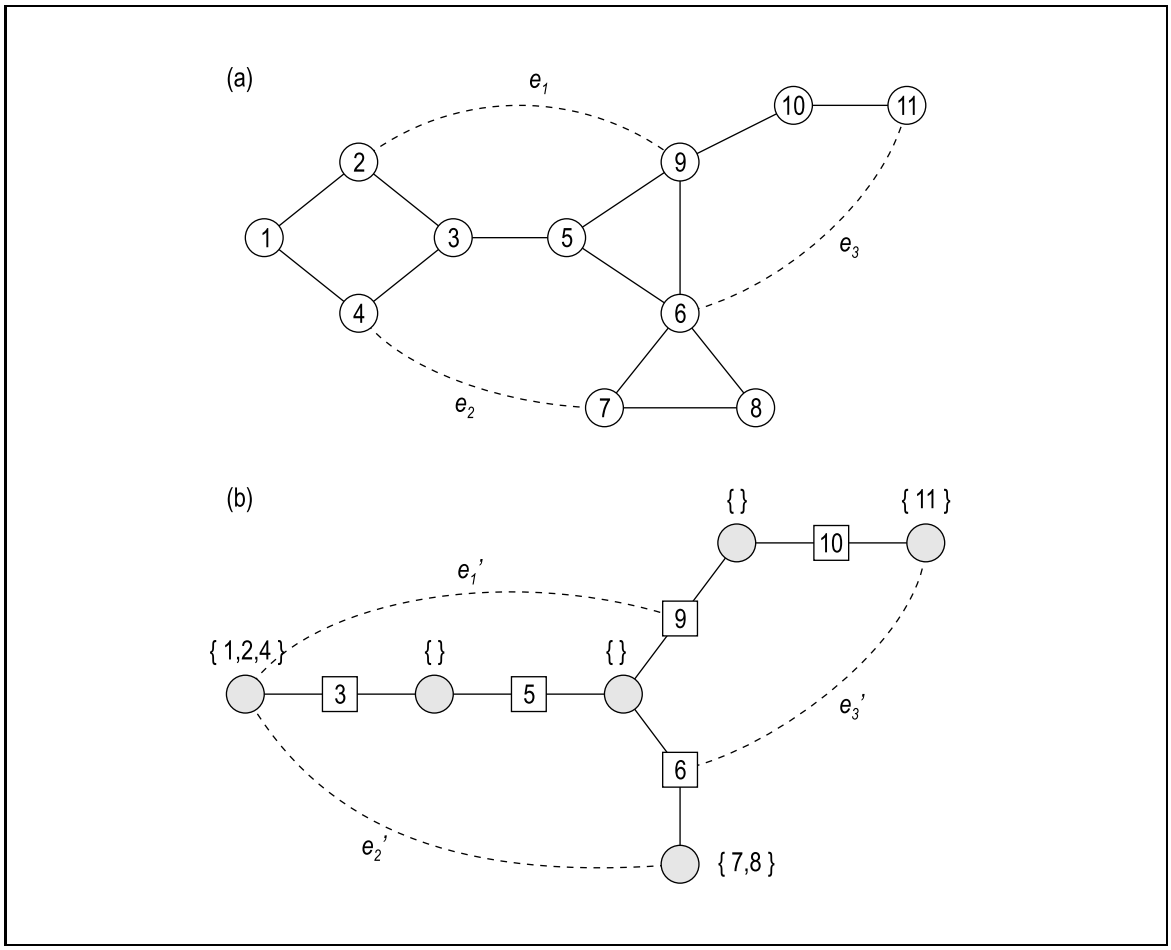


Abbildung 5.3: Übertragen der Erweiterungskanten in einen BCT. Bild a) zeigt den Graphen G_0, in dem beispielhaft drei Kanten $e_i \in E_a$ eingezeichnet wurden. In Bild b) sind die entsprechenden Kanten $e_i' \in E_A$ dargestellt, die an Hand der Zuordnung der Knoten im Graphen G_0 zu denen des Block-Cut-Tree generiert wurden. Betrachtet man die Kante e_1, so ist zu sehen, dass sie im Graphen G_0 die Knoten 2 und 9 verbindet. Im BCT wird ein Abbild dieser Kante generiert, indem eine Verbindung zwischen dem Block-Knoten $\{1, 2, 4\}$, dem die Knoten $1, 2$ und 4 zugeordnet sind, und dem Cut-Knoten 9 erstellt wird. Analog dazu erfolgt das Übertragen der Kanten e_2 und e_3.

5.1.1 Ein modifizierter BCT

Wie bereits besprochen alternieren in einem BCT Block-Knoten und Cut-Knoten auf jedem beliebig gewählten Pfad. Diese Eigenschaft wird von dem nachfolgenden Memetischen Algorithmus nicht benötigt. Um den Ablauf des Algorithmus zu beschleunigen, wird daher ein modifizierter BCT verwendet. Dieser weist folgende Vereinfachung zu einem üblichen Block-Cut-Tree auf: Leere Block-Knoten vom Grad zwei werden nicht aufgebaut. Anstatt dessen werden die adjazenten Cut-Knoten direkt über eine Kante verbunden. Diese Modifikation ist zwar nur gering, verringert aber in der Praxis die Anzahl der Knoten des BCT sehr deutlich, und dies um so stärker, je höher der Anteil von Artikulationsknoten im Vergleich zu bereits vorhandenen Blöcken in G_0 ist. Die somit verkleinerte Knotenmenge wirkt sich sowohl in den weiteren Vorverarbeitungsschritten als auch im MA positiv aus. Wenn nun im Folgenden von einem BCT gesprochen wird, so ist diese modifizierte Variante gemeint. Abbildung 5.4 auf Seite 34 stellt die BCT-Variante mit alternierenden Knoten der modifizierten gegenüber.

5.1.2 Der verwendete Algorithmus zum Aufbau eines modifizierten BCT

Die Art und Weise, in der hier ein BCT aufgebaut wird, ist recht spezifisch und nur im Zusammenhang mit den hier vorgestellten Vorverarbeitungsschritten sinnvoll, da eine für die nachfolgenden Reduktionsschritte benötigte Methode verwendet wird, um auf einfachem Weg einen wie zuvor beschriebenen, modifizierten BCT zu generieren.

Gehen wir zunächst davon aus, dass eine Funktion `VerkleinereBCT`(e') zur Verfügung steht, die einen bestehenden BCT korrekt modifiziert, wenn dem durch den Block-Cut-Tree repräsentierten Graphen G_0 eine weitere Kante hinzugefügt wird – oder anders formuliert: Der Übergang $G_0(V, E_0) \rightarrow G_0'(V, E_0 \cup e)$ hat eine strukturelle Änderung des BCT, $T(V_T, E_T) \rightarrow T'(V_T, E_T \cup e')$, zur Folge, was durch die Funktion `VerkleinereBCT`(e') korrekt ausgeführt wird. Die generelle Idee besteht darin, zunächst nur einen BCT für einen Teilgraph von G_0 zu generieren, genauer gesagt für einen Spannbaum von G_0. Wie wir gleich sehen werden, ist dies sehr einfach. Danach wird dieser BCT so lange verkleinert, bis er G_0 repräsentiert.

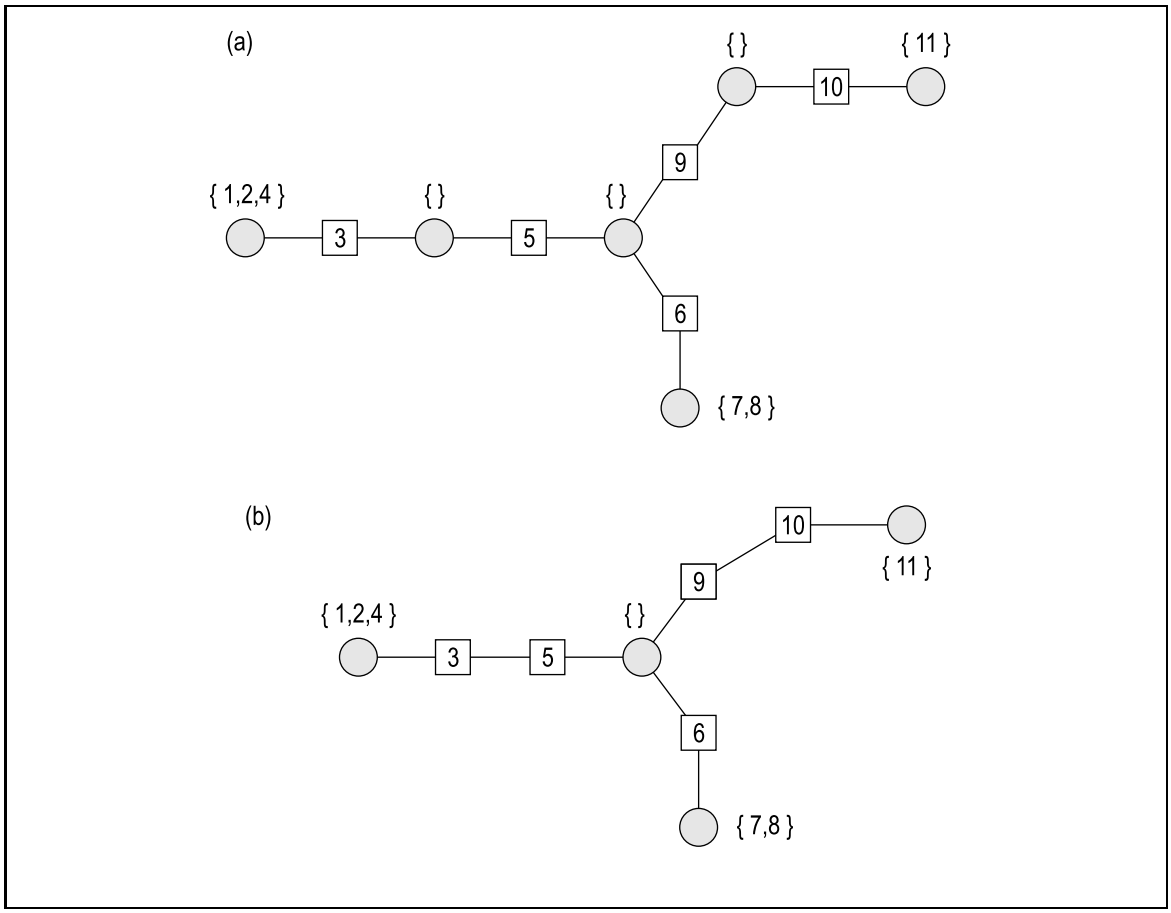


Abbildung 5.4: Gegenüberstellung beider BCT-Varianten. Abbildung a) zeigt einen BCT, in dem Block- und Cut-Knoten jeweils alternieren, Abbildung b) vernachlässigt diese Bedingung und spart dadurch zwei leere Block-Knoten ein. Je höher der Anteil an Artikulationsknoten im zu Grunde liegenden Problemgraphen ist, desto mehr wirkt sich diese Einsparung auf die Gesamtzahl an Knoten im erstellten BCT aus.

Mittels einer Tiefensuche ausgehend von einem willkürlich gewählten Knoten wird ein Spannbaum $ST(V, E_{DFS})$ in $G_0(V, E_0)$ fixiert, $E_{DFS} \subseteq E_0$. Dieser bildet das Grundgerüst für den zu erstellenden BCT und wird in die für den Block-Cut-Tree vorgesehene Datenstruktur übertragen, indem für jeden Knoten $v \in V$ ein Knoten $v' \in V_T$ generiert wird, dem der Knoten v zugewiesen wird. Für jede Kante $e \in E_{DFS}$ mit den Endpunkten (u, v) wird eine Kante e' angelegt, die die Knoten (u', v') verbindet. Als nächstes gilt es nun, die generierten Knoten $v' \in V_T$ in Block-Knoten und Cut-Knoten zu unterteilen beziehungsweise zusammenzufassen. Folgende einfache Überlegung führt uns zum Ziel: In einem Spannbaum gibt es keine Zyklen, deshalb sind die zweifach zusammenhängenden Teilbereiche minimal. Also besteht jeder Teilbereich aus exakt zwei Knoten. Weiters ist in einem Spannbaum jeder Knoten ein Artikulationsknoten, außer er ist ein Blattknoten. Da jeder Artikulationsknoten durch exakt einen Cut-Knoten im BCT repräsentiert werden soll, und jeder Block-Knoten all jene Knoten eines zweifach zusammenhängenden Bereiches zugeordnet bekommt, die nicht selbst Artikulationsknoten sind, werden diese Anforderungen bereits erfüllt, indem wir jeden Blattknoten des derzeitigen BCT als Block-Knoten generieren und den Rest als Cut-Knoten. Nun gilt es den BCT so zu modifizieren, dass er G_0 darstellt. Hierzu wird für jede Kante $e \in E_0 \setminus E_{DFS}$ ihr Abbild e' im BCT ermittelt und die Funktion `VerkleinereBCT`(e') aufgerufen. Diese Funktion eruiert den durch die Kante e' generierten Zyklus im BCT und restrukturiert diesen Teilbereich, indem sie Block-Knoten und nicht mehr benötigte Cut-Knoten zu einem größeren Block-Knoten kombiniert und die Zuordnung der Knoten zwischen dem Graphen G_0 und dem BCT aktualisiert. Eine genauere Beschreibung der Funktionsweise folgt in Abschnitt 5.3.3. Auf Seite 36 wird die Vorgangsweise in Pseudocode-Darstellung zusammengefasst, in Abbildung 5.6 auf Seite 37 wird Schritt für Schritt gezeigt, wie in der hier beschriebenen Weise ein BCT von einem Graphen G_0 erstellt wird. Vergleicht man nun den daraus resultierenden BCT mit jenem von Abbildung 5.2, so zeigt sich, dass die wiedergegebene Struktur dieselbe ist, aber auf zwei leere Block-Knoten in der modifizierten Variante verzichtet wird. Abbildung 5.4 auf Seite 34 stellt die Resultate beider BCT-Varianten einander gegenüber.

```
ErzeugeBCT(G_0(V, E_0))
BEGINN
    // Ermittle einen Spannbaum in G_0
    Wähle einen beliebigen Knoten v ∈ V
    ST(V, E_DFS) ← Tiefensuche(v)

    // Erzeuge T(V_T, E_T) für diesen Spannbaum
    FÜR ALLE v ∈ V
    BEGINN
        WENN v ein Blattknoten ist DANN
            erzeuge einen Block-Knoten v' ∈ V_T
        SONST
            erzeuge einen Cut-Knoten v' ∈ V_T

        setze abgebildet_in[v] = v'
    ENDE
    FÜR ALLE e(u, v) ∈ E_DFS
        erzeuge e' ∈ E_T mit Endknoten (abgebildet_in[u], abgebildet_in[v])
    FÜR ALLE e ∈ E_0 \ E_DFS
        VerkleinereBCT(abgebildet_in[u], abgebildet_in[v])
ENDE
```

Abbildung 5.5: Pseudocode zum Erzeugen eines BCT.

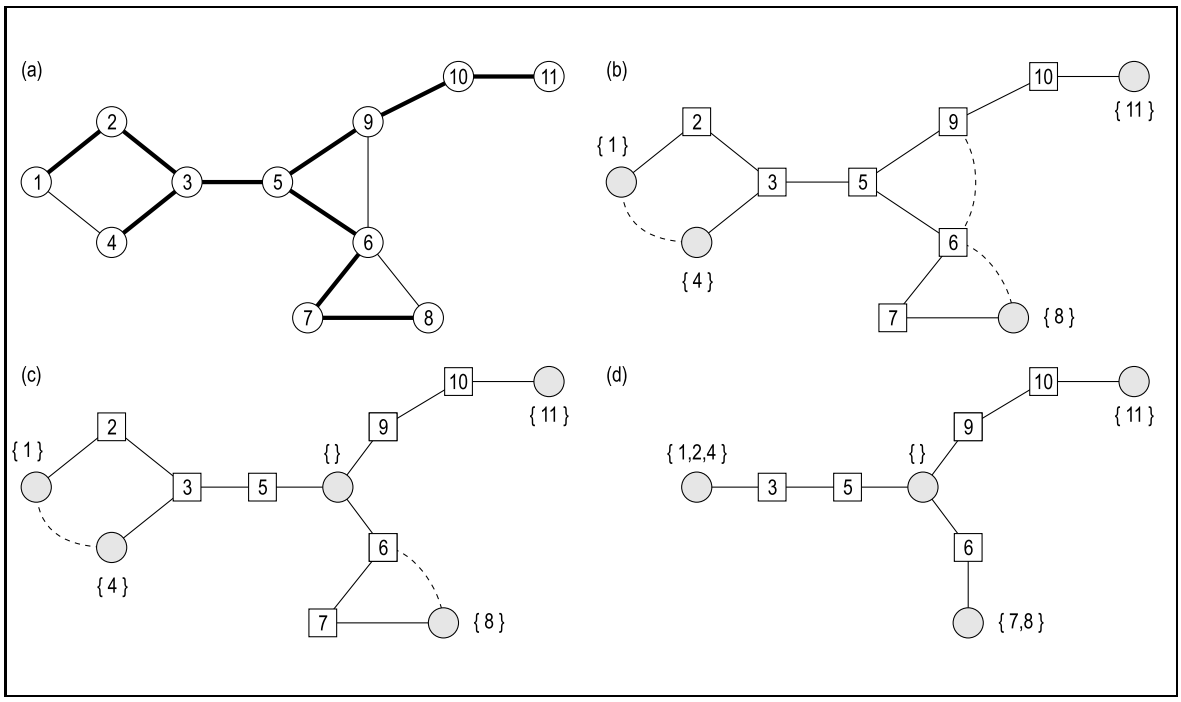


Abbildung 5.6: Der Aufbau eines modifizierten Block-Cut-Trees. Bild a) zeigt einen Graphen G_0, in dem bereits ein Spannbaum, dargestellt durch die hervorgehobenen Kanten, markiert wurde. In Bild b) wird der vorläufige BCT wiedergegeben. Für die Blattknoten des Spannbaumes wurden Block-Knoten generiert, für die Artikulationsknoten Cut-Knoten. Bislang nicht berücksichtigte Kanten von G_0 sind gestrichelt eingezeichnet. In Bild c) wurde nun die Kante $(6, 9)$ hinzugefügt und der BCT an Hand des entstandenen Zyklus umstrukturiert. Bild d) zeigt den resultierenden BCT.

5.2 Das Abdecken von Artikulationsknoten

Wenn ein Artikulationsknoten des Graphen G_0 entfernt wird, so entfernt man den Knoten selbst sowie alle zu ihm benachbarten Kanten. In der BCT-Darstellung geschieht dabei dasselbe. Der Cut-Knoten, der den Artikulationsknoten repräsentiert, wird entfernt, und mit ihm gehen alle zu ihm benachbarten BCT-Kanten verloren, da diese Verbindungen zu angrenzenden Teilbereichen symbolisieren, die die Kanten des Artikulationsknotens schaffen. Teil des Problems ist es also, für jene durch die Kanten des BCT dargestellten Verbindungen Alternativen aus der Menge der Erweiterungskanten zu wählen, was im Folgenden als *Abdecken einer Kante* bezeichnet wird.

Eine Kante $e_t \in E_T$ des BCT $T = (V_T, E_T)$ wird von einer Erweiterungskante $e \in E_A$ mit den Endknoten $(u, v) \in V_T$ genau dann abgedeckt, wenn der Pfad im BCT von u nach v über die Kante e_t führt (Abbildung 5.7 Bild a). Um einen Cut-Knoten $v_c \in V_T$ abzudecken, müssen alle zu ihm benachbarten Kanten abgedeckt werden. Dies allein ist im Allgemeinen aber noch nicht ausreichend. Wenn v_c aus T entfernt wird, zerfällt T in k nicht zusammenhängende Teilkomponenten $C_1^{v_c}, \ldots, C_k^{v_c}$, wobei k dem Grad von v_c entspricht (Abbildung 5.7 Bild b). Diese Teilkomponenten werden als Cut-Komponenten des Cut-Knotens v_c bezeichnet und durch die zu v_c benachbarten Kanten eindeutig bestimmt (Abbildung 5.7 Bild c). Eine Erweiterungskante $e \in E_A$ mit Endknoten $(u, v) \in V_T$ trägt zum Abdecken eines Cut-Knotens v_c genau dann bei, wenn sie zwei Kanten abdeckt, die in v_c enden, und somit zwei Cut-Komponenten $C_i^{v_c}$ und $C_j^{v_c}$ vereinigt. Eine solche Kante kann daher niemals v_c als Endpunkt haben (Abbildung 5.7 Bild d). Um v_c komplett abzudecken, werden exakt $k-1$ Erweiterungskanten benötigt, die zudem auch alle Cut-Komponenten $C_1^{v_c}, \ldots, C_k^{v_c}$ miteinander verbinden und so zu einer einzigen vereinen. Die Auswahl einer solchen Menge von Kanten wird im Folgenden als *Abdecken eines Cut-Knotens* bezeichnet. Abbildung 5.8 auf Seite 40 zeigt ein Beispiel für diesen Vorgang.

Für jeden Cut-Knoten v_c sei nun $A(v_c) \subseteq E_A$ die Menge aller Erweiterungskanten, die beitragen v_c abzudecken, indem sie zwei Cut-Komponenten verbinden. Des Weiteren sei für jede Erweiterungskante $e \in E_A$, $R(e)$ die Menge aller Cut-Knoten, bei denen e beiträgt sie abzudecken. Für den in Abschnitt 6 vorgestellten Memetischen Algorithmus werden alle Mengen $R(e)$ und $A(v_c)$ als Hilfsdatenstrukturen explizit aufgebaut, wobei

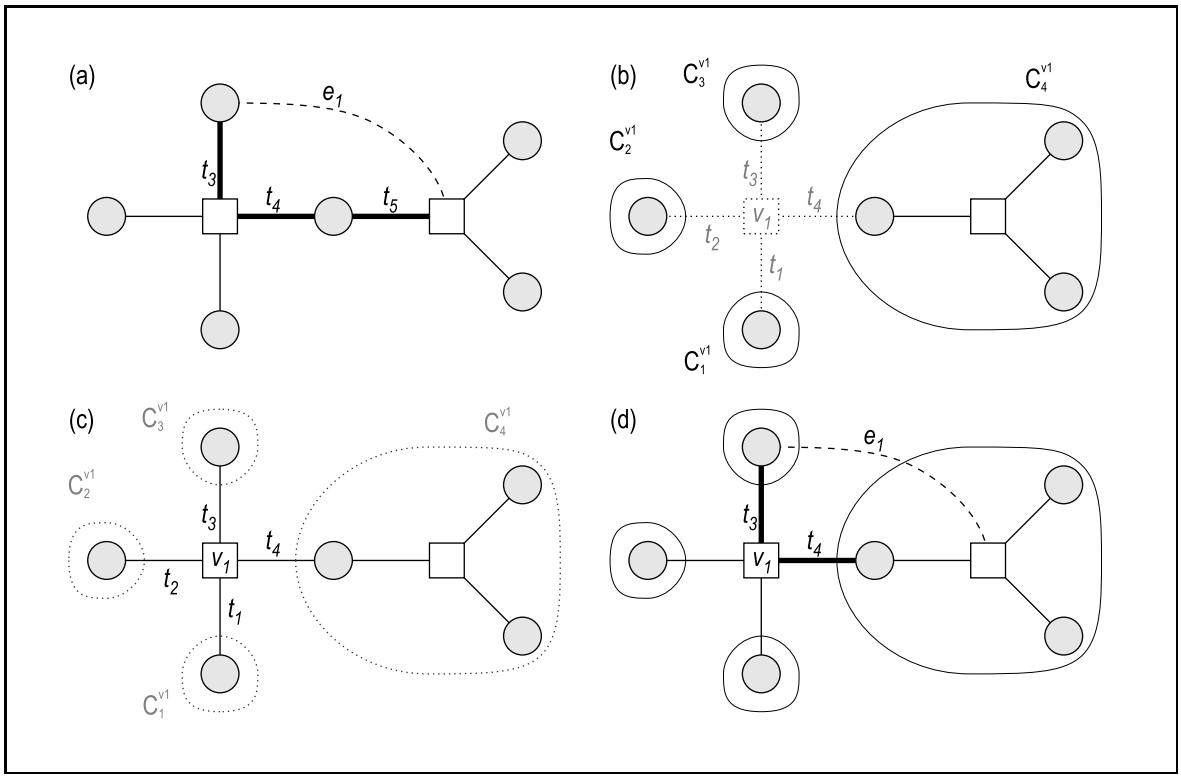


Abbildung 5.7: Veranschaulichende Beispiele zum Abdecken von Cut-Knoten. Dargestellt wird ein BCT T mit zwei Cut-Knoten (quadratisch) und sechs Block-Knoten (kreisförmig). Erweiterungskanten $e_i \in E_A$ werden gegebenenfalls strichliert eingezeichnet.
(a) Die Kanten $t_3, t_4, t_5 \in E_T$ werden von $e_1 \in E_A$ abgedeckt. (b) Wenn $v_1 \in V_C$ entfernt wird, zerfällt T in die vier Cut-Komponenten $C_1^{v1} \dots C_4^{v1}$. (c) Diese vier Cut-Komponenten werden durch die zu v_1 benachbarten Kanten $t_1 \dots t_4 \in E_T$ eindeutig bestimmt. (d) Die Kante $e_1 \in E_A$ trägt zum Abdecken von v_1 bei, da sie die zwei zu v_1 benachbarten BCT-Kanten t_3 und t_4 abdeckt und somit zwei Cut-Komponenten verbindet.

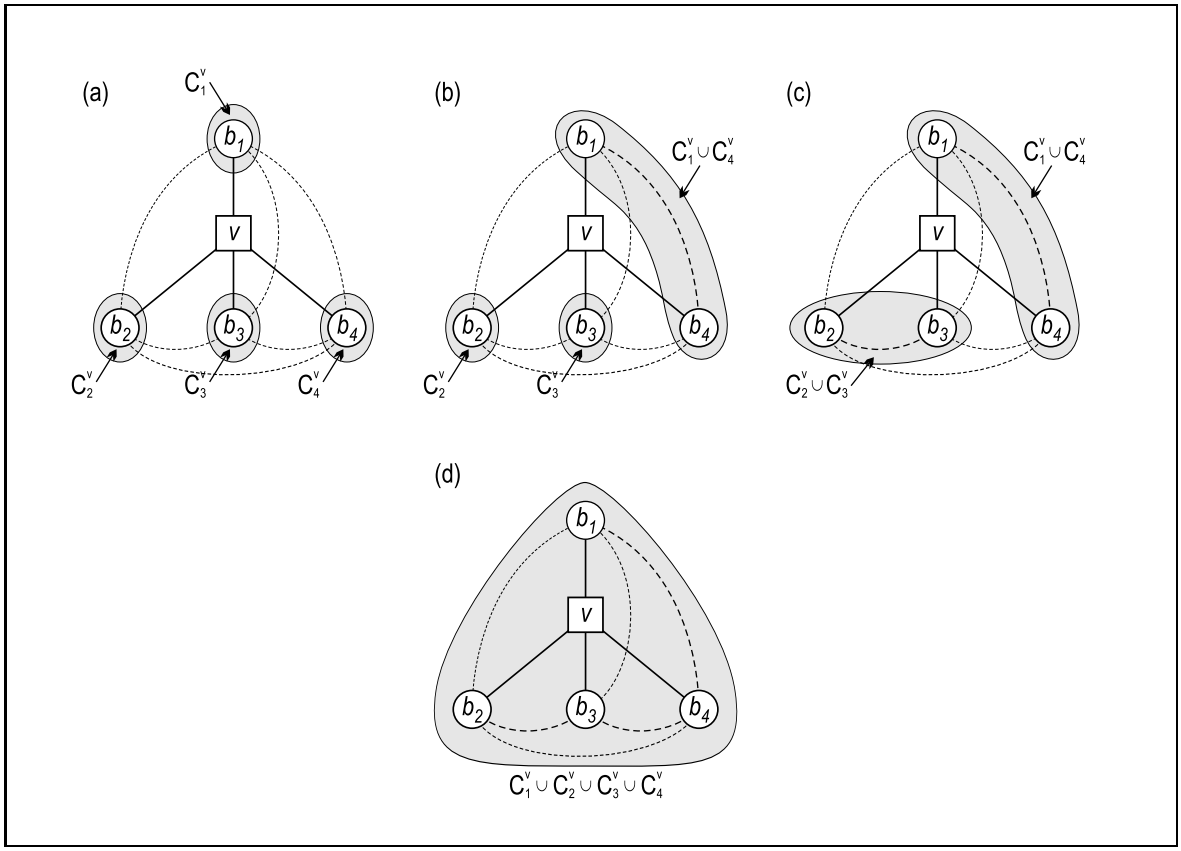


Abbildung 5.8: Schrittweise Abdecken eines Cut-Knotens. (a) zeigt einen Cut-Knoten $v \in V_C$ und die Cut-Komponenten $C_1^v \dots C_4^v$, die durch seine benachbarten Kanten im BCT eindeutig bestimmt werden. Jede dieser Komponenten besteht in diesem kleinen Graphen nur aus einem Block-Knoten, es könnten aber genau so gut auch umfangreiche Teilgraphen sein. Die Erweiterungskanten, die beitragen können diesen Cut-Knoten abzudecken, sind als gestrichelte Kanten dargestellt. In (b) wurde nun jene Kante gewählt und der Lösungsmenge hinzugefügt, die b_1 mit b_4 verbindet und somit die Cut-Komponenten C_1^v und C_4^v vereint. Durch das Hinzufügen der Kante zwischen b_2 und b_3 in (c) sind nun alle dem Knoten v benachbarten Kanten des BCT abgedeckt, allerdings würde der Graph noch immer in zwei Komponenten zerfallen, wenn Knoten v entfernt wird. Erst durch eine dritte Kante wird v vollständig abgedeckt. In (d) wird hierzu die Erweiterungskante zwischen b_3 und b_4 gewählt, womit nun alle Cut-Komponenten vereint sind und der abgebildete Graph zweifach zusammenhängend ist.

jeder Eintrag $e \in A(v_c)$ aus den beiden durch e abgedeckten BCT-Kanten besteht. Diese geben eindeutig an, welche Cut-Komponenten miteinander verbunden werden.

Um die Referenzen aufzubauen wird zunächst eine Tiefensuche für den BCT durchgeführt. Dabei wird für jeden Knoten seine Tiefe und seine Referenz zu seinem Vorgängerknoten gespeichert, damit der Pfad im BCT zwischen zwei beliebigen Knoten effizient bestimmt werden kann. Danach kann die Berechnung aller Mengen $A(v_c)$ und $R(e)$ in einer Zeit proportional zu $O(|E_A| \cdot |V_T|)$ durchgeführt werden. Der Speicherverbrauch für die entsprechenden Datenstrukturen beträgt maximal $O(|E_A| \cdot |V_T|)$. Im Allgemeinen ist T ein nicht entarteter Baum mit einer durchschnittlichen Pfadlänge von $O(\log |V_T|)$. In diesem Fall beträgt der Speicherbedarf der Mengen $A(v_c)$ und $R(e)$ $O(|E_A| \log |V_T|)$. Die Vorverarbeitung speichert weiters für jeden Eintrag e von $A(v_c)$ eine Referenz zu den beiden Kanten, die benachbart zu v_c sind und von e abgedeckt werden. Diese bestimmen eindeutig die beiden Cut-Komponenten, die durch e verbunden werden.

Im folgenden Memetischen Algorithmus wird oft eine Überprüfung, ob ein bestimmter Cut-Knoten durch eine gegebene Menge an Erweiterungskanten $S \in E_A$ abgedeckt wird, notwendig. Mit Hilfe der vorberechneten Mengen A und R und unter Verwendung einer Union-Find-Datenstruktur mit weight-balancing und path-compression [1] kann diese Überprüfung in nahezu linearer Zeit $O(|S|)$ durchgeführt werden.

Meistens ist der Grad der Cut-Knoten $v_c \in V_C$ kleiner als vier. In diesen Fällen erübrigt sich die Verwendung einer Union-Find-Datenstruktur, da eine Überprüfung ausreicht, ob alle zu v_c benachbarten Kanten des BCT durch Erweiterungskanten, die selbst nicht zu v_c benachbart sind, abgedeckt werden.

5.3 Verkleinern des Suchraums

Je weiter der Suchraum für den Memetischen Algorithmus eingeengt werden kann, desto einfacher und kürzer wird im Allgemeinen die nachfolgende Suche nach einer optimalen Lösung. Aus E_A werden deshalb all jene Kanten $e \in E_A$ entfernt, die nicht zum Abdecken eines Cut-Knotens beitragen können und daher für eine optimale Lösung

irrelevant sind. Im Speziellen sind das nun Kanten, die eine Schleife bilden, einen Cut-Knoten mit einem benachbarten Knoten verbinden, wobei es irrelevant ist, ob es sich dabei um einen Block-Knoten oder einen Cut-Knoten handelt, oder Kanten, die zwei Cut-Knoten über einen Block-Knoten hinweg verbinden. Weiters können Mehrfachkanten im Graph $G_A = (V_T, E_T \cup E_A)$ bis auf eine mit minimalen Kosten gelöscht werden. Auf diese Weise wird G_A ein schlichter Graph. In Abbildung 5.9 auf Seite 43 werden Beispiele für die eben genannten Fälle gezeigt.

Diese Schritte geschehen bereits beim Übertragen der Kanten von E_a nach E_A und werden, sofern der BCT in seiner Struktur durch das Preprocessing verändert wird, in den nachfolgenden Reduktions-Methoden berücksichtigt. Es gelten folgende Regeln beim Aufbau der Menge E_A:

FÜR ALLE $e \in E_A$ mit Endknoten $(u, v) \in V_T$
BEGINN
    WENN $u == v$ DANN entferne e
    WENN benachbart(u, v) DANN entferne e
    WENN benachbart(u, w) UND benachbart(w, v)
        UND $u, v \in V_C$ UND $w \in V_B$ DANN
        entferne e
ENDE

Zusätzlich zu diesen einfachen Reduktionen werden folgende Schritte durchgeführt, die eine Weiterentwicklung jener Vorverarbeitungsschritte darstellen, wie sie von G. Raidl und I. Ljubić für das E2AUG-Problem vorgestellt wurden [42].

5.3.1 Eliminieren von Kanten

Wenn es zwei Kanten gibt, die denselben Bereich abdecken aber unterschiedlich viel kosten, dann kann die teurere verworfen werden. Etwas genauer formuliert: Wenn es zwei Kanten $e_i, e_j \in E_A$ gibt, wobei die Kosten $Kosten(e_i) \leq Kosten(e_j)$ sind, und e_i zumindest all jene Kanten abdeckt, die auch von e_j abgedeckt werden, dann wird e_j

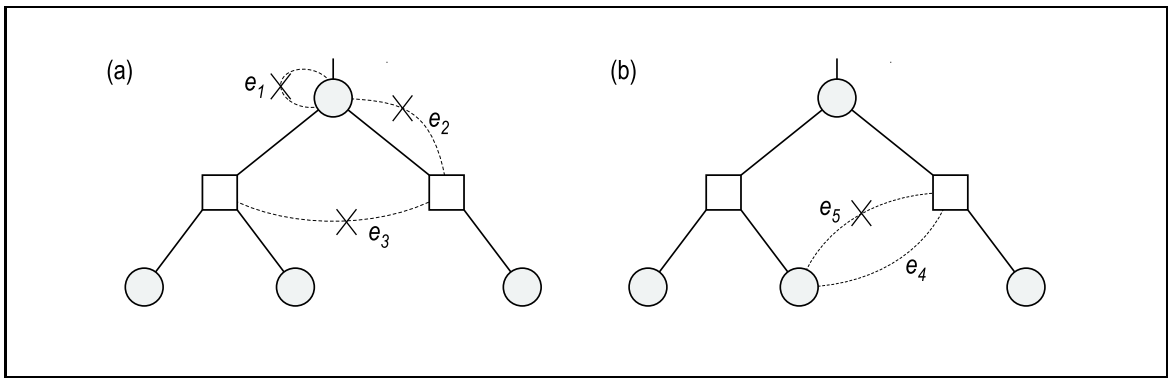


Abbildung 5.9: Die beiden Bilder zeigen Beispiele von Erweiterungskanten, die bereits beim Aufbau des BCT gelöscht werden können. Schlingen wie e_1 und Kanten zwischen zwei benachbarten Knoten wie e_2 können nicht zum Lösen der Problemstellung beitragen. Auch die Kante e_3, die zwei Cut-Knoten über einen Block-Knoten hinweg verbindet, ist ungeeignet. In (b) sehen wir zwei Mehrfachkanten e_4 und e_5, von denen nur die kostengünstigere behalten wird.

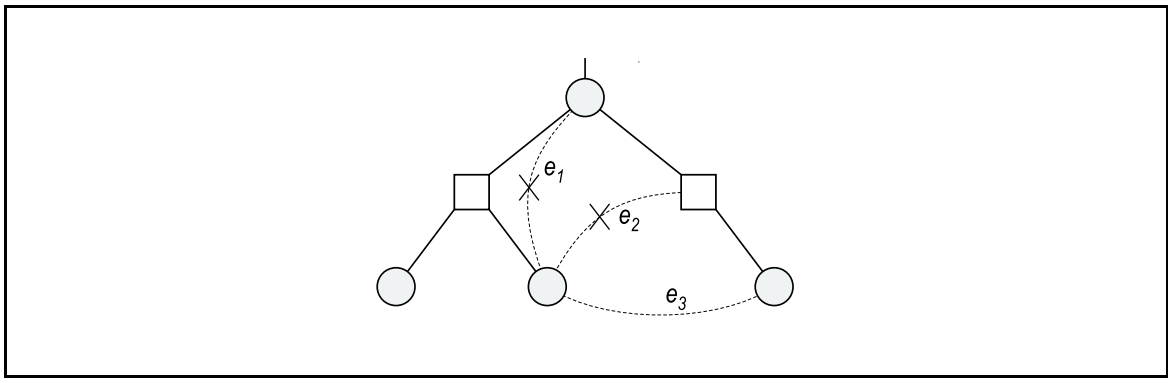


Abbildung 5.10: Eliminieren von Kanten. Die Erweiterungskante e_3 deckt einen größeren Bereich des dargestellten Teilgraphen ab als e_1 und e_2. Gilt nun $Kosten(e_3) \leq Kosten(e_2)$ und $Kosten(e_3) \leq Kosten(e_1)$, so können die Kanten e_1 und e_2 entfernt werden.

für eine optimale Lösung nicht benötigt und kann aus E_A entfernt werden. Abbildung 5.10 auf Seite 43 zeigt hierzu ein Beispiel.

Das Finden all solcher Kanten basiert auf einem Dynamic-Programming-Algorithmus, der zum Berechnen von Distanzwerten für einen Algorithmus von Frederickson und Jájá [13] verwendet wurde. Das Verfahren ist in Abbildung 5.11 auf Seite 45 in Pseudocode-Darstellung zusammengefasst und funktioniert dabei wie folgt: Zunächst müssen Informationen über alle möglichen Pfade und deren Länge generiert werden. Hierzu wird eine Tiefensuche, oder auch Depth-First-Search (DFS) genannt, von jedem Knoten des BCT aus gestartet. Sie speichert für alle ungeordneten Tupel $(u, v) \in V_T$ die Pfadlänge und für alle geordneten den vorletzten Knoten auf dem Weg von u nach v ($suc(u, v)$). Eine Datenstruktur zur Speicherung der aktuellen minimalen Kosten ($c[u, v]$) für jeden Teilpfad wird angelegt und mit ∞ initialisiert, sofern nicht die Kosten einer Erweiterungskante eingetragen werden können. Die ungeordneten (u, v) Tupel werden beginnend bei denen maximaler Pfadlänge abgearbeitet. Die Kosten, um den Pfad (u, v) abzudecken, werden nun ausgelesen (c_0) und mit denen der beiden um eins kürzeren Teilpfade verglichen. Sind die Kosten des längeren Pfades geringer oder gleich, wird – falls vorhanden – die Kante, die sich über den kürzeren Pfad spannt, gelöscht, und die Kosten für den kürzeren Pfad werden dem des längeren gleichgesetzt. Das Verfahren terminiert, sobald Tupel der Pfadlänge zwei bearbeitet werden, da kürzere Kanten ohnehin zur Lösung des V2AUG-Problems irrelevant sind. Der Zeitaufwand insgesamt ist proportional zu $O(|V_T|^2)$.

5.3.2 Fixieren von Kanten

Eine Kante $e \in E_A$ muss in jedem Fall Teil einer gültigen Lösung für das V2AUG-Problem sein, wenn sie die einzige Möglichkeit darstellt, eine Cut-Komponente C_i^v eines Cut-Knotens v mit irgendeiner anderen Cut-Komponente von v zu verbinden. Es wird also für jeden Cut-Knoten v die Menge $A(v)$ betrachtet und nach solchen Verbindungen Ausschau gehalten. Wird eine entsprechende Kante gefunden, so wird sie aus der Menge E_A entfernt und zur Menge E_T hinzugefügt. Die korrespondierende, ursprüngliche Erweiterungskante $e_a \in E_a$ wird markiert und ist somit fixer Bestandteil jeglicher zukünftig generierten Lösung. Das Fixieren benötigt die Zeit $O(|V_T| \cdot |E_A|)$.

Vorbereitungen: $\texttt{suc}(u, v)$ liefert den vorletzten Knoten auf dem Weg von u nach v.
$c[u, v]$ wird wie folgt initialisiert:

$c[u, v] = \texttt{Kosten}(e)$, wenn es eine Kante $e \in E_A$ mit Endknoten (u, v) gibt

$c[u, v] = \infty$ sonst

```
KantenElimination()
BEGINN
    FÜR ALLE Tupel (u, v), beginnend mit denen maximaler Pfadlänge
    BEGINN
        c_0 = c[u, v]
        WENN c_0 ≤ c[u, suc(u, v)] DANN
        BEGINN
            WENN es eine Kante e ∈ E_A mit Endknoten (u, suc(u, v)) gibt DANN
                entferne e
            c[u, suc(u, v)] = c_0
        ENDE
        WENN c_0 ≤ c[suc(v, u), v] DANN
        BEGINN
            WENN es eine Kante e ∈ E_A mit Endknoten (suc(v, u), v) gibt DANN
                entferne e
            c[suc(v, u), v] = c_0
        ENDE
    ENDE
ENDE
```

Abbildung 5.11: Pseudocode für die Kantenelimination.

Abbildung 5.12 zeigt ein Beispiel hierzu.

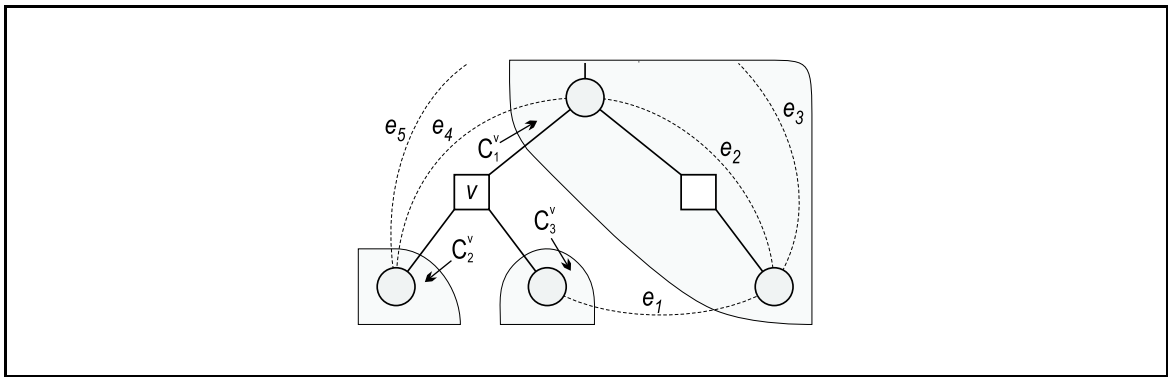


Abbildung 5.12: Fixieren von Kanten. Die Erweiterungskante e_1 ist die einzige Möglichkeit, die beiden Cut-Komponenten C_1^v und C_3^v zu verbinden. Daher muss sie Teil von jeder gültigen Lösung sein und kann als fester Lösungsbestandteil markiert werden.

5.3.3 Verkleinern des BCT

Durch das Fixieren einer Kante $e \in E_A$ mit Endknoten $(u, v) \in V_T$ entsteht ein Zyklus im BCT T. Dieser Zyklus formt nun einen neuen Bereich, der zweifach zusammenhängend ist. Es ist daher notwendig den BCT zu modifizieren, um seine Baumstruktur und anderen Eigenschaften zu erhalten. Da es nicht erstrebenswert ist, den gesamten BCT für ein großes Beispiel neu zu generieren, nur weil eine weitere Kante hinzugefügt wurde, wird nur der betroffene Teil umstrukturiert. Hierbei können nun weitere Kanten gefunden werden, die entfernt werden können. Abbildung 5.13 auf Seite 47 zeigt hierzu ein Beispiel, Abbildung 5.14 auf Seite 49 stellt das Verfahren als Pseudocode dar und wird nachfolgend beschrieben.

Zur Umstrukturierung wird der durch das Hinzufügen der Kante $e(u, v)$ neu gebildete Zyklus Z im BCT T ermittelt. Dieser definiert sich durch die Endknoten der Kante e. Z sei nun die Menge aller im Zyklus enthaltenen Knoten, es gilt $Z \subseteq V_T$. Ein neuer Block-Knoten $b \in V_B$ wird generiert. Die Umstrukturierung erfolgt nach diesen Regeln:

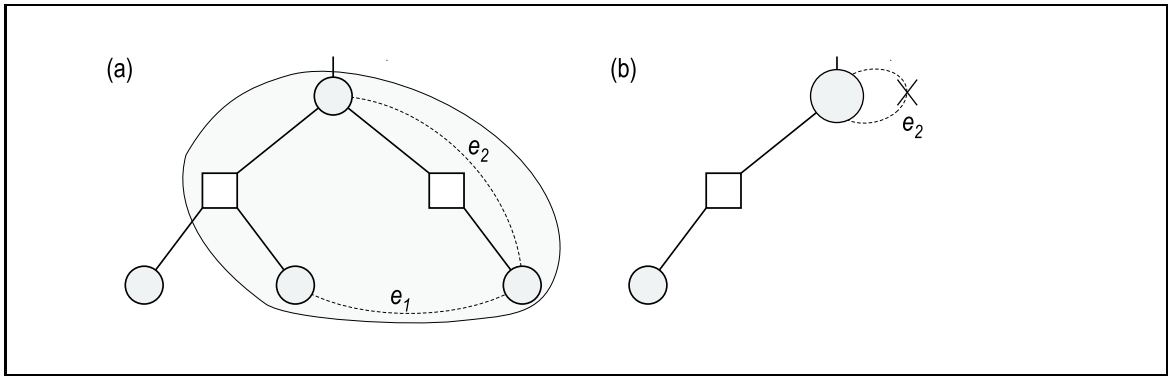


Abbildung 5.13: Verkleinern des BCT. Bild a) zeigt den durch das Fixieren der Kante e_1 gebildeten Zyklus. Durch das Zusammenfassen dieses Bereiches wird Kante e_2 als redundant erkannt, da sie nur noch innerhalb des neu gebildeten Block-Knotens verbindet. In (b) sieht man den zusammengefassten Block-Knoten mit der überflüssigen Schlinge e_2.

- Alle Cut-Knoten $v_c \in Z$, die durch das Hinzufügen von e vollständig abgedeckt werden, werden dem neuen Block-Knoten b hinzugefügt, $b = b \cup v_c$. Ein Cut-Knoten ist dann von e vollständig abgedeckt, wenn er vom Grad zwei ist und nicht u oder v ist.

- Alle Block-Knoten $v_b \in Z$ werden ebenfalls dem neuen Block-Knoten b zugeordnet, $b = b \cup v_b$. Alle Kanten des BCT zwischen v_b und einem beliebigen anderen BCT-Knoten $x \notin Z$ werden direkt auf b gerichtet, $e_t(x, v_b) \rightarrow e_t(x, b)$.

- Alle weiteren Cut-Knoten $v_c \in Z$, die von e nur teilweise abgedeckt werden, werden b nicht zugeordnet, sondern werden jeweils durch eine neue Kante $e_t \in E_T,\ e_t = (v_c, b)$ mit dem Block-Knoten b verbunden.

- Alle Kanten $e_t \in E_T$, die die Knoten im Zyklus Z verbunden haben, werden entfernt.

- Alle Erweiterungskanten $e_A \in E_A$, die in einem Knoten enden, der nun b zugeordnet ist, werden auf b übertragen. Hierbei werden Erweiterungskanten, für die

nun $e_A = (b, b)$ gilt, aus der Menge E_A entfernt. Abbildung 5.13 auf Seite 47 zeigt hierzu ein Beispiel.

Auf Grund der strukturellen Veränderungen des BCT kann es nun sein, dass weitere Kanten zum Eliminieren oder Fixieren gefunden werden können. Daher werden die drei Reduktionsschritte iteriert, bis keine weitere Verkleinerung des BCTs möglich ist.

5.4 Zusammenfassung

In der Vorverarbeitung wird der Suchraum für den nachfolgenden MA effektiv verkleinert, indem zuerst der Problemgraph in einen Block-Cut-Tree umgewandelt wird. Hierbei verringert sich die Komplexität der Problemstellung der Erweiterung eines Graphen auf zweifachen Zusammenhang auf die Erweiterung eines Baumes. Durch das Zusammenfassen von bereits zweifach zusammenhängenden Teilbereichen kann sich die Anzahl der zu berücksichtigenden Knoten verringern, dies ist jedoch vom Problemgraph abhängig. Auf jeden Fall ermöglicht diese Darstellung eine erste Reduktion der Menge der Erweiterungskanten, die bereits beim Aufbau des BCT durchgeführt wird. Hiernach folgen drei iterativ durchgeführte Reduktionsschritte. Die Kantenelimination entfernt überteuerte Kanten, wenn es andere gibt, die zumindest den gleichen Pfad abdecken. Die Kantenfixierung findet Erweiterungskanten, die in jedem Fall in einer optimalen Lösung enthalten sein müssen, und markiert diese, um sie von vornherein in einer Lösung zu berücksichtigen. Der letzte Schritt zur Reduktion besteht im Verkleinern des BCT, wodurch wiederum neue Kanten zum Entfernen entdeckt werden können und die zu betrachtende Knotenanzahl verringert wird.

Für den Rechenaufwand der gesamten Vorverarbeitung ist $O(|V_T|^2 \cdot |E_A|)$ eine obere Schranke, da das Eliminieren von Kanten, das Fixieren von Kanten und das Verkleinern des BCT theoretisch bis zu $O(V_T)$ mal ausgeführt werden kann. Allerdings geschieht dies nur in extrem seltenen Fällen, und der zu erwartende Rechenaufwand ist geringer, wie die empirischen Daten in Kapitel 7 zeigen.

`VerkleinereBCT`(e)
BEGINN
ermittle die Endknoten u, v von $e \in E_A$
u und v definieren den neu gebildeten Zyklus Z im BCT

erzeuge einen neuen leeren Block-Knoten $b \in V_B$

FÜR ALLE Knoten $w \in Z$ beginnend bei u
BEGINN
lösche Kante $e_t \in E_T$, die w mit seinem Vorgängerknoten in Z verbindet.
WENN $w \in V_B \vee \big(w \in V_C \wedge grad(w) == 2 \wedge w \neq u \wedge w \neq v\big)$
DANN
übertrage alle $e_A \in E_A$ mit Endknoten (w, x)
nach (b, x), $x \in V_T$
übertrage alle $e_t \in E_T$ mit Endknoten (w, x)
nach (b, x), $x \in V_T \wedge x \notin Z$
$b = b \cup w$
SONST
erzeuge eine neue Kante $e_t \in E_T$ mit Endknoten (w, b)
ENDE
ENDE

Abbildung 5.14: Pseudocode zum Verkleinern des BCT.

Kapitel 6

Der Memetische Algorithmus

Der hier vorgestellte Algorithmus gehört zur breiten Klasse der so genannten Local-Search-Based Memetischen Algorithmen. Er arbeitet mit einer Population von Lösungen, und eine Optimierungsfunktion in Form einer lokalen Suche stellt sicher, dass nur gültige und lokal optimale Lösungen weiterverarbeitet werden. Der schematische Aufbau des Algorithmus in Pseudocode ist in Abbildung 6.1 auf Seite 51 wiedergegeben.

In diesem Kapitel werden die einzelnen Teile des Algorithmus besprochen. Zur Zeit der Realisierung war es noch nicht absehbar, welche Strategie den bestmöglichen Erfolg erbringen würde. Daher wurden verschiedene Varianten implementiert und getestet. In Kapitel 7.3 werden der Auswahlvorgang zwischen den Varianten und die daraus resultierenden Ergebnisse erläutert.

6.1 Generelle Strategien

6.1.1 Wahl einer Kodierung

Eine Lösung $S \subseteq E_A$ wird direkt als Menge der verwendeten Kanten repräsentiert. Hierzu werden Referenzen der benötigten Erweiterungskanten in Hash-Tabellen, genauer gesagt in Hash-Mengen (siehe z.B. [8] für eine Realisierung) gespeichert. Der Vorteil dieser

```
Memetischer_Algorithmus für das V2AUG-Problem()
BEGINN
    // Initialisiere die Population
    FÜR ALLE Individuen S ∈ pop
    BEGINN
        S ← Initialisiere (S)
        S ← Lokale_Optimierung (S)
        Bewerte (S)
    ENDE

    // Erzeuge eine neue Generation
    WIEDERHOLE
    BEGINN
        S1, S2 ← Selektion zweier Individuen S1, S2 ∈ pop für die Rekombination
        Sc ← Rekombination (S1, S2)
        Sc ← Lokale_Optimierung (Sc)
        Sc ← Mutation (Sc)
        Sc ← Lokale_Optimierung (Sc)
        Bewerte (Sc)
        pop' ← pop;  das schlechtest bewertete Individuum wird durch Sc ersetzt.
    ENDE
    BIS das Abbruchkriterium erfüllt ist
ENDE
```

Abbildung 6.1: Aufbau des MA für das V2AUG-Problem.

Art der Repräsentationsform liegt im geringen Speicherverbrauch ($O(|S|)$) und in der Effizienz, mit der Individuen manipuliert werden können. Das Hinzufügen, Überprüfen und Löschen einer Erweiterungskante zu einer Lösung erfolgt in konstanter erwarteter Zeit. Bei einer lokal optimalen Lösung des V2AUG-Problems gilt immer $|S| < |V|$. Bei zunehmender Anzahl von Knoten gilt im Allgemeinen sogar $|S| \ll |V|$. Der Speicherverbrauch pro Individuum lässt sich somit mit $O(|S|) = O(|V|)$ abschätzen.

6.1.2 Selektion

Die Selektion für die Rekombination erfolgt turnierbasiert [6, 7]. Hierzu wird eine kleine Anzahl von k Kandidaten aus der aktuellen Population zufällig gewählt, danach entscheiden die Kosten der durch das Individuum repräsentierten Lösung über den Ausgang des Turniers. Die Wahl der Kandidaten erfolgt mit Zurücklegen, wodurch sichergestellt wird, dass auch das schlechteste Individuum gewählt werden kann.

6.1.3 Ersetzungsschema

Pro Generationsschritt wird nur ein einzelnes neues Individuum generiert, es wird also eine Steady-State-Strategie angewandt. Das neue Individuum ersetzt in der Regel das schwächste der Population. Eine Ausnahme hierzu wird allerdings gemacht, wenn das neue Individuum ident mit einem bereits vorhandenen ist. In diesem Fall wird es selbst verworfen. Dies geschieht, um eine möglichst breit gefächerte Population zu erhalten und einem vorzeitigen Konvergieren des Verfahrens entgegenzuwirken [41]. Individuen der aktuellen Generation werden mittels einer Hash-Tabelle verwaltet, womit das Überprüfen auf Duplikate effizient möglich ist. Ein Heap ermöglicht das rasche Auffinden des jeweils schlechtest bewerteten Individuums innerhalb der Population.

6.1.4 Abbruchkriterium

Der Algorithmus beendet seine Suche, wenn innerhalb der letzten n Generationen keine Verbesserung der derzeit besten Lösung erreicht wurde. Alternativ kann auch das

Erreichen einer bestimmten Anzahl von Generationsschritten als Abbruchkriterium verwendet werden. Diese Alternative ist insbesondere für die Verwendung von Early-Restart Strategien interessant, bei denen versucht wird, mit mehreren kurzen Läufen ein besseres Ergebnis zu erzielen als mit einem langen (siehe [14] für ein konkretes Beispiel, auch [30]).

6.2 Lokale Optimierung

Die Lokale Optimierung von generierten Lösungen spielt eine zentrale Rolle im Ablauf des Algorithmus. Sie kommt nach der Initialisierung, Rekombination und Mutation eines Individuums zur Anwendung, um stets nur gültige und lokal optimale Lösungen weiterzuverwenden. Hierzu entfernt sie aus einer gültigen Lösung S redundante Kanten, bis S nicht weiter reduziert werden kann, ohne ungültig zu werden. Die Vorgangsweise der Lokalen Optimierung wurde auf die Anforderungen des MA abgestimmt und optimiert effizient Lösungen mit wenigen redundanten Kanten ($|S| = O(V_T)$), wie sie im Allgemeinen durch die Initialisierung, Mutation und Rekombination erzeugt werden.

6.2.1 Die Suche nach redundanten Kanten

Um redundante Kanten in einer Lösung S zu finden, müssen die Cut-Komponenten der einzelnen Cut-Knoten betrachtet werden, und, wie diese untereinander verbunden sind. Anzahl und Struktur der zu vereinenden Cut-Komponenten ändern sich nach Beenden der Vorverarbeitungsschritte nicht mehr. Daher wird die Struktur als wiederverwendbare Datenstruktur implementiert, die nur einmalig a priori aufgebaut werden muss.

Für die einzelnen Cut-Komponenten eines jeden Cut-Knotens wird zunächst bestimmt, wie oft diese durch die in S enthaltenen Kanten zu anderen Cut-Komponenten verbunden sind. Die im Preprocessing zwischengespeicherte Zuordnung zwischen Erweiterungskanten und Cut-Komponenten ermöglicht dies effizient. All jene Kanten $e \in S$, die als einzige Verbindung zwischen zwei Cut-Komponenten aufscheinen, brauchen nicht weiter untersucht zu werden, da sie auf jeden Fall in der Lösung enthalten bleiben

müssen. Dieser erste Schritt der Lokalen Optimierung benötigt im schlechtesten Fall eine Rechenzeit von $O(|S| \cdot |V_T|)$. Da jedoch im Erwartungsfall $R(e) = O(\log |V_T|)$ ist, beträgt der durchschnittliche Rechenaufwand $O(|V_T| + |S| \log |V_T|)$.

Die restlichen Kanten, im Folgenden als Kantenmenge S_R bezeichnet, werden iterativ auf Redundanz überprüft. Hierzu ist eine Betrachtung der Cut-Knoten, an deren Abdecken sie beteiligt sind, notwendig. Jede Kante $e \in S_R$ wird hierzu temporär aus S entfernt. Für die Menge der von e (teilweise) abgedeckten Cut-Knoten, also $v_c \in R(e)$, wird nun überprüft, ob die zu den Cut-Knoten gehörigen Cut-Komponenten noch immer verbunden sind. Dies geschieht mittels einer Union-Find-Datenstruktur über die von e erzeugten Verbindungen unter den CutKomponenten, beziehungsweise durch einfaches Abzählen der erzeugten Verbindungen, sofern es sich um einen Cut-Knoten vom Grad kleiner als vier im BCT handelt, siehe hierzu Abschnitt 5.2. Bleiben alle Cut-Komponenten untereinander vereint, so ist e redundant und kann aus der Lösung S entfernt werden.

Der Aufwand, den diese Funktion verursacht, ist stark von der Struktur des Graphen abhängig. Der Aufwand an Rechenzeit um festzustellen, ob ein einzelner Cut-Knoten abgedeckt bleibt, wenn eine Kante e entfernt wird, beträgt maximal $O(|S|)$. Um eine Kante komplett auf Redundanz zu überprüfen, müssen $R(e)$ Cut-Knoten überprüft werden, das sind im schlechtesten Fall $O(V_C) = O(|V_T|)$. Wenn alle Kanten beitragen, alle Cut-Knoten im BCT abzudecken, und keine Kante im Vorhinein als notwendig angesehen werden kann, erfolgt diese Überprüfung für jede Kante $e \in S$. Somit beträgt der Gesamtaufwand an Rechenzeit $O(|S|^2 \cdot |V|) = O(|V|^3)$ im schlechtesten Fall. Im allgemeinen Fall liegt der Rechenaufwand weit darunter.

Eine andere Möglichkeit, um eine Erweiterungskante $e \in S$ auf Redundanz zu überprüfen, bietet Tarjans Algorithmus zum Überprüfen auf zweifachen Zusammenhang eines Graphen [47]. Dazu muss er für den ursprünglichen Graphen G_0 erweitert um $S \setminus \{e\}$, also $G_S = (V, E_0 \cup (S \setminus \{e\}))$ ausgeführt werden. Die Überprüfung einer einzelnen Kante ist auf diese Weise in Zeit $O(|V| + |S|)$ möglich. Diese Alternative wurde in Betracht gezogen und experimentell erprobt. Im Zusammenspiel mit dem MA zeigte sich, dass sie insbesondere bei umfangreichen Instanzen längere Laufzeiten für eine Optimierung aufweist als das oben dargelegte Verfahren. Die Begründung hierfür liegt zum einen darin, dass Erweiterungskanten oft zum Abdecken vieler Cut-Knoten bei-

tragen und daher $|S|$ bei lokal optimalen Lösungen im Allgemeinen wesentlich kleiner als $|V_T|$ ist. Zum anderen erzeugen die Methoden des Memetischen Algorithmus nur wenige redundante Kanten.

Betrachtet man beide Verfahren abseits des Umfeldes des Memetischen Algorithmus, so ist Tarjans Algorithmus bei steigender Anzahl von redundanten Kanten zu bevorzugen.

6.2.2 Die Suche nach fehlenden Kanten

Im vorangegangenen Abschnitt 6.2.1 wurde besprochen, wie redundante Kanten an Hand der Verbindungen zwischen den Cut-Komponenten gefunden werden können. Auf die gleiche Art und Weise kann nun auch bestimmt werden, welche Cut-Komponenten noch mit anderen vereint werden müssen, um den dazugehörigen Cut-Knoten abzudecken. Bislang unverbundene Cut-Komponenten sind mittels der gezählten Verbindungen beziehungsweise über die noch nicht vereinten Komponenten der Union-Find-Datenstruktur einfach zu finden. Eine Auswahl innerhalb der benötigten und möglichen Kanten ist über die im Preprocessing erstellten Referenzen effizient möglich.

6.2.3 Heuristiken

Ein Punkt, der durchaus Einfluss darauf hat, welches lokale Optimum für eine Lösung S gefunden wird, ist die Reihenfolge, in der man die in S enthaltenen Erweiterungskanten auf Redundanz überprüft und entfernt. Es kann sehr leicht sein, dass eine beliebige von mehreren redundanten Kanten entfernt werden kann, aber abhängig von der zuerst gewählten in der Folge nur noch die eine oder andere. Abbildung 6.2 zeigt dazu ein einfaches Beispiel.

Um die Auswahl zwischen Kanten positiv zu beeinflussen, wurde nach einer generell anwendbaren Abschätzung gesucht, welche Kanten geeigneter sind als andere, unabhängig von der aktuellen Probleminstanz. Die folgenden beiden Varianten stehen in der Implementation zur Verfügung.

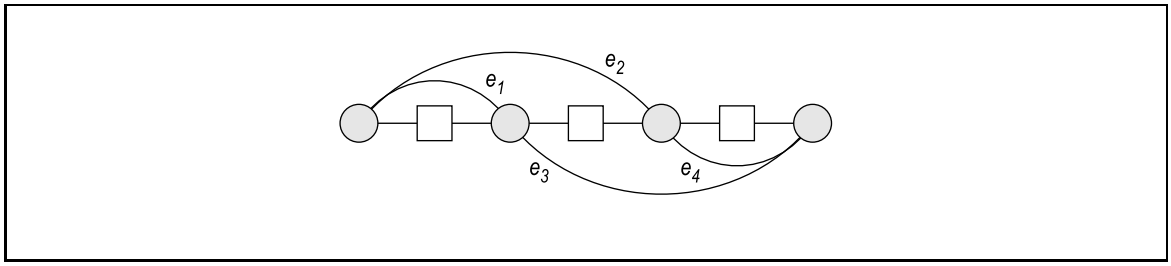


Abbildung 6.2: Beispiel für abhängige Kanten. Die Abbildung zeigt einen einfachen Graphen mit vier Erweiterungskanten, von denen zwei redundant sind. Eine jede dieser vier Kanten kann entfernt werden, allerdings schränkt die erste Wahl die weiteren Möglichkeiten ein. Wird zum Beispiel Kante e_2 zuerst entfernt, kann anschließend nur noch e_4 als zweite Kante entfernt werden.

- *Kostenheuristik:* Es ist recht unwahrscheinlich, dass die Wahl der teuersten Kanten zu einer optimalen Lösung führen wird. Aus diesem Grund werden günstigere Kanten als geeigneter betrachtet als teurere.

- *Kosten-pro-Cut-Knoten-Heuristik:* Eine Kante, die zwei entfernt liegende Knoten miteinander verbindet, wird meist teurer sein, als eine, deren Endpunkte nahe beieinander liegen. Allerdings trägt die längere Kante aber auch zum Abdecken einer größeren Anzahl von Cut-Knoten bei. Deshalb werden hier die relativen Kosten der Erweiterungskanten ermittelt, indem die jeweils tatsächlichen Kosten durch die Anzahl der Cut-Knoten, bei denen die Kante zum Abdecken beiträgt, dividiert werden. Im Folgenden wird diese Heuristik auch einfach als *Cut-Knotenheuristik* bezeichnet.

Die Anwendung dieser beiden Heuristiken beschränkt sich nicht nur auf die lokale Optimierung, sondern findet auch in anderen Methoden des Algorithmus Verwendung. Die Auswahl der Heuristik gilt jeweils global. Eine Unterscheidung je nach Methode ist nicht vorgesehen.

6.2.4 Die implementierten Strategien

Tests von links nach rechts. Die Lokale Optimierung ist einer der am häufigsten ausgeführten Programmabschnitte des Algorithmus. In Hinblick auf eine möglichst kurze Laufzeit wurde diese Variante inkludiert. Die Kanten $e \in S$ werden in der Reihenfolge überprüft, in der sie in der Lösungsmenge aufscheinen. Zu beachten ist, dass Mengen-Darstellungen in der in der Implementation verwendeten Standard-Template-Library (STL) intern eine fixe Anordnung der Elemente haben. Dieser Effekt ist zwar nicht wünschenswert, jedoch sollte es den Testläufen überlassen werden zu zeigen, ob und wie stark sich dies beim Finden einer möglichst optimalen Lösung auswirkt.

Tests in zufälliger Reihenfolge. Um ungewünschten Nebeneffekten, bedingt durch eine fixe Reihung der Kanten abhängig von der internen Speicherung der Lösungsdatenstruktur, entgegenzuwirken, werden hier die Kanten explizit zufällig gewählt und getestet.

Tests in umgekehrter Reihenfolge der Kosten. In dieser Variante werden zuerst teure Kanten aus der zu überprüfenden Lösungsmenge entfernt. Abhängig von der gewählten Heuristik werden die Kanten in umgekehrter Reihenfolge ihrer Kosten, beziehungsweise Kosten pro Cut-Knoten, auf Redundanz überprüft und gegebenenfalls entfernt.

6.3 Initialisierung der Startpopulation

Eine gute Startpopulation kann sowohl die Laufzeit als auch die Qualität der Lösung eines Memetischen Algorithmus positiv beeinflussen. Wünschenswert wäre zum einen eine breite Streuung der Individuen im Suchraum (Diversität), zum anderen ein schon möglichst hoher Qualitätsdurchschnitt, um dem Algorithmus die Suche zu erleichtern.

Zur Initialisierung wurden zwei Gruppen von Varianten implementiert. Die erste Gruppe basiert auf der iterativen Auswahl von Kanten für noch nicht vereinte Cut-Komponenten und deckt somit Cut-Knoten Schritt für Schritt ab.

Die zweite Gruppe wählt jeweils aus der Gesamtheit der Kantenmenge E_A. Da diese Wahl nicht so zielgerichtet wie jene der vorher beschriebenen Gruppe erfolgen kann, ist die Laufzeit natürlich höher. Allerdings war es im Voraus nicht abzusehen, inwieweit die globale Sicht dieser Methoden zum Finden einer besseren Lösung beziehungsweise zur Initialisierung einer geeigneteren Population beitragen würde.

6.3.1 Varianten ausgehend von noch nicht verbundenen Cut-Komponenten

Diese Varianten basieren auf dem schrittweisen Abdecken von Cut-Knoten, indem sie für noch nicht vereinte Cut-Komponenten Erweiterungskanten wählen, die diese mit anderen verbinden. Die in Abschnitt 6.2.1 beschriebene Methode wird verwendet um zu bestimmen, für welche Cut-Komponenten noch Erweiterungskanten selektiert werden müssen. Die Auswahl der Kanten erfolgt iterativ. Mit jeder Kante, die einem Individuum hinzugefügt wird, werden gleichzeitig die Datenstrukturen der lokalen Optimierung schrittweise angepasst, wodurch die weitere Auswahl zielgerichtet bleibt und ein Auftreten von redundanten Kanten soweit als möglich verhindert wird. Sobald alle Cut-Knoten abgedeckt sind, schließt ein Aufruf der Optimierungsmethode die Initialisierung des Individuums ab.

Initialisiere von links nach rechts und wähle zufällig. In dieser einfachsten und in Bezug auf die Laufzeit günstigsten Version werden alle Cut-Knoten und ihre Cut-Komponenten der Reihe nach bearbeitet. Ist eine Cut-Komponente noch nicht mit anderen vereint, so wird aus der Menge der möglichen Verbindungen eine Erweiterungskante zufällig gleichverteilt gewählt und dem Individuum hinzugefügt. Dieser Vorgang wird wiederholt, bis alle Cut-Knoten abgedeckt sind. Abschließend entfernt die lokale Optimierung möglicherweise vorhandene redundante Kanten, und das neu generierte Individuum wird der Population hinzugefügt.

Initialisiere in zufälliger Reihenfolge und wähle zufällig. Durch eine fixe Anordnung der abzudeckenden Cut-Knoten und ihrer Cut-Komponenten kann es selbst

bei zufälliger Auswahl der Kanten zu unerwünschten Effekten kommen. Man stelle sich hierzu vor, der erste bearbeitete Cut-Knoten habe unter anderem eine Kante zur Verbindung zweier seiner Cut-Komponenten zur Auswahl, die einen sehr langen Pfad im BCT abdeckt, dadurch beiträgt, viele Cut-Komponenten verschiedener Cut-Knoten zu vereinen, aber unverhältnismäßig hohe Kosten verursacht und somit nicht Teil einer optimalen Lösung ist. Diese Kante würde nun mit gleicher Häufigkeit in allen Individuen auftreten, und durch ihre Selektion zu Beginn des Aufbaus des Individuums würde die Wahl von günstigeren Kantenkombinationen unterbunden werden. Um solchen in lichten Graphen mit erhöhter Wahrscheinlichkeit auftretenden Situationen vorzubeugen, wird in dieser Variante die Reihenfolge, in der Cut-Komponenten bearbeitet werden, explizit zufällig und unabhängig von den ihnen zugeordneten Cut-Knoten gewählt. Die weitere Vorgangsweise ist ident mit der zuvor vorgestellten Variante.

Heuristisch gesteuerte Zufallsauswahl mittels Normalverteilung. Für diese Art der Initialisierung werden die in der Vorverarbeitung erstellten Referenzen zwischen Cut-Komponenten und Erweiterungskanten vorweg einmalig entsprechend der gewählten Heuristik aufsteigend sortiert. Wie zuvor werden die Cut-Komponenten in zufälliger Reihenfolge bearbeitet und für noch nicht verbundene Komponenten wird eine Kante gewählt, die diese mit anderen vereint. Ziel dieser Variante ist es, bevorzugt günstige Kanten im Sinne der gewählten Heuristik zu selektieren. Hierzu wird eine normalverteilte Zufallszahl ermittelt, die dem Rang der zu wählenden Kante in der aufsteigend sortierten Liste entspricht. Die verwendete Formel stellt eine Abwandlung der von G. Raidl in [39] angewandten dar und lautet:

$$Rang = \left\lfloor |\mathcal{N}(0,s)| \cdot |E'_A| \right\rfloor \mod |E'_A| \tag{6.1}$$

wobei $\mathcal{N}(0,s)$ eine normalverteilte Zufallszahl mit Mittelwert 0 und Standardabweichung s generiert, und $E'_A \in E_A$ die Menge der in Frage kommenden Erweiterungskanten angibt. Die Nummerierung beginnt wie in C++ üblich mit dem Rang Null. Abhängig vom gewählten Parameter s erhält man eine verschieden starke Favorisierung der kostengünstigen Erweiterungskanten. Abbildung 6.3 auf Seite 60 illustriert den Einfluss des Strategieparameters s.

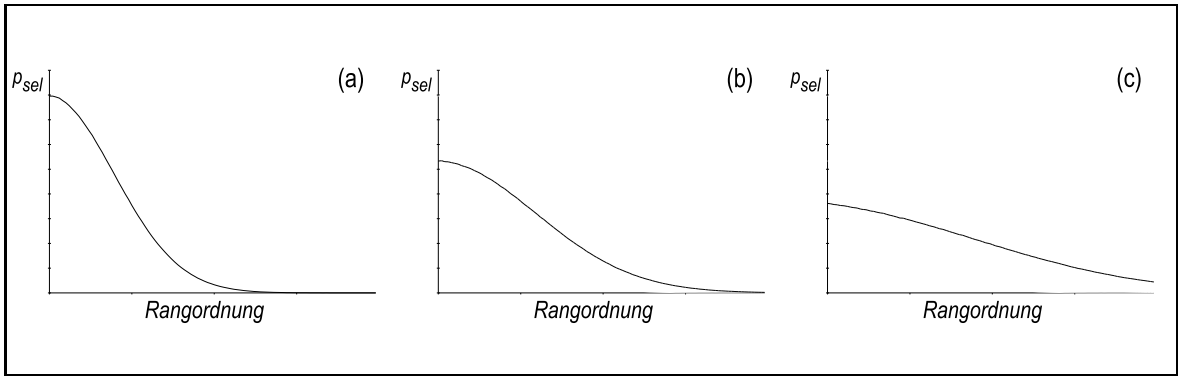


Abbildung 6.3: Die Bildfolge zeigt den Einfluss des Strategieparameters s auf die Wahrscheinlichkeit p_{sel}, mit der eine Kante gewählt wird. Je niedriger s gewählt wird, desto stärker werden kostengünstige Kanten bevorzugt. Die Kurven zeigen die Verteilung für $s = 0.2$, $s = 0.3$ und $s = 0.5$.

Invers proportionale Zufallsauswahl. Eine rangbasierte Auswahl birgt gewisse Fallstricke in sich. Da es nicht erstrebenswert ist, den Parameter für die Standardabweichung für jede Testinstanz neu zu optimieren, kann ein allgemein gut funktionierender Wert für Beispiele mit Kanten mit nur sehr wenig variierenden Kosten ungeeignet sein, da er die geringfügig günstigeren unverhältnismäßig bevorzugt. Eine robustere Möglichkeit zur Bevorzugung der kostengünstigeren Kanten schaffen diese und die folgende Methode. Auch hier werden die Cut-Komponenten in zufälliger Reihenfolge betrachtet. Für noch nicht vereinte Komponenten werden zwei Kanten zufällig gleichverteilt gewählt, die diese mit anderen verbinden. Zwischen den beiden Erweiterungskanten wird mit invers proportionaler Wahrscheinlichkeit zu ihren Kosten beziehungsweise Kosten pro Cut-Knoten entschieden. Der Vorgang wird wiederholt, bis das neue Individuum eine gültige Lösung darstellt.

Binäre Turniere. Um den Druck im Vergleich zur vorhergehenden Methode bei der Auswahl der Kanten zu erhöhen und somit günstigere Kanten verstärkt zu wählen, wurde diese Variante implementiert. Für jede noch nicht verbundene Cut-Komponente werden in zufälliger Reihenfolge zwei Kanten zufällig gleichverteilt gewählt. Die heuristisch günstigere Kante wird dem Individuum hinzugefügt. Die Auswahl erfolgt mit Zurücklegen, sodass auch die ungünstigste Kante gewählt werden kann.

6.3.2 Varianten, die die gesamte Menge der Erweiterungskanten betrachten

Diese Varianten wählen Erweiterungskanten jeweils aus der gesamten Menge E_A. Hierdurch ist es möglich, generell günstige Kanten zu favorisieren, unabhängig davon, welche Cut-Komponenten durch sie vereint werden. Da die Auswahl nicht so zielgerichtet erfolgen kann wie in den zuvor besprochenen Methoden in Abschnitt 6.3.1, muss für diesen Vorteil eine erhöhte Laufzeit in Kauf genommen werden.

Normalverteilte Wahl über alle Kanten. Für diese Variante werden alle Kanten der Menge E_A zuvor einmalig aufsteigend sortiert. Die Reihung der Kanten wird entsprechend der gewählten Heuristik durchgeführt, sowohl die Kosten- als auch Cut-Knotenheuristik sind möglich. Die Auswahl der Kanten erfolgt über eine normalverteilte Zufallszahl, die den Rang der zu wählenden Kante angibt. Ist die für die Erweiterung der bisher generierten Teillösung selektierte Kante ungeeignet, das heißt sie vereint nicht zumindest zwei bislang unverbundene Cut-Komponenten miteinander, so werden als Ersatz Kanten mit heuristisch ähnlichem Wert probiert. Das sind nun ihre Nachbarn im Sinne der Rangordnung, also jene Kanten, deren Rang um ± 1 vom zuvor gewählten abweicht. Sind auch diese ungeeignet, werden die um jeweils einen Schritt weiter entfernten Nachbarn getestet, bis eine passende Kante gefunden wird. Nachdem dem Individuum eine passende Erweiterungskante hinzugefügt wurde wiederholt sich der Vorgang, beginnend mit einer neu generierten Zufallszahl, bis eine gültige Lösung erreicht wird.

Zur Wahl des Ranges stehen zwei verschiedene Funktionen zur Verfügung. Die erste von der Normalverteilung abgeleitete Funktion erstreckt sich gleichförmig über den gesamten Bereich der Erweiterungskanten E_A.

$$Rang = \left\lfloor \mathcal{N}(0, s) \cdot |E_A| \right\rfloor \mod |E_A| \tag{6.2}$$

wobei $\mathcal{N}(0, s)$ eine normalverteilte Zufallszahl mit Mittelwert 0 und Standardabweichung s generiert.

Als zweite Variante kommt eine auf der Normalverteilung basierende Funktion zum Einsatz, die zusätzlich noch über dem kostengünstigeren Teil verdichtet ist und sich

bereits in [39] bewährt hat. Um den bevorzugten Bereich der Kanten weiter einzuengen, wird die Mächtigkeit der Knotenmenge V als heuristisches Maß verwendet.

$$Rang = \left\lfloor \mathcal{N}(0, s) \cdot |V| \right\rfloor \mod |E_A| \tag{6.3}$$

wobei $\mathcal{N}(0, s)$ wiederum eine normalverteilte Zufallszahl mit Mittelwert 0 und einer Standardabweichung s wählt.

Sobald das neu generierte Individuum einer gültigen Lösung entspricht, wird noch die lokale Optimierung durchgeführt und das Individuum in die Population eingegliedert. Zur Veranschaulichung folgt eine Gegenüberstellung beider Funktionen auf Seite 63.

Normalverteilte Wahl und aktueller Nutzen. Diese Initialisierungsvariante ist sehr aufwändig und wurde ursprünglich in deterministischer Form angewandt, um erste Vergleichswerte für den MA im frühen Stadium des Projektes zu generieren, als Referenzwerte von anderen Verfahren noch nicht beziehungsweise nur für wenig umfangreiche Testinstanzen vorhanden waren. Da die Ergebnisse trotz der Einfachheit des Algorithmus überraschend gut waren, wurde sie für die Initialisierung abgewandelt um zu evaluieren, ob der beträchtliche Mehraufwand an Laufzeit die Population dermaßen verbessert, dass er gerechtfertigt ist.

Das Verfahren beruht auf einer Abschätzung des Nutzens der Kanten, die verwendet werden können, um eine noch unvollständige Teillösung zu erweitern. Hierzu dient eine Maßzahl, die sich aus der Anzahl der durch e neu verbundenen Cut-Komponenten dividiert durch die Kosten der Kante e ergibt. Somit gilt

$$Nutzen(e) = \frac{CC(S) - CC(S \cup e)}{Kosten(e)} \tag{6.4}$$

wobei $CC()$ die Anzahl der Cut-Komponenten des BCTs unter Berücksichtigung der Erweiterungskanten in S angibt. Alle Kanten $e \in E_A$ werden an Hand ihres aktuellen Nutzens absteigend sortiert, wobei Kanten ohne Nutzen vernachlässigt werden. Die erste Sortierung wird zwischengespeichert. Nun wird mittels einer normalverteilten

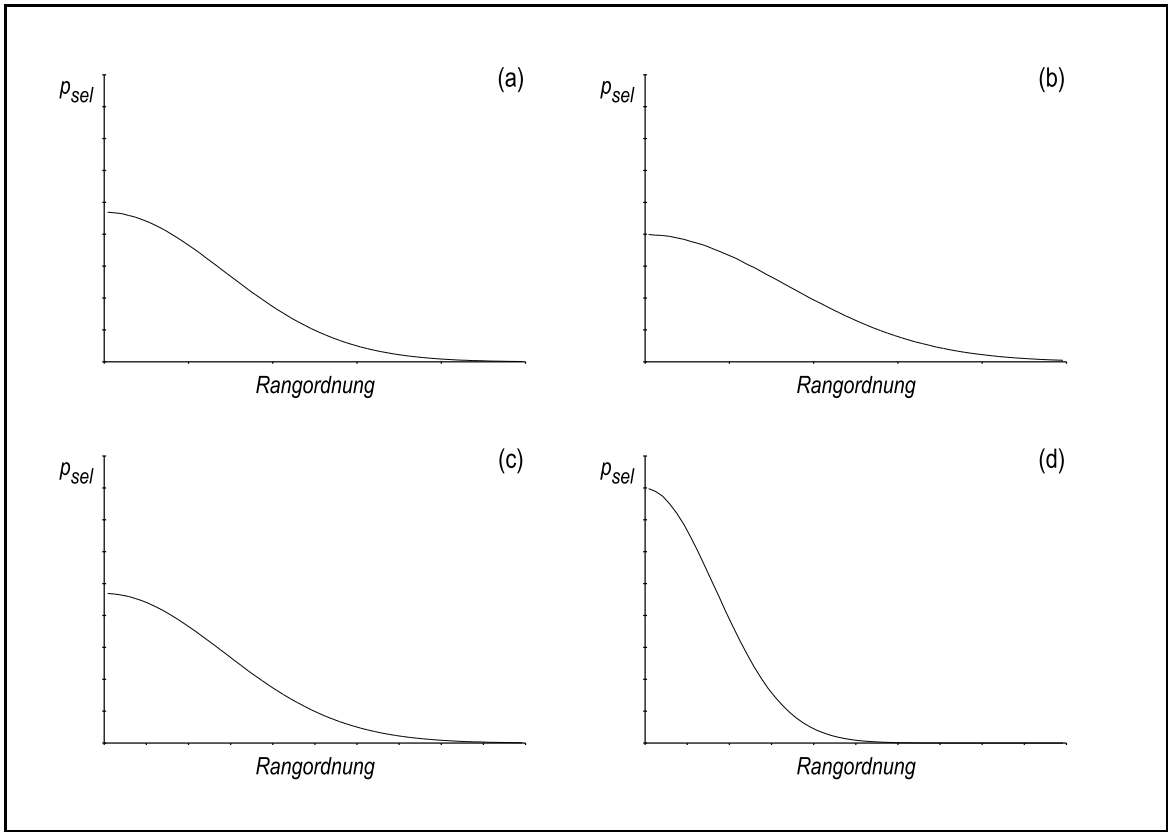


Abbildung 6.4: Gegenüberstellung der beiden Normalverteilungen. Die X-Achse entspricht der Rangordnung der Erweiterungskanten, die Y-Achse gibt die Wahrscheinlichkeit p_{sel} an, mit der eine Kante gewählt wird. In (a) ist eine Verteilung unter Verwendung von Formel 6.2 zu sehen. Rechts daneben in (b) wird zur Auswahl Formel 6.3 verwendet. Wird nun die Kantenanzahl bei gleichbleibender Knotenanzahl erhöht, treten die unterschiedlichen Eigenschaften deutlich zu Tage. Die Auswahl mit Formel 6.2 dehnt sich gleichmäßig über die vergrößerte Kantenmenge, die Kurve in (c) ist somit gleichgeblieben. Formel 6.3 fokussiert den abgeschätzten Bereich, die Kurve staucht sich somit über dem kostengünstigen Teil.

Zufallszahl der Rang der zu wählenden Kante ermittelt. Die verwendete Formel ist ident mit Formel 6.1 auf Seite 59.

$$Rang = \left\lfloor |\mathcal{N}(0,s)| \cdot |E'_A| \right\rfloor \mod |E'_A|$$

Dabei wählt $\mathcal{N}(0,s)$ eine normalverteilte Zufallszahl mit Mittelwert 0 und Standardabweichung s und $E'_A \in E_A$ stellt die Menge der geeigneten Erweiterungskanten dar. Nachdem die gewählte Kante zur Lösung hinzugefügt wurde, wird der aktuelle Nutzen für die verbliebenen Kanten aktualisiert. Der Vorgang wird wiederholt, bis das Individuum eine gültige Lösung repräsentiert. Bevor das neu generierte Individuum zur Population hinzugefügt wird, wird noch die lokale Optimierungsfunktion angewandt.

6.4 Rekombination

Die Rekombination ist einer der wichtigsten Teile für einen Memetischen Algorithmus. Durch sie wird primär die Weitergabe von Information innerhalb der Population auf nachfolgende Generationen bestimmt. Für diesen Algorithmus wurde ein 2-Merger-Ansatz gewählt, also zwei Elternteile S_1 und S_2 werden zur Erzeugung eines Nachkommen S_c verwendet. Als Ziel wurde für diesen Operator gesetzt, möglichst viele Merkmale der Elternindividuen an ihren Nachkommen weiterzugeben, also die Kanten der beiden Elternteile zu benutzen um eine neue Lösung zu generieren, ohne dabei auf Erweiterungskanten zurückzugreifen, die nicht in S_1 oder S_2 enthalten sind. Da S_1 und S_2 bereits gültige Lösungen darstellen, ist dies immer möglich. Ein neu generierter Nachkomme wird zur Population P hinzugefügt, sofern er nicht mit einem bereits vorhandenen Individuum ident ist. Die Überprüfung auf Übereinstimmung erfolgt durch hashing und ist somit effizient durchführbar.

Mehrere Möglichkeiten der Rekombination wurden implementiert, um in den später folgenden Testläufen die geeignetste zu bestimmen.

Vereinige und reduziere. Diese Variante stützt sich beim Generieren eines Nachkommen auf die lokale Optimierung. Zunächst werden alle Kanten der beiden Eltern-

individuen auf den Nachkommen übertragen, somit also $S_c = S_1 \cup S_2$. Der Rechenaufwand hierfür beträgt $O(|S_1| + |S_2|)$. Die lokale Optimierung reduziert danach das neue Individuum so weit als möglich. Abbildung 6.5 auf Seite 66 zeigt hierzu ein Beispiel.

Schnittmenge und invers proportionale Wahl. Unabhängig von der weiteren Vorgehensweise sollen bei dieser Variante Kanten, die einen überdurchschnittlichen Nutzen aufweisen und somit in beiden Elternteilen bereits vorhanden sind, übernommen werden. Hierzu wird zunächst die Schnittmenge der beiden Elternindividuen auf den Nachkommen übertragen, $S_c = S_1 \cap S_2$, und gleichzeitig die symmetrische Differenz S_D der beiden Mengen gebildet, $S_D = (S_1 \cup S_2) \setminus (S_1 \cap S_2)$. Aus S_D werden wiederholt jeweils zwei Kanten zufällig gewählt. Zwischen diesen beiden wird dann mit invers proportionaler Wahrscheinlichkeit abhängig von der gewählten Heuristik entschieden. Sowohl die Kosten- als auch die Cut-Knotenheuristik werden unterstützt. Die selektierte Kante wird nun zur Erweiterung der Lösung S_c verwendet. Werden durch sie keine bislang unverbundenen Cut-Komponenten vereint, wird sie verworfen. Die andere verbleibt in der Menge der möglichen Erweiterungskanten S_D und kann somit erneut gewählt werden. Es werden so lange weitere Kanten gezogen, bis S_c eine gültige Lösung darstellt. Der Rechenaufwand für die Rekombination beträgt bedingt durch den zusätzlichen Test $O((|S_1| + |S_2|) \cdot |V_T|)$ im schlechtesten und $O((|S_1| + |S_2|) \cdot \log |V_T|)$ im durchschnittlichen Fall. Dies amortisiert sich jedoch in der nachfolgenden lokalen Optimierung, da es effizienter ist zu überprüfen, ob eine Kante notwendig, also nicht redundant ist, als ihre Redundanz nachzuweisen. Vergleiche hierzu 6.2.1. Zum Schluss wird der neu generierte Nachkomme noch lokal optimiert, bevor er der Population hinzugefügt wird. In Abbildung 6.6 auf Seite 67 wird die Vorgangsweise grafisch veranschaulicht.

Schnittmenge und binäre Turniere. Auch hier wird zunächst die Schnittmenge der Kanten beider Elternteile direkt auf den Nachkommen S_c vererbt und danach aus der symmetrischen Differenz S_D ergänzt. Die weitere Auswahl erfolgt durch binäre Turniere zwischen Kanten der Menge S_D, um die Wahl von günstigen Erweiterungskanten im Vergleich zur zuvor beschriebenen Methode weiter zu verstärken. Die Turniere erfolgen mit Zurücksetzen, sodass jede Kante zuerst gewählt werden kann.

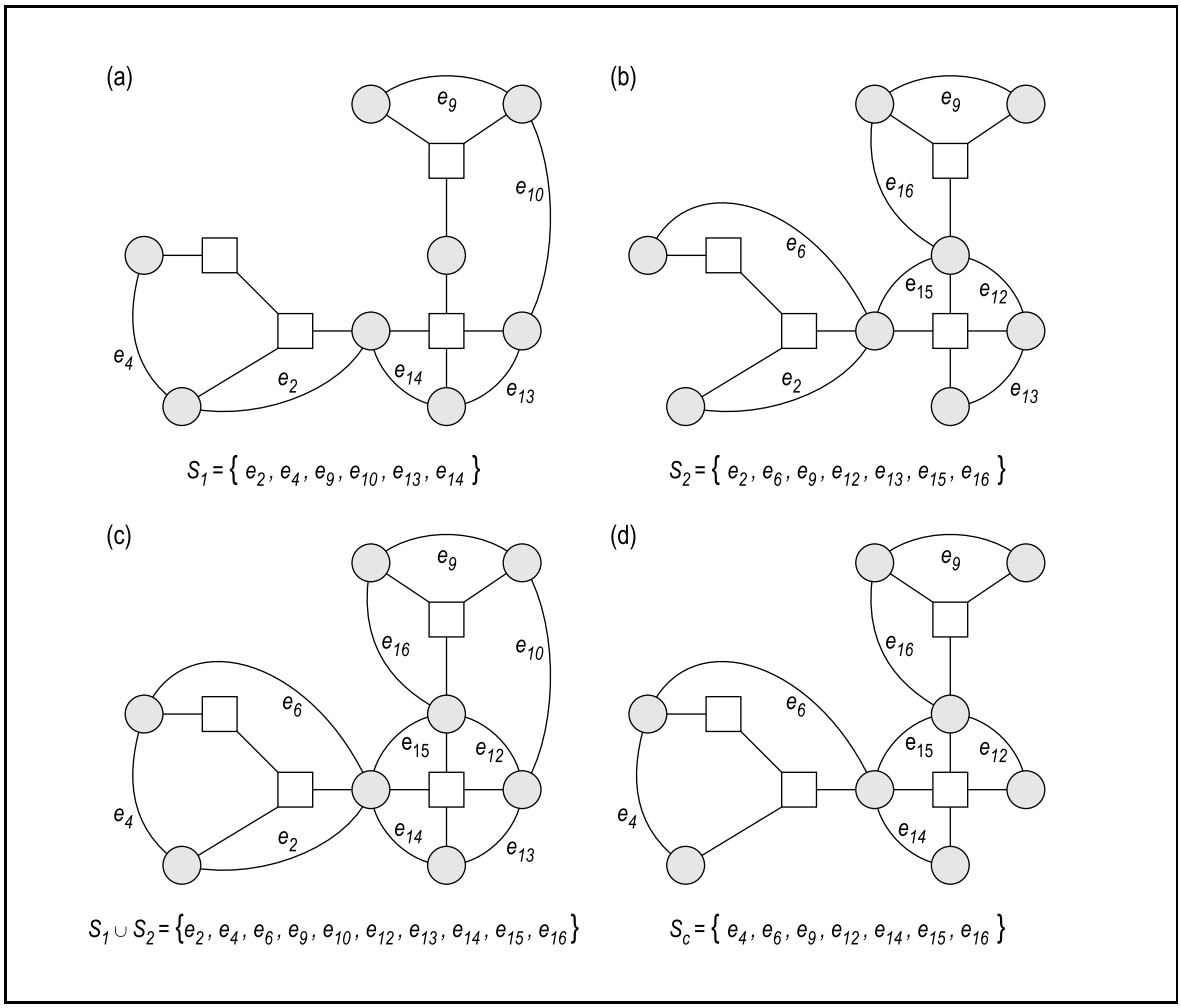


Abbildung 6.5: Beispiel zur Rekombinations-Variante „vereinige und reduziere“. In (a) und (b) werden zwei für die Rekombination gewählte Individuen S_1 und S_2 eines kleinen Beispielgraphen dargestellt, die von ihnen repräsentierte Lösung ist jeweils im BCT darüber veranschaulicht. Quadrate stellen Cut-Knoten dar, grau unterlegte Kreise Block-Knoten. Erweiterungskanten sind mit e_i beschriftet. Beide Elternteile übertragen ihre Kanten nun auf den Nachkommen S_c, dargestellt in (c), danach reduziert die lokale Optimierung das neu erstellte Individuum. (d) zeigt ein mögliches Resultat.

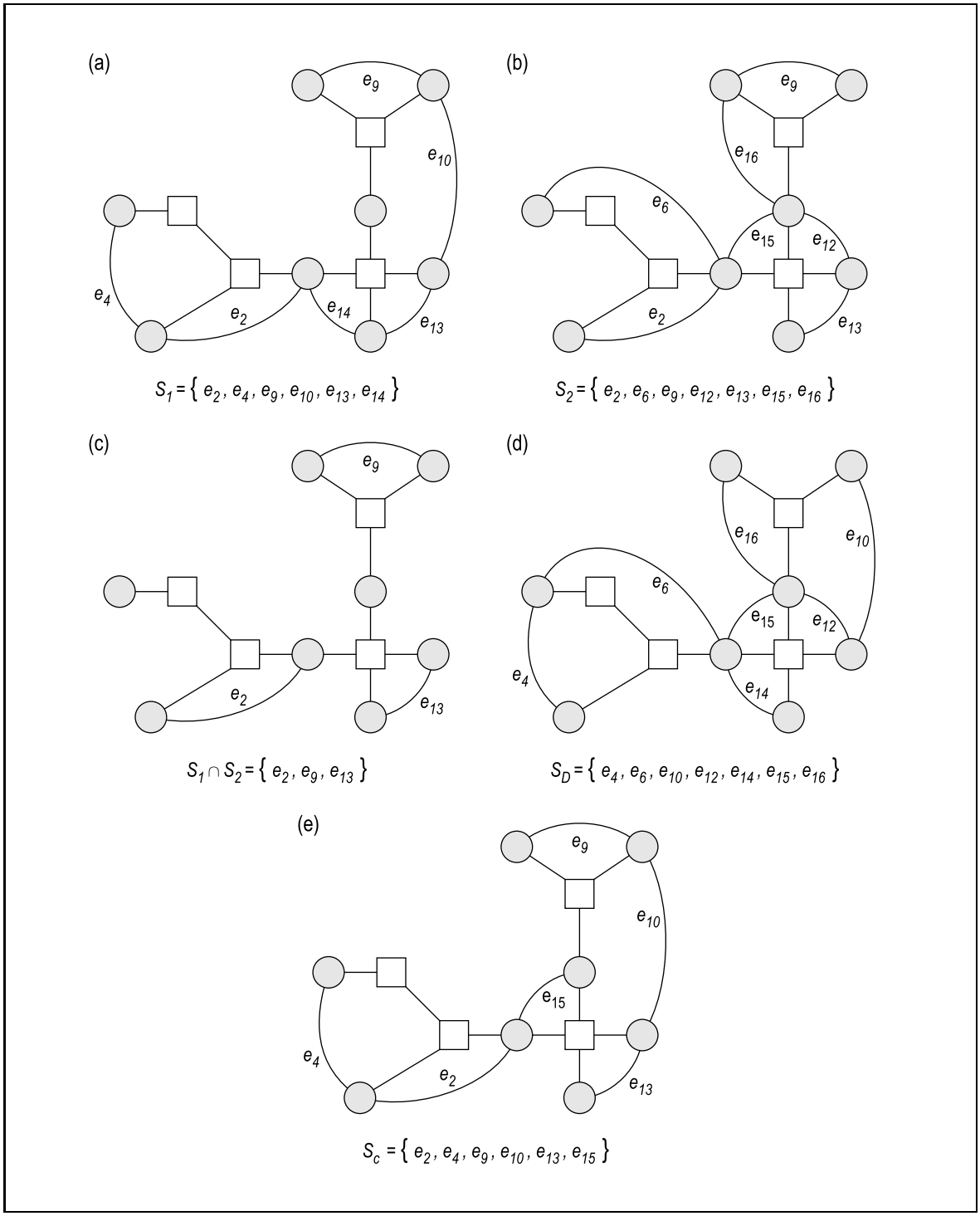


Abbildung 6.6: Beispiel zur Rekombination mittels Schnittmenge und symmetrischer Differenz.

Beschreibung zu Abbildung 6.6: Die Abbildung zeigt den generellen Ablauf der Rekombination von zwei Individuen S_1 und S_2 unter Verwendung der Schnittmenge und der symmetrischen Differenz beider Elternindividuen. Die Bilder a) und b) zeigen die beiden Elternindividuen, die von ihnen repräsentierte Lösung ist jeweils im BCT darüber veranschaulicht. Quadrate stellen Cut-Knoten dar, grau unterlegte Kreise Block-Knoten. Erweiterungskanten sind mit e_i beschriftet und als gekrümmte Linien eingezeichnet. Als erster Schritt wird die Schnittmenge der in S_1 und S_2 enthaltenen Kanten gebildet und auf den Nachkommen S_c vererbt, dargestellt in Bild c). Gleichzeitig wird die symmetrische Differenz S_D ermittelt, aus der anschließend weitere Kanten für S_c gewählt werden (Abbildung d). Nach mehrmaligem Hinzufügen von Kanten aus S_D ist der Nachkomme fertiggestellt, ein mögliches Ergebnis ist in Bild e) zu sehen.

6.5 Mutation

Der Mutationsoperator wird mit einer zuvor festgelegten Wahrscheinlichkeit auf einen neu generierten Nachkommen angewandt. Es wird eine lokal begrenzte Veränderung an dem Individuum vorgenommen, was hier das Löschen einer Kante bedeutet. Diese wird anschließend durch eine oder mehrere Kanten ersetzt, womit neue, nicht in den Elternindividuen enthaltene Kanten dem Nachkommen, und somit auch der Population, hinzugefügt werden können. Für alle vorgestellten Varianten gilt, dass zum Ersetzen der gelöschten Kante auch dieselbe wieder gewählt werden kann. Die Vorgangsweise erfolgt in zwei Teilschritten. Im ersten Schritt wird eine Kante bestimmt, die danach gelöscht wird. Im zweiten Schritt erfolgt nun die Wahl jener Kante oder Kantenkombination, durch die das Individuum zu einer gültigen Lösung ergänzt wird. Als obere Schranke für den Rechenaufwand der Mutation gilt $O(|V_T| \cdot |E_A|)$, jedoch liegt der in der Praxis verursachte Zeitaufwand weit unter dem der lokalen Optimierung.

6.5.1 Wähle eine Kante, die gelöscht werden soll

Wähle eine zufällige Kante. Eine Kante des Individuums wird zufällig gleichverteilt gewählt und gelöscht. Durch die rein zufällige Auswahl ist sichergestellt, dass in

jedem Bereich des Suchraums Alternativen inkludiert werden können und auch für Kanten Ersatz gesucht wird, die nur scheinbar von großem Nutzen sind.

Wähle zwei Kanten und entscheide proportional. Um die Mutation verstärkt auf Bereiche zu lenken, in denen eine Verbesserung wahrscheinlicher ist als in anderen, werden in dieser Variante zwei Erweiterungskanten des Individuums gleichverteilt gewählt. Abhängig von der gewählten Heuristik wird nun mit einer Wahrscheinlichkeit proportional zu den Kosten der Kanten oder den Kosten pro Cut-Knoten zwischen diesen beiden entschieden.

Binäre Turniere. Wie in der zuvor beschriebenen Variante werden auch hier zwei Kanten zufällig aus einem Individuum gewählt. Um die Mutation noch stärker auf teure Kanten zu lenken, wird durch ein Turnier entschieden, welche Kante gelöscht wird.

6.5.2 Wähle Ersatz für die gelöschte Kante

In Abschnitt 6.2.2 wurde bereits gezeigt, wie noch nicht vereinte Cut-Komponenten gefunden werden können. Analog zu den Initialisierungsvarianten aus Abschnitt 6.3.1 wird auch hier nach einer Menge von Kanten gesucht, die die noch nicht verbundenen Cut-Komponenten vereinen. Der Unterschied besteht lediglich darin, dass bereits ein Großteil der Lösung vorhanden ist und nur noch eine oder wenige Kanten hinzugefügt werden müssen, um wieder eine gültige Lösung zu erhalten.

Zunächst gilt es die noch verbliebenen Kanten des Individuums S zu berücksichtigen. Alle Kanten $e \in S$ werden hierzu der Reihe nach bearbeitet und die durch sie verbundenen Cut-Komponenten vereint. Cut-Knoten, die nur teilweise abgedeckt sind, scheinen nun durch ihre nicht verbundenen Cut-Komponenten auf. Diese Cut-Komponenten werden in zufälliger Reihenfolge unabhängig von ihrer Zuordnung zu den Cut-Knoten bearbeitet. Die implementierten Methoden nutzen die im Preprocessing erstellten Referenzen, um auf direktem Weg Erweiterungskanten zu wählen, die eine Cut-Komponente mit einer anderen vereinen. Durch diese Vorgehensweise wird es ermöglicht, dass eine

gelöschte Kante entweder durch eine andere Erweiterungskante, oder durch eine Kantenkombination, die dieselben Vereinigungen erzeugt, ersetzt wird. Die implementierten Varianten unterscheiden sich in der Art und Weise, wie zwischen den zur Auswahl stehenden Erweiterungskanten gewählt wird.

- *Wähle zufällig:* Es wird mit gleicher Wahrscheinlichkeit unter den Kandidaten gewählt. Im Zusammenspiel mit dem Löschen einer ebenfalls rein zufällig gewählten Kante ergibt dies die Strategie einer klassischen Mutation. Der Vorteil dieser Vorgangsweise ist ein positiver Effekt auf den Erhalt des Variantenreichtums innerhalb der Population.

- *Wähle normalverteilt:* Für die Wahl der Kandidaten wird eine rangbasierte normalverteilte Zufallsauswahl getroffen. Günstigere Kanten im Sinne der gewählten Heuristik erhalten dadurch eine höhere Chance inkludiert zu werden. Die Auswahl der Kante erfolgt nach der bereits vertrauten Formel 6.1:

  $$Rang = \left\lfloor |\mathcal{N}(0,s)| \cdot |E'_A| \right\rfloor \mod |E'_A|$$

  wobei $\mathcal{N}(0,s)$ eine normalverteilte Zufallszahl mit Mittelwert 0 und Standardabweichung s generiert und $E'_A \in E_A$ die Menge der geeigneten Erweiterungskanten darstellt. Für diese Variante ist es notwendig, die im Preprocessing erzeugten Referenzen vorab einmalig abhängig von der gewählten Heuristik zu sortieren.

Wurde eine Kante gewählt, so werden die durch sie verbundenen Cut-Komponenten vereint. Falls erforderlich, werden nun weitere Kanten nach demselben Schema selektiert. Auf das mutierte Individuum wird zum Schluss noch die lokale Optimierungsfunktion angewandt. Abbildung 6.7 auf Seite 71 veranschaulicht den Vorgang.

6.6 Zusammenfassung

In diesem Kapitel wurden die verschiedenen Operatoren und ihre implementierten Variationen besprochen. Individuen werden direkt als Menge von Kanten codiert, wodurch eine effiziente Verwaltung und Manipulation möglich wird. Für die Initialisierung der

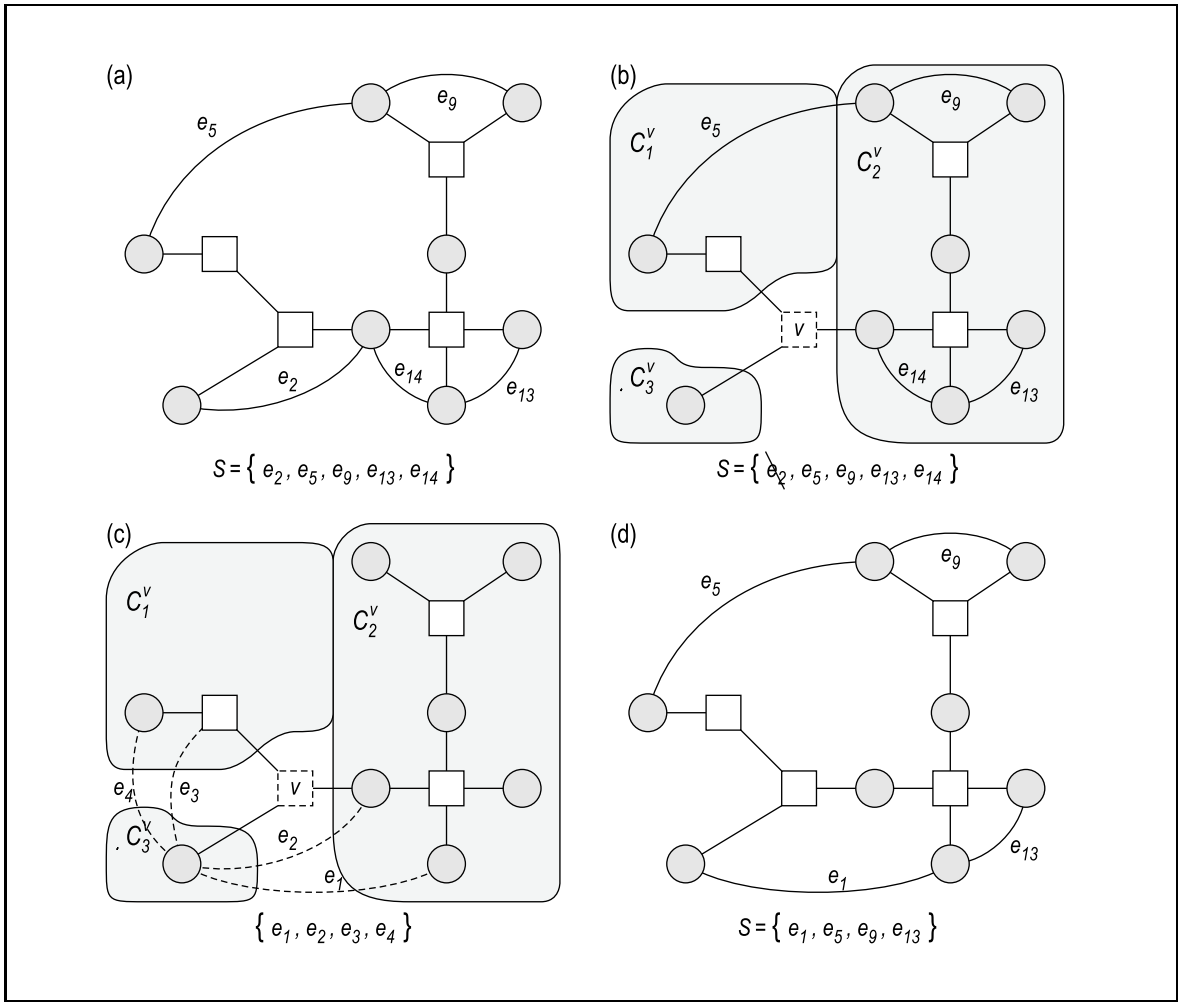


Abbildung 6.7: Beispiel zur generellen Funktionsweise der Mutation. Nachdem ein Individuum zur Mutation selektiert wurde, wird eine Kante der von ihm repräsentierten Lösung gelöscht. Bei dem in Bild a) dargestellten Individuum S wird die Kante e_2 entfernt. Hierdurch ist zumindest ein Cut-Knoten nicht mehr abgedeckt. Abbildung b) zeigt das Individuum ohne die Kante e_2, die Cut-Komponenten $C_1^v \ldots C_3^v$ des Cut-Knotens v sind nur noch teilweise untereinander verbunden. Um wieder eine gültige Lösung zu erhalten, muss eine Erweiterungskante gewählt werden, die die Cut-Komponente C_3^v mit den noch vereinten Komponenten C_1^v und C_2^v verbindet. Bild c) zeigt nur die Menge der möglichen Kanten, aus denen eine gewählt und dem Individuum hinzugefügt wird. In Bild d) wurde S um die Kante e_1 erweitert, und die nun redundante Kante e_{14} entfernt.

Individuen stehen zwei Gruppen von Methoden zur Verfügung. Die eine Gruppe wählt direkt Verbindungen für noch nicht vereinte Cut-Komponenten, die andere betrachtet die gesamte Menge der Erweiterungskanten und wählt global, bis eine gültige Lösung gefunden ist. Die Rekombination ist als 2-Merger realisiert. Sie kombiniert zwei Elterninidividuen je nach Variante unterschiedlich kostenorientiert zu einem Nachkommen. Der Mutationsoperator bewirkt punktuelle Veränderungen an einem Individuum. Er wird mit einer vorab gewählten Wahrscheinlichkeit auf neu generierte Nachkommen angewandt und ersetzt eine Kante durch eine andere oder eine äquivalente Kantenkombination. Je nach gewählter Methode fungiert er als klassische Mutation oder unterstützt den Algorithmus bei seiner Suche. Die lokale Optimierung wird nach jedem Operator, der eine Veränderung an einem Individuum durchführt, angewandt und basiert auf der Analyse der Cut-Knoten, deren zugeordneten Cut-Komponenten und ihrer Vereinigung durch Erweiterungskanten. Sie stellt sicher, dass nur gültige und lokal optimale Lösungen weiterverwendet werden. Pro Generationsschritt wird nur ein neues Individuum generiert. Dieses ersetzt in der Regel das populationsschwächste. Eine stetige Evolutionskurve ist somit gewährleistet. Der Algorithmus terminiert, wenn innerhalb der letzten n Generationen kein besser bewertetes Individuum als das zuletzt beste gefunden wurde.

Kapitel 7

Empirische Resultate

7.1 Einleitung

In diesem Kapitel geht es um die Leistungsfähigkeit des in Kapitel 5 und 6 vorgestellten Algorithmus und die Auswahl zwischen den implementierten Varianten für die Methoden des MA. Zunächst werden in Abschnitt 7.2 die verwendeten Testinstanzen beschrieben, um einen Überblick über die Beschaffenheit der verschiedenen Problembeispiele und deren unterschiedlichen Komplexitätsgrad zu bekommen. Danach folgt in Abschnitt 7.3 eine kurze Zusammenfassung des Auswahlvorganges für die Methoden, die in Konkurrenz zu anderen Verfahren treten. Abschnitt 7.4 vergleicht den hier beschriebenen Memetischen Algorithmus mit anderen Algorithmen zur Lösung des Problems auf zweifachen Zusammenhang und fasst die Ergebnisse zusammen.

7.2 Die Testinstanzen

Um den hier vorgestellten MA zu erproben, wurden Testinstanzen von unterschiedlicher Größe, Struktur und Dichte verwendet. Da durch die Vorverarbeitungsschritte das Problem der Erweiterung eines allgemeinen, zusammenhängenden Graphen G_0 immer auf die Erweiterung eines Baumes reduziert werden kann, ist G_0 in den Testinstanzen immer ein Spannbaum.

Die verwendeten Testinstanzen stammen aus drei unterschiedlichen Quellen. Die erste Gruppe wurde mit einem Problemgraphgenerator von Zhu [48] erstellt und wurde schon in [42, 26, 29] verwendet. Die zweite Gruppe besteht aus Adaptionen von Beispielen aus Reinelts[1] TSPLIB [44], einer wohl bekannten und oft verwendeten Sammlung euklidischer Graphen für das Travelling-Salesman-Problem. Die dritte Gruppe wurde mit einem institutseigenen Generator, implementiert von Ljubić[2], erzeugt, der die beiden vorangegangenen Beispielkategorien um weitere Instanzen ergänzt. Die selbst erzeugten Problembeispiele laufen außer Konkurrenz und wurden für den abschließenden Vergleich nicht verwendet. Sie fungierten als zusätzliche Trainingsinstanzen für die vorab durchgeführten Testläufe zur Auswahl der Parameter des Algorithmus und wirken einer ungewollten Spezialisierung des Verfahrens entgegen.

7.2.1 Testinstanzen basierend auf Zhus Generator

Dieser Generator erzeugt für eine gewählte Knotenanzahl $|V|$ einen zufälligen Graphen G. Ein Parameter für die Dichte gibt die Wahrscheinlichkeit an, mit der eine Kante, ausgehend von einem kompletten Graphen, in die Menge E übernommen wird. Ist zum Beispiel eine Dichte von 0.5 gewählt worden, so enthält der resultierende Graph ca. die Hälfte der Kanten eines kompletten Graphen für dieselbe Knotenmenge. Innerhalb des Graphen $G = (V, E)$ wird nun ein beliebiger Spannbaum fixiert. Seine Kanten bilden die Menge E_0. Die Menge der Erweiterungskanten ergibt sich aus $E_a = E \setminus E_0$. Die Gewichtung der Kanten erfolgt zufällig. Ein Wert aus einem zuvor festgelegten Bereich $[min, max]$ wird jeder Kante zugeordnet. Mit Ausnahme der beiden Instanzen R_1 und R_2 handelt es sich hierbei um ganzzahlige Werte. In Tabelle 7.1 auf Seite 77 sind die Instanzen zusammen mit ihren Eckdaten aufgelistet. In Anhang A ist eine grafische Darstellung zu jeweils einem der Graphen pro Kategorie zu finden.

Ein nach dem gleichen Prinzip arbeitender Generator wurde von Ljubić implementiert, um weitere Testbeispiele zu generieren. Auch hier wird für eine zuvor festgelegte Knotenanzahl $|V|$ ein zufälliger Graph $G = (V, E)$ erstellt, und ein Parameter für die Dichte des Graphen entscheidet, wie viele Kanten erzeugt werden. Ein Unterschied zu

[1] Universität Heidelberg, www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95

[2] Technische Universität Wien, Institut für Computergraphik und Algorithmen

Zhus Generator liegt in der Möglichkeit, wahlweise einen minimalen Spannbaum statt eines beliebigen zum Bilden von $G_0 = (V, E_0)$ zu wählen. Die Gewichtung der Kanten erfolgt zufällig aus einem gewählten Bereich innerhalb der positiven ganzen oder reellen Zahlen.

7.2.2 Testinstanzen basierend auf der TSPLIB

Diese Beispiele beinhalten nur euklidische Graphen. Für $G = (V, E)$ wird die Knotenmenge V unverändert von den TSPLIB-Instanzen übernommen. Die Kantenmenge E wird abhängig vom willkürlich gewählten Parameter k generiert, indem jeder Knoten mit zumindest k seiner topographisch nächstliegenden Nachbarn verbunden wird. Gilt $k = \infty$, so wird ein vollständiger Graph erzeugt. Wie in der TSPLIB üblich wird den Kanten eine Gewichtung entsprechend der Distanz zwischen den Knoten, die sie verbinden, gerundet auf ganze Zahlen zugewiesen. In diesem so erstellten Graph $G = (V, E)$ werden nun die Kanten eines beliebigen minimalen Spannbaumes fixiert und bilden die Kantenmenge E_0. Die Menge der Erweiterungskanten ergibt sich aus $E_a = E \setminus E_0$. Tabelle 7.2 auf Seite 78 gibt einen Überblick über diese Testinstanzen, in Anhang A ist eine grafische Darstellung zu jeweils einem Graphen pro Beispielkategorie zu finden.

Anmerkungen: Da für zwei Instanzen $G^k = (V, E^k)$ und $G^l = (V, E^l)$, die auf derselben Knotenmenge V basieren, sich aber in der Anzahl der generierten Kanten unterscheiden, jeweils ein beliebiger minimaler Spannbaum fixiert wird, gilt zwar $E^k \subset E^l$ wenn $k < l$, aber zumeist auch $E_0^k \neq E_0^l$, was durch die Anordnung der Knoten für diese Instanzen bedingt ist[3]. Daraus folgt, dass eine für G^k gefundene optimale Lösung nicht als obere Schranke für G^l gewertet werden kann. Abbildung 7.1 auf Seite 76 zeigt hierzu ein einfaches Beispiel.

Ergänzend zu den aus der TSPLIB abgeleiteten Probleminstanzen wurden mit dem im vorherigen Abschnitt bereits erwähnten Generator weitere euklidische Problemgraphen generiert. Der Vorgang ist weitgehend ident mit dem Erstellen der Beispiele aus der TSPLIB. Die Unterschiede bestehen in einer zufälligen Anordnung der Knoten des

[3]Siehe auch Darstellungen in Anhang A.

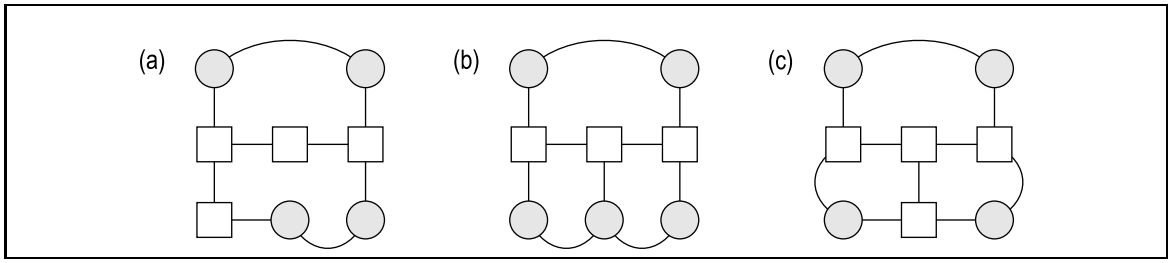


Abbildung 7.1: (a) zeigt einen kleinen euklidischen Graphen, in dem ein minimaler Spannbaum fixiert wurde. Ändert man die Auswahl der Kanten für den Spannbaum, so ändern sich zumeist die Kosten der Erweiterung. Dies kann die Anzahl der Cut-Knoten beeinflussen (b), muss es aber nicht (c).

Graphen und in einer nicht gerundeten Gewichtung der Kanten. Die Kosten der Kanten entsprechen somit reellen Distanzwerten zwischen den Knoten, die sie verbinden.

	Knoten		Kanten				Anzahl der Knoten von Grad(i) in G_0								
Name	$\|V\|$	$\|C\|$	$\|E_a\|$	$\|E_0\|$	Dichte	$Kosten(e)$	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9
A1	20	10	24	19	0.16	$[1, 190]$	10	5	3	1	1	–	–	–	–
A2	30	18	28	29	0.10	$[1, 435]$	12	11	5	1	1	–	–	–	–
A3	40	18	29	39	0.08	$[1, 780]$	22	7	4	6	0	1	–	–	–
A4	30	17	31	29	0.12	$[1, 435]$	13	10	4	2	1	–	–	–	–
A5	40	19	40	39	0.10	$[1, 780]$	21	11	4	0	1	3	–	–	–
B1	60	34	55	59	0.05	$[1, 1770]$	26	18	9	6	1	–	–	–	–
B2	20	10	62	19	0.50	$[1, 190]$	10	4	4	2	–	–	–	–	–
B3	50	30	62	49	0.06	$[1, 1225]$	20	17	8	5	–	–	–	–	–
B4	50	27	63	49	0.08	$[1, 1225]$	23	15	8	1	2	0	1	–	–
B5	60	31	76	59	0.07	$[1, 1770]$	29	15	9	3	4	–	–	–	–
C1	80	46	126	79	0.06	$[1, 3160]$	34	25	15	3	2	0	1	–	–
C2	90	49	127	89	0.05	$[1, 4005]$	41	27	11	6	4	1	–	–	–
C3	100	53	149	99	0.05	$[1, 4950]$	47	27	15	6	2	3	–	–	–
C4	30	14	182	29	0.50	$[1, 435]$	16	7	3	1	3	–	–	–	–
D1	70	36	315	69	0.15	$[1, 2415]$	34	16	13	4	1	2	–	–	–
D2	40	19	354	39	0.50	$[1, 780]$	21	11	2	4	1	0	0	1	–
D3	90	49	366	89	0.15	$[1, 4005]$	41	26	12	7	3	1	–	–	–
D4	80	43	381	79	0.15	$[1, 3160]$	37	24	10	4	3	2	–	–	–
D5	100	55	398	99	0.15	$[1, 4950]$	45	30	17	2	3	2	1	–	–
M1	70	37	290	69	0.15	$[10, 1000]$	33	19	10	5	2	0	1	–	–
M2	80	41	327	79	0.15	$[10, 1000]$	39	24	9	4	0	2	1	0	1
M3	90	42	349	89	0.15	$[10, 1000]$	48	18	15	3	2	2	1	1	–
N1	100	50	1104	99	0.25	$[11, 50]$	50	25	11	8	3	3	–	–	–
N2	110	56	1161	109	0.25	$[11, 50]$	54	26	18	7	2	1	2	–	–
R1	200	115	9715	199	0.50	$[1.0, 100.0]$	85	60	35	13	6	1	–	–	–
R2	200	122	9745	199	0.50	$[5.0, 100.0]$	78	67	40	10	4	1	–	–	–
E1	200	154	19701	199	0.99	euklidisch	46	111	42	1	–	–	–	–	–
E2	300	192	11015	299	0.25	euklidisch	108	113	57	17	5	–	–	–	–
E3	400	238	7621	399	0.10	euklidisch	162	131	67	28	11	1	–	–	–

Tabelle 7.1: Daten für die Testinstanzen erstellt mit dem Generator von A. Zhu. Für jeden Graph $G(V, E)$ sind in der Rubrik Knoten die Anzahl der insgesamt beinhalteten Knoten ($|V|$) und die Anzahl der Cut-Knoten ($|C|$) angegeben. In der Rubrik Kanten folgen die Anzahl der Erweiterungskanten ($|E_a|$) und die Anzahl der zum Bilden von $G_0(V, E_0)$ verwendeten Kanten. Die Spalte Dichte listet den bei der Generierung verwendeten Wert für die Dichte des Graphen auf, $Kosten(e)$ den Bereich für die Kantengewichtung. Die letzte Spalte gibt Auskunft über die Verteilung der Knoten nach Grad innerhalb des Graphen G_0.

	Knoten		Kanten			Knoten mit Grad(i) in G_0			
Name	$\|V\|$	$\|C\|$	$\|E_a\|$	$\|E_0\|$	$Kosten(e)$	i=1	i=2	i=3	i=4
eil101 (∞)	101	68	4950	100	TSP-eukl.	33	41	23	4
a280 (50)	280	218	7654	279	TSP-eukl.	62	160	56	2
a280 (100)	280	220	15769	279	TSP-eukl.	60	164	54	2
a280 (∞)	280	217	38781	279	TSP-eukl.	63	156	61	0
rd400 (∞)	400	304	79401	399	TSP-eukl.	96	214	86	4
pr439 (100)	439	364	26865	438	TSP-eukl.	75	294	67	3
pr439 (200)	439	362	55111	438	TSP-eukl.	77	290	69	3
pr439 (∞)	439	362	95703	438	TSP-eukl.	77	290	69	3
rat575 (50)	575	438	15422	574	TSP-eukl.	137	307	127	4
rat575 (∞)	575	436	164451	574	TSP-eukl.	139	304	127	5
rat783 (30)	783	599	12062	782	TSP-eukl.	184	424	168	7
rat783 (50)	783	601	20717	782	TSP-eukl.	182	427	168	6
d2103 (50)	2103	1963	55630	2102	TSP-eukl.	140	1828	132	3
pcb1173 (30)	1173	953	18321	1172	TSP-eukl.	220	742	204	7

Tabelle 7.2: Daten für die aus der TSPLIB abgeleiteten Testinstanzen. Für jeden Graph $G(V, E)$ sind in der Rubrik Knoten die Anzahl der insgesamt beinhalteten Knoten ($|V|$) und die Anzahl der Cut-Knoten ($|C|$) angegeben. In der Rubrik Kanten folgen die Anzahl der Erweiterungskanten ($|E_a|$) und die Anzahl der zum Bilden von $G_0(V, E_0)$ verwendeten Kanten. Die letzte Spalte gibt Auskunft über die Verteilung der Knoten nach Grad innerhalb des Graphen G_0.

7.3 Die Auswahl der Methoden und Parameter

Im vorangegangenen Kapitel 6 wurden die verschiedenen Varianten für die einzelnen Methoden des Algorithmus vorgestellt. In der folgenden Auflistung werden die einzelnen Optionen noch einmal mit den Parametern zusammengefasst.

Zusammenfassung der Parameter

- Heuristik

  H_1 : Kosten der Kante.
  H_2 : Kosten pro Cut-Knoten.

- lokale Optimierung

  O_1 : Tests von links nach rechts.
  O_2 : Tests in zufälliger Reihenfolge.
  O_3 : Tests in umgekehrter Reihenfolge der Kosten.

- Initialisierung

  I_1 : Initialisiere in zufälliger Reihenfolge und wähle zufällig.
  I_2 : Heuristisch gesteuerte Zufallsauswahl mittels Normalverteilung.
  I_3 : Invers proportionale Zufallsauswahl zwischen zwei Kandidaten.
  I_4 : Binäre Turniere.
  I_5 : Normalverteilte Wahl über alle Kanten.
  I_6 : Heuristisch angepasste, normalverteilte Wahl über alle Kanten.
  I_7 : Normalverteilte Wahl über den aktuellen Nutzen aller Kanten.

- Rekombination:

  R_1 : Vereinige und reduziere.
  R_2 : Schnittmenge und invers proportionale Wahl zwischen zwei Kandidaten.
  R_3 : Schnittmenge und binäre Turniere.

- Mutation:

  M_1 : Lösche eine zufällige Kante.
  M_2 : Wähle zwei Kanten und entscheide proportional.
  M_3 : Binäre Turniere.

- Auffüllstrategie für die Mutation:

  A_1 : Wähle zufällig.
  A_2 : Wähle normalverteilt.

- Strategieparameter für den Ablauf des Algorithmus:

  $tsize$: Anzahl der Teilnehmer an Turnieren zur Selektion.
  $idev$: Standardabweichung für Initialisierungsvarianten mit Normalverteilung.
  dev : Standardabweichung verwendet beim Auffüllen von Kanten.
  pop : Größe der Population.
  $pmut$: Wahrscheinlichkeit für die Mutation.

Um eine Auswahl für spätere Tests treffen zu können, war es notwendig, die einzelnen Variationen zu bewerten. Dies gestaltete sich insofern als schwierig, da eine isolierte Betrachtung der Parameter kaum sinnvoll ist, zumal gerade bei den Methoden des MA eine jede Einfluss auf die Leistungsfähigkeit der anderen nimmt.

Nach einigen Vorabtests, um eine Abschätzung der zu erwartenden Laufzeit zu erhalten, wurde dann entschieden, ganze Parametervektoren zu bewerten, um Fehler bei Rückschlüssen auf den Beitrag einer Methode zu einem guten Ergebnis zu vermeiden. Im Weiteren wurde nun diese Strategie verfolgt:

> Einschränken des Suchraums durch eine Vorauswahl.
> WIEDERHOLE
> Wähle eine repräsentative Teilmenge an Testinstanzen.
> Evaluiere die aktuellen Parametervektoren.
> Verfeinere die Suche.
> BIS ein geeigneter Kandidat gefunden wurde.

Das Ziel der Vorauswahl war es, gute Kombinationen der implementierten Methoden aufzuspüren. Um die Auswahl von Methoden möglichst objektiv durchzuführen, wurde die Suche mit einer sehr großen Menge an Parametervektoren begonnen, die die Kombinationen der zuvor aufgelisteten Varianten abdeckte. Für die weiteren Strategieparameter des MA wurden zunächst Werte gewählt, die sich bei einem verwandten Evolutionären Algorithmus [42] für die E2AUG-Problemstellung als günstig erwiesen hatten.

Die Parametervektoren wurden nun an fünf verschiedenen Probleminstanzgruppen erprobt. Der Komplexitätsgrad der gewählten Beispiele war gering, wenn auch leicht steigend zwischen den Gruppen. Graphen mit 20-200 Knoten wurden verwendet. Bewertet wurde primär die Qualität der Lösung, sekundär die benötigte Laufzeit und die Anzahl an Evaluierungen. Um die Ergebnisse zwischen den Gruppen vergleichen zu können, wurden die Resultate für jede Instanz normiert, wodurch sich prozentuelle Abweichungen im Vergleich zum jeweils besten Ergebnis sowohl innerhalb als auch zwischen den Gruppen auswerten ließen. Eine Reduktion auf knapp über 100 Kandidaten an Kombinationen für die Operatoren des MA wurde in diesem Schritt erzielt.

Für die verkleinerte Menge wurden nun drei neue Gruppen von Testinstanzen gebildet. Die Graphen umfassten hierbei 70 bis 400 Knoten. Um dem erhöhten Einfluss der Strategieparameter bei den komplexer gewählten Instanzen gerecht zu werden, wurden Abwandlungen der Parametervektoren im Bereich der Strategieparameter generiert und der Kandidatenmenge hinzugefügt. Danach wurde erneut bewertet. Die besten 21 Kombinationen wurden zusammen mit ihren bisher geeignetsten Strategieparametern für eine weitere Suche gewählt.

Über die nun folgende verfeinerte Suche wurden robuste Werte für die Strategieparameter bestimmt und eine endgültige Auswahl der Methoden getroffen, indem die bereits dargelegte, iterative Strategie verfolgt wurde. Die Parametermenge wurde um Abwandlungen der verbliebenen Vektoren im Bereich der Strategieparameter ergänzt. Eine neue Gruppe an repräsentativen Beispielen wurde gewählt und die Bewertung der Vektoren vorgenommen. Parametervektoren, die sich als signifikant schlechter erwiesen, schieden aus. Dies wurde wiederholt, bis sich eine überaus robuste Kombination aus Parametern herauskristallisierte. Das Ergebnis lautet wie folgt:

H_1 : Kosten der Kante.
O_3 : Teste in umgekehrter Reihenfolge der Kosten.
I_6 : Heuristisch angepasste, normalverteilte Wahl über alle Kanten.
R_3 : Schnittmenge und binäre Turniere.
M_2 : Wähle zwei Kanten und entscheide proportional.
A_2 : Wähle normalverteilt.

kombiniert mit einer Population von 800 Individuen, einer Turniergröße von 3 für die Selektion, einer Mutationswahrscheinlichkeit von 0.7 und 0.5 für die Parameter der Standardabweichungen. Diese Methoden wurden nun für den MA fixiert und die Strategieparameter für abschließende Testläufe übernommen.

7.4 Vergleiche zu anderen Verfahren

Ohne alternative Lösungsstrategien zur Problemstellung in Betracht zu ziehen, sind Aussagen über die gefundenen Lösungen und zur Leistungsfähigkeit des Algorithmus schwer möglich. Deshalb wurde ein vergleichender Test mit anderen aktuellen Verfahren zum Lösen des Problems der Erweiterung eines Graphen auf zweifachen Zusammenhang durchgeführt.

Die in Konkurrenz tretenden Algorithmen[4] waren der hier vorgestellte Memetische Algorithmus (MA), der heuristische Ansatz von Khuller und Thurimella [24] (KT), der Algorithmus von Zhu et al. [48] (ZKR) und der hybride Genetische Algorithmus von Ljubić und Kratica [26] (LK).

Jedem Algorithmus wurde pro Testlauf ein Zeitfenster von 20 000 Sekunden CPU-Zeit eingeräumt, danach wurde der Lauf gegebenenfalls abgebrochen. Als Probleminstanzen kamen Graphen zum Einsatz, die in der Literatur bereits mehrfach Verwendung fanden. Dies umfasst die Problemgraphen der TSPLIB von G. Reinelt und A. Zhu, wie sie bereits in Abschnitt 7.2 vorgestellt wurden.

Der erste Teil des Algorithmus, die in Kapitel 5 erläuterten Vorverarbeitungsschritte, reduzieren deterministisch die Komplexität der Probleminstanzen, bevor diese dem MA

[4]Siehe auch Kapitel 3

zur weiteren Bearbeitung übergeben werden. Die Tabellen auf den Seiten 85 und 86 fassen die Ergebnisse für die einzelnen Problemstellungen zusammen.

Die Resultate zeigen, dass eine Fixierung von Kanten nur in lichten Graphen wie zum Beispiel den Instanzen B1 und C3 durchgeführt werden kann. Dann jedoch führt sie durch die unmittelbar darauf folgende Verkleinerung des BCT zu einer deutlichen Verringerung der zu berücksichtigenden Cut-Knoten. Für Graphen, bei denen die Mächtigkeit der Menge E_0 größer als die für einen Spannbaum benötigte Anzahl an Kanten ist, würden die zusätzlichen Kanten eine vergleichbare Reduktion der Komplexität der Instanz zur Folge haben. Für alle Testbeispiele gilt $|E_0| = |V|-1$, womit diese Möglichkeit jedoch ausgeschlossen ist.

Als sinnvoll hat sich die Entscheidung erwiesen, die Bedingung einer strikt alternierenden Abfolge von Block- und Cut-Knoten in einem BCT aufzuheben. Für Graphen, in denen kaum oder nur wenig Blöcke zusammengefasst werden können, verringert sich die Zahl der zu verwaltenden Knoten deutlich, und auch die durchschnittliche Pfadlänge innerhalb des BCTs wird entsprechend reduziert.

Für dichte Graphen zeigt sich die Elimination von Kanten als überaus effektiv, die Menge der Erweiterungskanten konnte im Allgemeinen drastisch reduziert werden. Für komplette Graphen wurde bei allen Testinstanzen eine Verringerung auf ca. ein Viertel der ursprünglichen Größe erreicht.

Für jede Probleminstanz wurde der MA 30-mal ausgeführt. Die protokollierten Resultate wurden gemittelt und der Mittelwert für Vergleiche verwendet. Die für alle Testläufe verwendeten Parameter waren: Populationsgröße: 800; Gruppengröße der Turniere zur Selektion von Individuen: 3; Parameter zur Steuerung der Normalverteilung bei der Initialisierung: 0.5; Wahrscheinlichkeit für die Mutation: 0.7; Abbruchkriterium: kein neues bestes Individuum innerhalb von 10000 Iterationen.

KT wurde für jeden Blattknoten einer Instanz einmal gestartet, wodurch jeder dieser Knoten einmal Startknoten für den Branching Algorithmus war. Der beste Lauf wurde gewertet. ZKR generiert für jede Instanz jeweils dieselbe Lösung, daher wurde nur ein Lauf pro Beispiel benötigt. Auf Grund des hohen Bedarfs an Rechenleistung war es jedoch nicht möglich, den Algorithmus erfolgreich für jede Instanz anzuwenden. Nur für weniger umfangreiche Beispiele konnten Lösungen ermittelt werden. Das Gleiche

gilt auch für LK. Die für ihn ermittelten Ergebnisse wurden aus [26] übernommen und stellen jeweils das beste Ergebnis aus 10 Versuchen pro Instanz dar. Die Laufzeiten für KT, ZKR und LK sind nur als Richtwerte zu verstehen, weil sie mit 30 zufälligen Instanzen ermittelt wurden.

Wie die zusammenfassenden Tabellen 7.5 und 7.6 auf den Seiten 87 und 88 zeigen, rangieren die von KT gelieferten Ergebnisse generell hinter denen der anderen Algorithmen, jedoch konnte dieses Verfahren genauso wie MA für alle vorhandenen Instanzen erfolgreich beendet werden. Für jene Instanzen, für die Resultate von ZKR und LK ermittelt werden konnten, fand MA jeweils die optimale Lösung in allen 30 Läufen. Weiters zeigt sich, dass MA auch für größere Problemstellungen gut verwendbar ist. In allen Testläufen konnte eine Abweichung zur optimalen, beziehungsweise zur besten bekannten, Lösung von kleiner als drei Prozent erreicht werden. Die Testläufe pro Instanz weisen sehr geringe Schwankungen auf, wie die ermittelten Standardabweichungen belegen. Die Verlässlichkeit des Verfahrens ist somit sehr hoch. Weiters skalieren die Rechenzeit und Anzahl der benötigten Evaluationen moderat mit dem Grad der Komplexität der Testinstanzen.

Es folgen zunächst die Tabellen mit den Ergebnissen für die Vorverarbeitungsschritte, danach die Vergleiche der Algorithmen. Das Diagramm 7.2 stellt die erreichten Lösungen der einzelnen Verfahren einander gegenüber. Den Abschluss bilden zwei Grafiken. Abbildung 7.3 zeigt beispielhaft zwei Lösungen aus dem durchgeführten Vergleich, Abbildung 7.4 zeigt Beispiel B4 vor und nach der Vorverarbeitung.

	Graph G			BCT T					
Instanz	$\|V\|$	$\|E_a\|$	C(G_0)	$\|V_T\|$	$\|E_A\|$	C(T)	$\frac{\|V_T\|}{\|V_{T_{Std}}\|}$ [%]	$\frac{C(T)}{C(G_0)}$ [%]	$\frac{\|E_A\|}{\|E_a\|}$ [%]
A1	20	24	10	12	10	6	80,0	60,0	41,7
A2	30	28	18	22	16	12	73,3	66,7	57,1
A3	40	29	18	16	15	7	84,2	38,9	51,7
A4	30	31	17	8	6	4	88,9	23,5	19,4
A5	40	40	19	27	28	14	81,5	73,7	70,0
B1	60	55	34	1	0	0	100,0	0,0	0,0
B2	20	62	10	20	41	10	69,0	100,0	66,1
B3	50	62	30	44	42	26	72,1	86,7	67,7
B4	50	63	27	35	34	18	81,0	66,7	54,0
B5	60	76	31	39	35	21	85,0	67,7	46,1
C1	80	126	46	60	76	34	75,3	73,9	60,3
C2	90	127	49	59	64	32	82,8	65,3	50,4
C3	100	149	53	59	67	33	79,7	62,3	45,0
C4	30	182	14	30	104	14	69,8	100,0	57,1
D1	70	315	36	70	218	36	66,7	100,0	69,2
D2	40	354	19	40	209	19	69,0	100,0	59,0
D3	90	366	49	90	278	49	65,2	100,0	76,0
D4	80	381	43	80	274	43	65,6	100,0	71,9
D5	100	398	55	100	301	55	64,9	100,0	75,6
M1	70	290	37	70	227	37	66,0	100,0	78,3
M2	80	327	41	80	242	41	66,7	100,0	74,0
M3	90	349	42	90	262	42	68,7	100,0	75,1
N1	100	1104	50	100	687	50	67,1	100,0	62,2
N2	110	1161	56	110	734	56	66,7	100,0	63,2
R1	200	9715	115	200	3995	115	63,7	100,0	41,1
R2	200	9745	122	200	3702	122	62,3	100,0	38,0
E1	200	19701	154	200	4104	154	56,7	100,0	20,8
E2	300	11015	192	300	4462	192	61,1	100,0	40,5
E3	400	7621	238	400	4806	238	62,8	100,0	63,1

Tabelle 7.3: Die Tabelle stellt den Umfang der Probleminstanzen von Zhu vor und nach der Vorverarbeitung dar. Unter der Rubrik Graph G werden die Anzahl der Knoten ($|V|$), der Erweiterungskanten ($|E_a|$) und der Cut-Knoten (C(G_0)) des Graphen G den entsprechenden Werten des BCT nach den Vorverarbeitungsschritten gegenübergestellt. Die Spalte $|V_T|/|V_{T_{Std}}|$ vergleicht die Anzahl der generierten Knoten der verschiedenen BCT-Darstellungen. Die Größe des verwendeten BCT wird als Prozentwert eines Standard-BCT angegeben. Die letzten beiden Spalten verdeutlichen die Reduktion der Menge an Cut-Knoten und Erweiterungskanten.

	Graph G			BCT T				
Instanz	$\|V\|$	$\|E_a\|$	$C(G_0)$	$\|V_T\|$	$\|E_A\|$	$C(T)$	$\|V_T\|/\|V_{T_{Std}}\|$ [%]	$\|E_A\|/\|E_a\|$ [%]
eil101 (∞)	101	4950	68	101	1734	68	60,1	35,0
a280 (50)	280	7654	218	280	1561	218	56,3	20,4
a280 (100)	280	15769	220	280	3322	220	56,1	21,1
a280 (∞)	280	38781	217	280	10182	217	56,5	26,3
rd400 (∞)	400	79401	304	400	15515	304	56,9	19,5
pr439 (100)	439	26865	364	439	6095	364	54,7	22,7
pr439 (200)	439	55111	362	439	11890	362	54,9	21,6
pr439 (∞)	439	95703	362	439	19574	362	54,9	20,5
rat575 (50)	575	15422	438	575	3424	438	56,8	22,2
rat575 (∞)	575	164451	436	575	34514	436	56,9	21,0
rat783 (30)	783	12062	599	783	2953	599	56,7	24,5
rat783 (50)	783	20717	601	783	4802	601	56,6	23,2
pcb1173 (30)	1173	18321	953	1173	4492	953	55,2	24,5
d2103 (50)	2103	55630	1963	2103	6657	1963	51,7	12,0

Tabelle 7.4: Ergebnisse der Vorverarbeitungsschritte für die aus der TSPLIB erstellten Testinstanzen. Die Tabelle stellt den Umfang der Probleminstanzen vor und nach der Vorverarbeitung dar. Unter der Rubrik Graph G werden die Anzahl der Knoten ($|V|$), der Erweiterungskanten (E_a) und der Cut-Knoten ($C(G_0)$) des Graphen G den entsprechenden Werten des BCT nach den Vorverarbeitungsschritten gegenübergestellt. Die Spalte $|V_T|/|V_{T_{Std}}|$ vergleicht die Anzahl der generierten Knoten der verschiedenen BCT-Darstellungen. Die Größe des verwendeten BCT wird als Prozentwert eines Standard-BCT angegeben. Die Spalte $|E_A|/|E_a|$ verdeutlicht die Reduktion der Menge an Erweiterungskanten. Alle Daten sind gegebenenfalls auf eine Nachkommastelle gerundet.

		KT		ZKR		LK			MA				
	$\$(E_S^*)$	*%gap*	*t* [*s*]	*%gap*	*t* [*s*]	*%gap*	*t* [*s*]	*Eval.*	*%gap*	σ	*t* [*s*]	*Eval.*	*Opt*[%]
A1	722*	1,4	0.1	1,4	0.8	0,0	<0.1	550	0,0	0,0	<1	800	100,0
A2	496*	11,5	0.2	0,0	4.3	0,0	<0.1	550	0,0	0,0	<1	800	100,0
A3	6732*	2,9	0.2	0,0	14.2	0,0	0.1	400	0,0	0,0	<1	800	100,0
A4	1538*	4,6	0.2	0,0	5.5	0,0	0.1	400	0,0	0,0	<1	800	100,0
A5	4823*	5,6	0.2	0,0	18.5	0,0	0.1	600	0,0	0,0	<1	800	100,0
B1	15512*	8,6	0.4	0,0	73.5	0,0	0.4	900	0,0	0,0	<1	0††	100,0
B2	224*	4,0	0.1	0,0	6.1	0,0	0.1	1500	0,0	0,0	<1	958	100,0
B3	7352*	2,7	0.3	0,0	33.1	0,0	0.3	1200	0,0	0,0	<1	800	100,0
B4	5984*	15,4	0.3	0,0	64.4	0,0	0.4	1250	0,0	0,0	<1	800	100,0
B5	13522*	1,1	0.4	0,0	131.9	0,0	0.7	1250	0,0	0,0	<1	800	100,0
C1	22917*	11,9	0.9	0,0	687.2	0,0	3.5	3100	0,0	0,0	1,4	1385	100,0
C2	38191*	5,8	1.1	0,0	1121.2	0,0	4.4	10600	0,0	0,0	<1	800	100,0
C3	59129*	10,0	1.6	1,1	2331.1	0,0	7.6	7300	0,0	0,0	<1	820	100,0
C4	817*	2,9	0.3	0,0	71.3	0,0	0.7	5100	0,0	0,0	1,3	1883	100,0
D1	4957*	5,1	1.1	0,0	1805.5	0,0	6.0	8450	0,0	0,0	7,7	1762	100,0
D2	899*	3,1	0.6	0,0	403.0	0,0	2.4	9900	0,0	0,0	3,2	2900	100,0
D3	20321*	6,1	2.4	0,4	11046.5	0,0	15.9	21850	0,0	0,0	26,9	4509	100,0
D4	8142*	3,3	1.8	0,0	5208.8	0,0	11.2	10300	0,0	0,0	11,8	1831	100,0
D5	19355*	12,5	3.5	0,0	21762.5	0,0	25.5	78350	0,0	0,0	16,4	1960	100,0
M1	2940*	8,2	1.2	0,0	237.0	0,0	6.7	2700	0,0	0,0	8,1	2144	100,0
M2	4600*	5,9	1.7	0,0	503.7	0,0	21.9	8700	0,0	0,0	10,1	1157	100,0
M3	4980*	6,2	2.4	0,4	1178.4	0,0	33.4	8100	0,0	0,0	14,2	1508	100,0
N1	390*	31,5	5.2	1,9	14993.3	4,9	107.0	27800	0,0	0,0	36,9	6264	100,0
N2	429*	39,2	7.0	5,8	16006.1	2,3	147.5	90000	0,0	0,0	68,1	10980	100,0
R1	121,4*	16,8	64.3	1,9	–	–	–	–	0,5	0,6	818,1	38547	6,7
R2	321,4	37,6	91.6	2,0	–	–	–	–	0,8	0,7	391,2	31230	3,3
E1	2873,8*	21,1	–	–	–	–	–	–	0,7	0,3	87,8	44667	3,3
E2	9588,5	33,0	–	–	–	–	–	–	0,9	0,5	863,4	41385	3,3
E3	21605,1	31,1	–	–	–	–	–	–	0,3	0,2	4495,6	59918	3,3

Tabelle 7.5: Resultate für die Probleminstanzen von Zhu. Eine ausführliche Tabellenbeschreibung befindet sich unter Tabelle 7.6 auf Seite 88.

†† Optimale Lösung durch die Vorverarbeitung gefunden.

		KT	ZKR	LK		MA				
Instanz	$Kosten(E_S^*)$	*%-gap*	*%-gap*	*%-gap*	*Eval.*	*%-gap*	σ	t $[s]$	*Eval.*	*Opt.*[%]
eil101 (∞)	193*	32,6	–	–	–	0,1	0,3	18,9	12598	93,3
a280 (50)	474*	25,1	–	–	–	0,1	0,1	27,6	5420	56,7
a280 (100)	473*	29,4	–	–	–	0,1	0,4	62,3	15487	76,7
a280 (∞)	490	21,6	–	–	–	1,3	0,8	102,7	24636	6,7
rd400 (∞)	4057	28,5	–	–	–	2,2	1,4	780,5	48299	3,3
pr439 (100)	27907	20,5	–	–	–	0,5	0,4	245,6	34450	10,0
pr439 (200)	28518	18,1	–	–	–	0,9	0,5	352,6	35548	3,3
pr439 (∞)	27940	19,9	–	–	–	1,5	1,1	663,5	41361	3,3
rat575 (50)	1518	34,5	–	–	–	0,1	0,1	646,2	40371	20,0
rat575 (∞)	1558	33,4	–	–	–	1,9	1,1	1925,8	40331	3,3
rat783 (30)	2015	35,2	–	–	–	0,5	0,1	1938,0	57317	3,3
rat783 (50)	2021	34,8	–	–	–	0,5	0,5	2030,7	72116	6,7
pcb1173 (30)	11464	28,1	–	–	–	0,3	0,1	5551,8	77509	3,3
d2103 (50)	7333	9,6	–	–	–	<0,1	<0,1	15389,2	18885	10,0

Tabelle 7.6: Resultate für die aus der TSPLIB abgeleiteten Probleminstanzen. In der Spalte $Kosten(E_S^*)$ wird jeweils der beste bekannte Lösungswert angegeben, mit „*“ markierte Werte kennzeichnen eine optimale Lösung. Ausgehend von diesen Werten wird die Differenz (*%-gap*) zu den von den vier hier betrachteten Algorithmen gefundenen Lösungen E_S mit der Formel $\%\text{-}gap = \frac{Kosten(E_S) - Kosten(E_S^*)}{Kosten(E_S^*)} \cdot 100$ berechnet. Sie stellt somit die prozentuelle Abweichung dar. Für nicht-deterministische Verfahren ist zusätzlich die Anzahl der benötigten Evaluierungen (*Eval.*) bis zum Finden der besten Lösung angegeben. Für MA wurden alle Werte gemittelt. σ bezeichnet die daraus resultierende Standardabweichung für die *%-gap*-Werte des MA, t $[s]$ die benötigte Prozessorzeit in Sekunden auf einem Pentium III/800MHz PC. Der unter *Opt.* angeführte Prozentwert gibt an, wie oft MA in den Testläufen die optimale beziehungsweise beste bekannte Lösung erreicht hat. Alle Ergebnisse wurden auf eine Nachkommastelle gerundet.

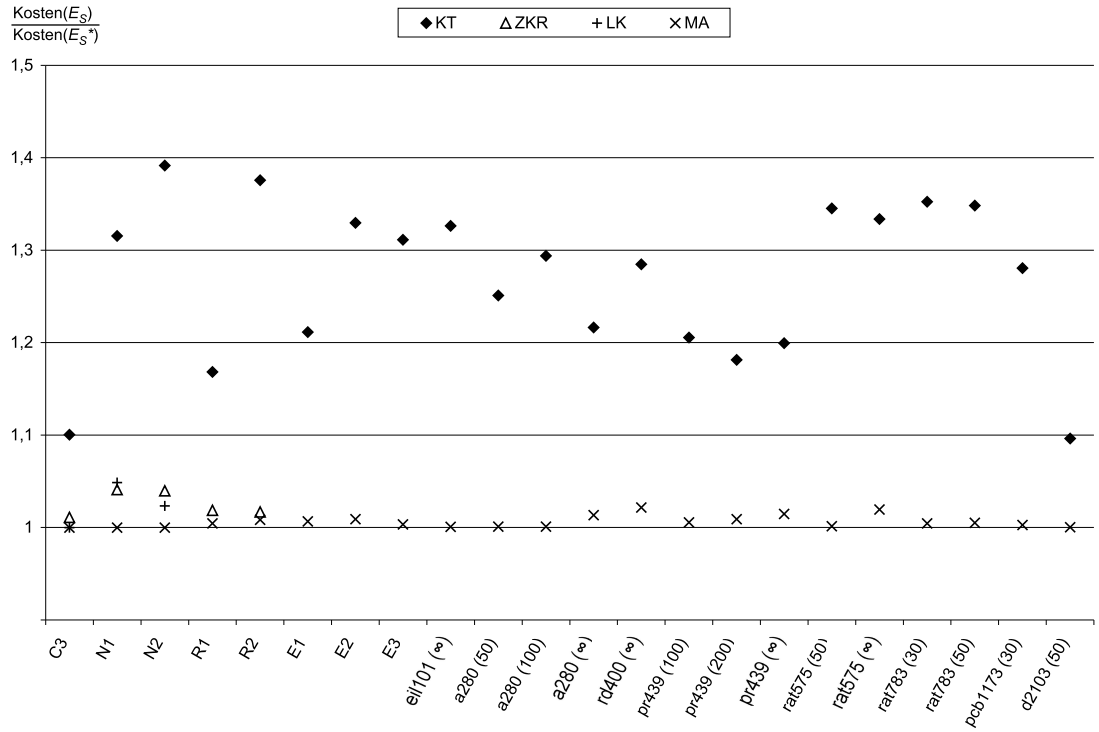


Abbildung 7.2: Das Diagramm stellt die erzielten Ergebnisse der vier Algorithmen für Testinstanzen mit 100 oder mehr Knoten dar. Entlang der horizontalen Achse sind die entsprechenden Problemgraphen in der Reihenfolge aufgelistet, in der sie auch in den vorangegangenen Tabellen aufscheinen. Ausgehend von der optimalen beziehungsweise besten bekannten Lösung wurden die erzielten Resultate normiert. Die vertikale Achse veranschaulicht somit den Faktor, um den die generierten Lösungen abweichen. Ein Faktor von 1.0 kennzeichnet das Erreichen der optimalen bzw. besten bekannten Lösung. Analog zu den Tabellen werden die unterschiedlichen Algorithmen mit KT, ZKR, LK und MA abgekürzt. Konnte ein Verfahren keine Lösung ermitteln, so fehlt der entsprechende Eintrag im Diagramm.

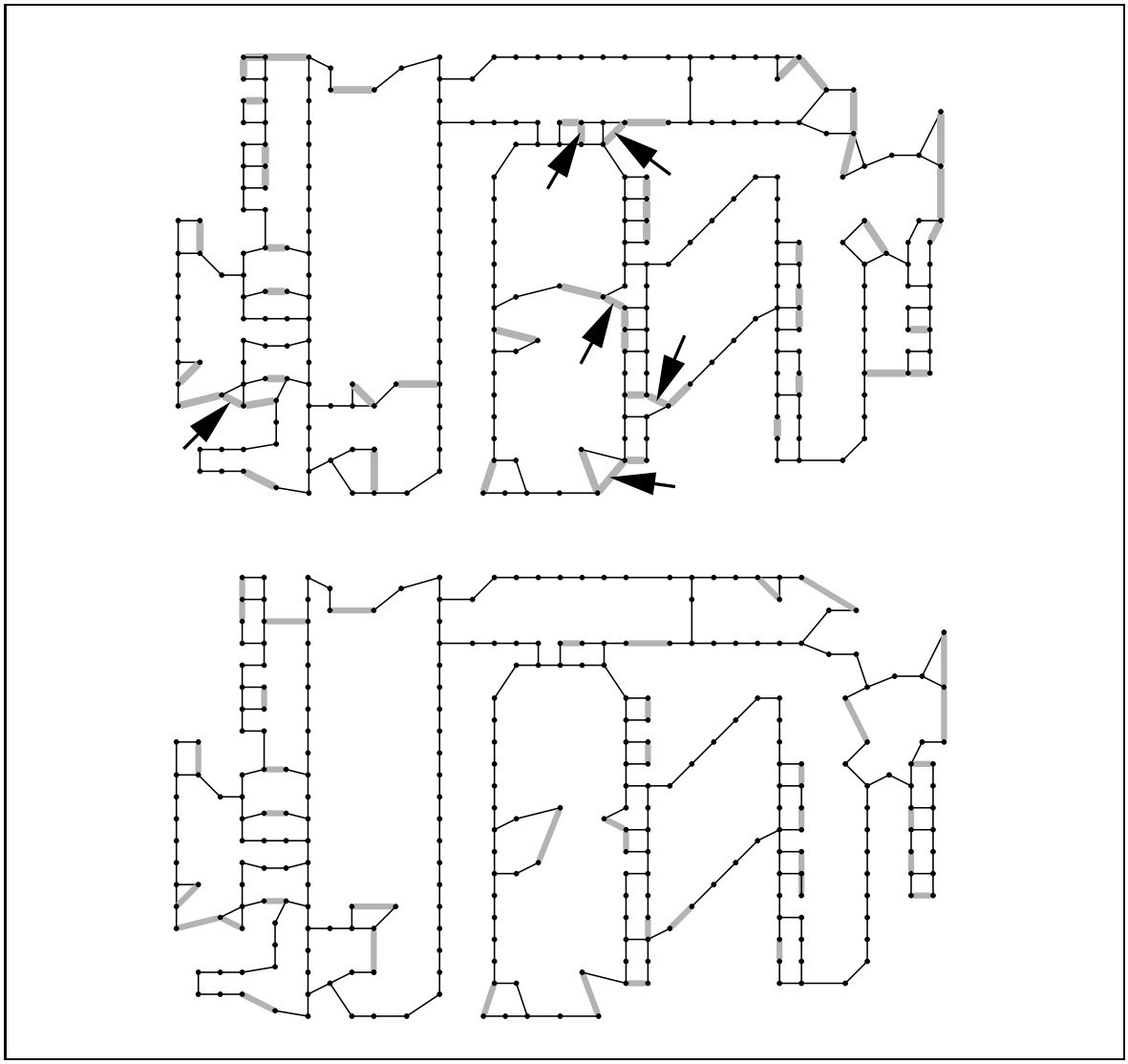


Abbildung 7.3: Zwei Lösungen für den euklidischen Problemgraphen a280 (100). Das obere Bild zeigt die von Khuller und Thurimellas Algorithmus generierte Lösung. Es wurden 53 Kanten zur Erweiterung des Graphen gewählt. Im unteren Bild ist eine Lösung generiert von MA dargestellt, es werden 40 Kanten verwendet. Mittels eines Branch-and-Cut Algorithmus wurde nachgewiesen, dass es sich um eine optimale Lösung handelt. Die zur Erweiterung verwendeten Kanten sind grau eingezeichnet, die Pfeile im oberen Bild markieren Beispiele für redundante Kanten.

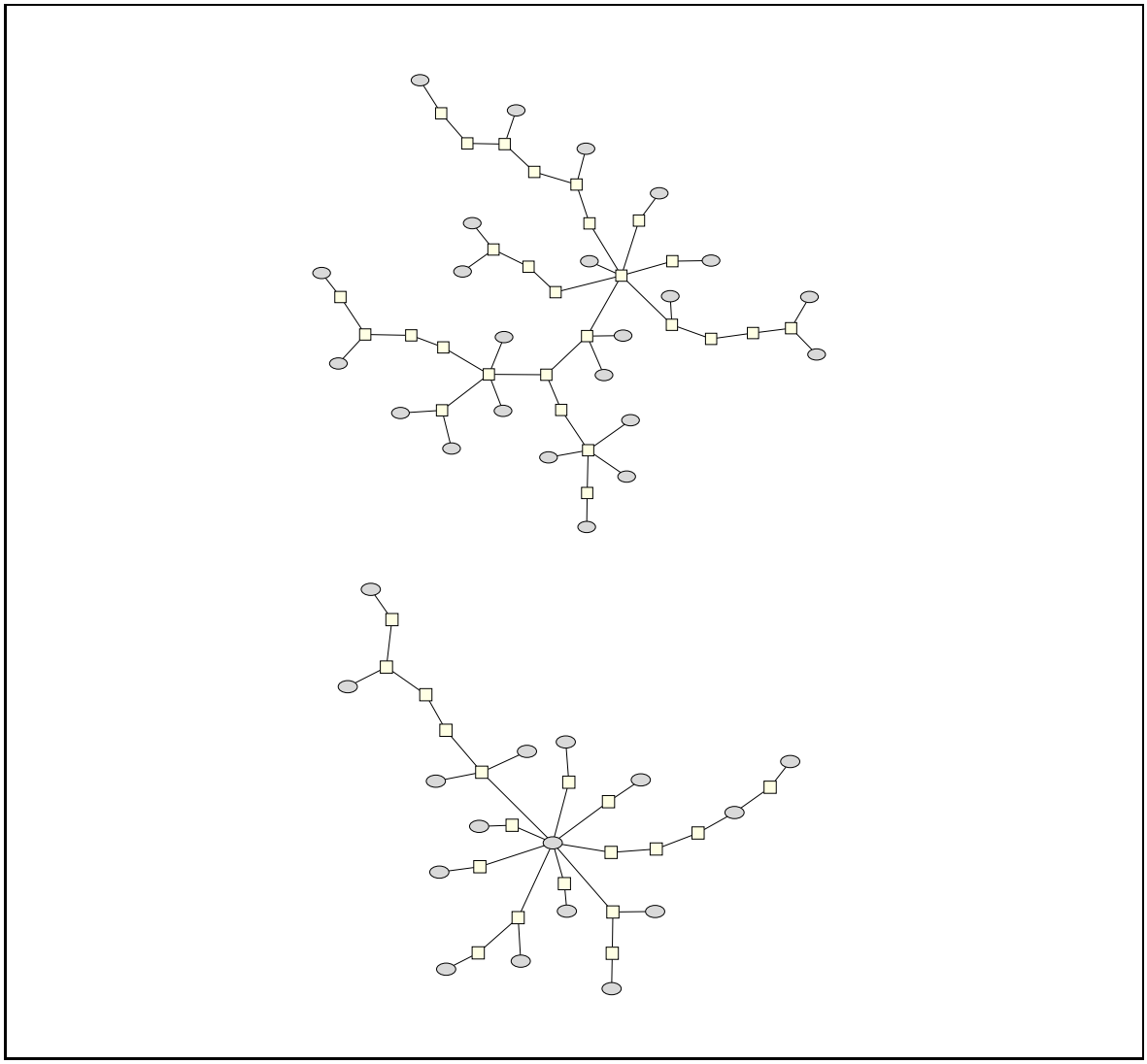


Abbildung 7.4: Ein Beispiel zur Reduktion der Problemkomplexität durch die Vorverarbeitung. Das obere Bild zeigt die Instanz B4, bevor die Vorverarbeitungsschritte ausgeführt wurden. Der Graph besteht aus insgesamt 50 Knoten. 27 davon sind Artikulationsknoten und als Quadrate markiert. Unten wird der BCT nach Ausführung der Vorverarbeitungsschritte dargestellt. Es müssen nur noch 18 Cut-Knoten berücksichtigt werden, der Rest wurde durch das Fixieren von Kanten zu Blöcken zusammengefasst.

7.4.1 Weiterführende Untersuchungen

Um die Leistungsfähigkeit des MA ausführlicher zu untersuchen wurden im mittleren Komplexitätsbereich zusätzliche Testläufe mit dafür angepassten Parametern durchgeführt [27]. Im Rahmen dieser Arbeit wurde auch gezeigt, dass bei der direkten Abfolge von Rekombination und Mutation nicht notwendigerweise zwischen beiden Operatoren lokal optimiert werden muss. Alle weiteren Daten beziehen sich auf einen dahingehend modifizierten MA.

Hierzu wurden fünf Beispiele aus der TSPLIB ausgewählt, für die jeweils drei Instanzen unterschiedlicher Dichte generiert wurden:

- Ein kompletter Graph. Instanzen: pr226, lin318, pr439, pcb442 und pa561.
- Ein Graph mit geringer Dichte, bei dem nur jene Kanten generiert wurden, die einen jeden Knoten mit den $\lceil |V| \cdot 10\% \rceil$ naheliegendsten seiner Nachbarknoten verbinden. Instanzen: lin318-sp, pr439-sp, pcb 442-sp und pa561-sp. Für Instanz pr226-sp wurden die jeweils 15% naheliegendsten Nachbarknoten verbunden, da eine Erweiterung auf zweifachen Zusammenhang bei 10% nicht möglich ist.
- Ein Graph resultierend aus der Berechnung von Delaunay Triangulationen. Dies sind die Instanzen pr225-dt, lin318-dt, pr439-dt und pcb442-dt.

Für jede Instanz wurde ein minimaler Spannbaum fixiert. Die Kosten der Kanten sind euklidisch, abgesehen von den Instanzen pa561 und pa561-sp. Hier wurden sie direkt durch eine Matrix vorgegeben. Tabelle 7.7 auf Seite 93 gibt eine Übersicht über die Testinstanzen und die Ergebnisse der Vorverarbeitung.

Ergebnisse vergleichender Testläufe

Mit den oben beschriebenen Instanzen wurden weitere vergleichende Testläufe zwischen MA, KT, ZKR und LK durchgeführt. Die Bedingungen für diese Testläufe sind ident mit den zuvor durchgeführten, bis auf folgende Ausnahmen: MA wurde mit 5 statt 3 Kandidaten für die Tournament-Selection gestartet, und für den Strategie-Parameter s

Instance	$\|V\|$	*Kosten*(e)	$\|E_a\|$	C(G_0)	$\|V_T\|$	$\|E_A\|$	C(T)	$t_{pre}[s]$	$\frac{C(T)}{C(G_0)}$ [%]	$\frac{\|E_A\|}{\|E_a\|}$ [%]
pr226-dt	226	euklidisch	947	192	226	617	192	1.4	100,0	65,2
pr226-sp	226	euklidisch	3987	187	226	2550	187	1.7	100,0	64,0
pr226	226	euklidisch	25200	187	226	7089	187	11.0	100,0	28,1
lin318-dt	318	euklidisch	1563	248	318	1057	248	10.7	100,0	67,6
lin318-sp	318	euklidisch	5495	246	318	1874	246	12.6	100,0	34,1
lin318	318	euklidisch	50086	246	318	9473	246	95.6	100,0	18,9
pr439-dt	439	euklidisch	2156	358	439	1349	358	34.1	100,0	62,6
pr439-sp	439	euklidisch	11183	366	439	3026	366	42.7	100,0	27,1
pr439	439	euklidisch	95703	366	439	18700	366	280.2	100,0	19,5
pcb442-dt	442	euklidisch	2131	347	442	1298	347	42.9	100,0	60,9
pcb442-sp	442	euklidisch	10528	345	442	2557	345	53.7	100,0	24,3
pcb442	442	euklidisch	97020	345	442	19824	345	299.5	100,0	20,4
pa561-sp	561	Matrix	18504	406	561	5175	406	157.2	100,0	28,0
pa561	561	Matrix	156520	406	561	40601	406	417.6	100,0	25,9

Tabelle 7.7: Übersicht über zusätzliche Testinstanzen und die Ergebnisse der Vorverarbeitung. $|V|$ gibt die Anzahl der Knoten des Graphen G_0 an, die Spalte *Kosten*(e) die Art der Kantengewichtung und $|E_a|$ die Anzahl der Erweiterungskanten. Die Spalten $C(G_0)$ und $C(T)$ geben Auskunft über die Anzahl der in G_0 und in T vorkommenden Cut-Knoten. V_T gibt die Anzahl der Knoten des BCT T an, $|E_A|$ die Anzahl der verbliebenen Erweiterungskanten in T. In der Spalte $t_{pre}[s]$ wird die für die Vorverarbeitung benötigte Zeit in Sekunden aufgelistet. Die letzten beiden Spalten verdeutlichen die erreichte Reduktion in Bezug auf die Anzahl der Cut-Knoten und Erweiterungskanten.

der Normalverteilung wurde ein Wert von 2.5 gewählt. Wie auch in den zuvor beschriebenen Vergleichsläufen galt als oberste Schranke für die benötigte Zeit 20 000 Sekunden für einen Lauf. Innerhalb dieser Zeitvorgabe konnte ZKR für keine der Instanzen beendet werden und wird daher in Tabelle 7.8 nicht berücksichtigt. LK konnte nur teilweise Ergebnisse erzielen. Tabelle 7.8 auf Seite 95 führt in der Spalte $Kosten(E_S^*)$, aus Platzgründen $\$(E_S^*)$ in der Tabelle, die Kosten der besten bekannten Lösung an. Ein „*“ kennzeichnet optimale Lösungen (nachgewiesen mit einem Branch and Cut Algorithmus). Die Spalte $|E_S^*|$ gibt die Anzahl der für die optimale Lösung notwendigen Kanten an. Die durchschnittliche prozentuelle Abweichung der verschiedenen Verfahren zu $Kosten(E_S^*)$ findet sich in Spalte *%-gap*. $t_{[s]}$ ist die gemittelte Laufzeit in Sekunden. Die Standardabweichung von %-gap ist für LK und MA in der Spalte σ vermerkt, *Eval.* zeigt die benötigte Anzahl an Evaluationen. *Opt.*[%] gibt darüber Auskunft, in wie vielen Läufen eine optimale Lösung gefunden wurde (Prozentwert).

Wie aus der Tabelle ersichtlich ist, sind die Ergebnisse des MA hinsichtlich der Qualität denen von KT und LK überlegen. Auch sind die Abweichungen zwischen den einzelnen Läufen sehr gering, wie sich an Hand der niedrigen Standardabweichungen erkennen lässt. Bezüglich der Laufzeiten skaliert der MA moderat bei steigender Komplexität der Testinstanzen.

Auswirkung der Vorverarbeitung

Die für den MA entwickelte Vorverarbeitung ist auch für andere Algorithmen verwendbar. Durchgeführte Versuche mit KT, ZKR und LK haben gezeigt, dass sich die benötigte Laufzeit der Algorithmen signifikant verringert, die gefundenen Lösungen aber qualitativ nicht wesentlich besser werden. Getestet wurde für die Instanzen A,B,C,D,M,N,R aus Tabelle 7.1 und jene aus Tabelle 7.7, für die die einzelnen Algorithmen innerhalb der vorgegebenen 20 000 Sekunden Lösungen ermitteln konnten. Für KT ergab sich eine Zeitersparnis um den Faktor 0.62, für ZKR 0.81 und für LK 0.27. Die durchschnittliche Abweichung zur besten bekannten Lösung (*%-gap*) reduzierte sich bei KT um den Faktor 0.92, 0.96 bei ZKR und 0.91 bei LK. Trotz dieser Verringerung bleibt die Qualität der gefundenen Lösungen insbesondere bei den komplexeren Testinstanzen deutlich hinter denen des MA zurück.

			KT		LK			MA				
Instanz	$\$(E_S^*)$	$\|E_S^*\|$	*%gap*	t [s]	*%gap*	σ	t [s]	*%gap*	σ	t [s]	*Eval.*	*Opt*[%]
pr226-dt	25152*	25*	19.6	1.4	26.6	8.8	47.8	0.0	0.0	3.9	1910	100.0
pr226-sp	22824*	24*	22.5	11.7	27.3	4.8	640.2	0.1	0.6	17.6	9800	96.7
pr226	22824*	24*	22.0	138.9	–	–	–	2.6	1.2	33.2	13073	16.7
lin318-dt	12013*	45*	28.1	5.0	20.9	2.3	246.7	<0.1	<0.1	26.0	9895	0.0
lin318-sp	11797*	46*	29.3	33.2	41.1	3.8	2633.7	0.3	0.3	40.8	27130	6.7
lin318	11797*	46*	29.3	620.4	–	–	–	1.0	0.5	128.9	23391	0.0
pr439-dt	28310*	52*	20.6	8.3	25.1	6.0	491.6	0.0	0.0	52.6	7557	100.0
pr439-sp	26800*	48*	21.2	71.0	40.5	4.3	13471.2	1.1	0.7	79.5	12164	20.0
pr439	26800*	48*	21.2	1498.5	–	–	–	2.5	1.1	408.1	22301	0.0
pcb442-dt	10328*	60*	31.4	12.3	21.4	3.1	320.5	0.1	0.1	67.6	9306	43.3
pcb442-sp	10460*	60*	32.6	106.5	33.8	5.2	18429.2	0.3	0.2	91.9	16902	6.7
pcb442	10478	61	30.5	2030.8	–	–	–	0.3	0.2	366.3	21493	3.3
pa561-sp	782	101	31.1	303.0	–	–	–	0.3	0.1	250.0	20868	3.3
pa561	784	101	30.4	5482.7	–	–	–	0.4	0.4	599.8	34710	13.3

Tabelle 7.8: Ergebnisse weiterer vergleichender Testläufe.

Analyse der lokalen Optimierung

Um die verwendeten Probleminstanzen und den Einfluss der lokalen Optimierung weiter zu untersuchen, wurde eine Fitness-Distance Korrelations-Analyse durchgeführt, wie sie von J. Jones und S. Forrest [22] und P. Merz und B. Freisleben [31, 32] vorgestellt wurde. Für jede Probleminstanz wurden 10 000 mögliche Lösungen zufällig generiert und anschließend lokal optimiert, wie dies auch bei der Initialisierung des MA erfolgt. Die optimierten Lösungen werden bewertet und ihre Abweichung zur optimalen Lösung im Suchraum wird berechnet. Als Maß für die Abweichung wurde der Betrag der symmetrischen Differnz der entsprechenden Kantenmengen verwendet. Abbildung 7.5 auf Seite 97 stellt dies für die Instanzen R2 und pr439 grafisch dar. Die Darstellungen für die anderen Testinstanzen weisen eine ähnliche Struktur auf. Jeder Punkt in der Darstellung repräsentiert eine lokal optimierte Lösung. Optimale Lösungen liegen im Schnittpunkt der beiden Achsen im Punkt (0,0).

In Tabelle 7.9 auf Seite 98 sind die Fitness-Distance Korrelationskoeffizienten ρ für zehn der größeren Instanzen mit bekannten globalen Optima angegeben. Bei allen be-

trachteten Probleminstanzen zeigt sich, dass die Bewertung der Individuen mit ihrer Abweichung zum globalen Optimum korreliert ($0,51 \leq \rho \leq 0,71$). Dies deutet darauf hin, dass ein Evolutionärer Algorithmus zur Lösung dieser Problemstellungen gut geeignet sein kann. Des weiteren liegen alle lokalen Optima in der Darstellung nahe beieinander und weisen eine relativ große Distanz zum globalen Optimum auf. Daraus ergibt sich, dass das zufällige Generieren und anschließende Optimieren von Lösungen allein noch keine effektive Methode zur Problemlösung darstellt. Man beachte, dass aus der grafischen Darstellung jedoch nicht geschlossen werden kann, dass die lokalen Optima auch im Suchraum nahe beieinander liegen. Tabelle 7.9 auf Seite 98 zeigt die durchschnittliche Distanz der lokal optimierten und in der Grafik dargestellten Lösungen untereinander ($\overline{d_{lok}}$) und ihre durchschnittliche Distanz zum globalen Optimum ($\overline{d_{opt}}$). Aus der Tabelle ist ersichtlich, dass die durchschnittliche Entfernung untereinander jeweils größer ist, als die Entfernung zum globalen Optimum. Dies deutet darauf hin, dass das globale Optimum mehr oder weniger in der Mitte des Bereichs des Suchraums liegt, in dem sich die lokalen Optima befinden. In Merz und Freisleben [31] wird von solchen Problemstellungen gesagt, dass sie eine so genannte Big-Valley Struktur aufweisen, und dass von Rekombinationsoperatoren, die möglichst viele Eigenschaften bewahren, die die Elternindividuen gemeinsam haben, gute Ergebnisse zu erwarten sind.

Der Einsatz von Rekombinations- und Mutationsoperatoren mit anschließender lokaler Optimierung ist nur dann sinnvoll, wenn sie mit hoher Wahrscheinlichkeit Lösungen erzeugen, die sich vom Ausgangsindividuum beziehungsweise den Vorgängerlösungen unterscheiden. Ist dies nicht der Fall, deutet das darauf hin, dass der Operator möglicherweise entfernt werden kann, ohne dass die Qualität der Ergebnisse des Algorithmus sinkt. In Tabelle 7.9 sind die durchschnittlichen Wahrscheinlichkeiten angegeben, mit der die Mutation (PD_{mut}) des MA, bzw. die Rekombination (PD_{rek}) jeweils gefolgt von der lokalen Optimierung eine Lösung generiert, die nicht von der oder den Ausganglösungen abweicht. Die Wahrscheinlichkeiten wurden jeweils über einen kompletten Lauf des MA ermittelt und liegen maximal bei 18.5%. In vier von fünf Fällen wird also durch die Variationsoperatoren eine neue Lösung generiert. Insbesondere bei der Mutation ist die Wahrscheinlichkeit gering, dass sie bei dichten Graphen eine mit der Ausgangslösung idente Lösung generiert.

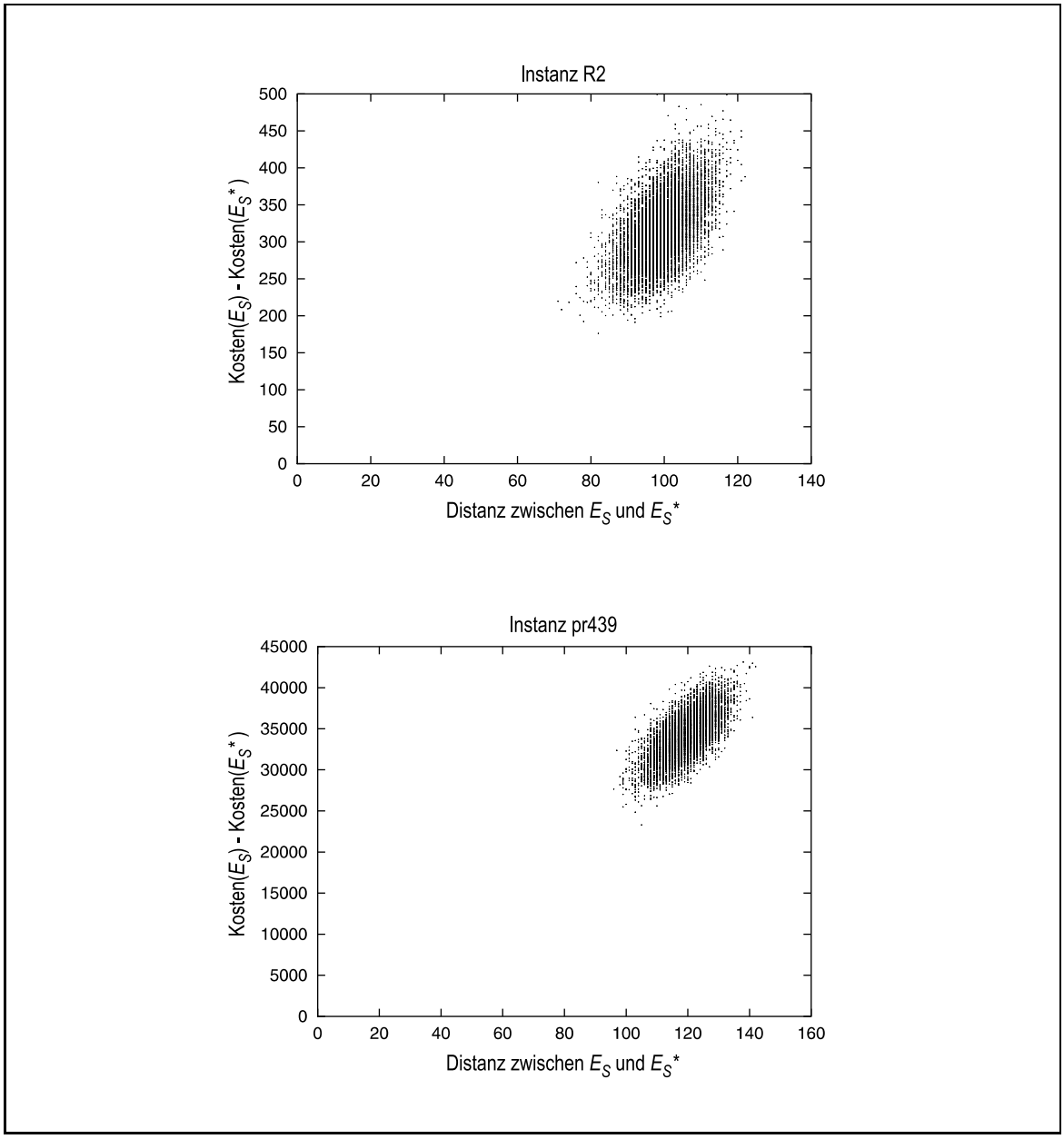


Abbildung 7.5: Darstellung der Abhängigkeit zwischen der Bewertung und der Distanz zum globalen Optimum (E_S^*) von zufällig generierten und anschließend optimierten Lösungen (E_S).

Instanz	ρ	$\overline{d_{opt}}$	$\overline{d_{lok}}$	PD_{rek}	PD_{mut}
N1	0,57	53,4	62,7	15,3	14,1
N2	0,54	58,2	69,4	13,8	14,3
R1	0,51	107,5	116,9	13,0	3,4
R2	0,55	99,9	118,2	12,5	4,5
pr226-sp	0,53	66,6	71,5	17,0	12,6
pr226	0,52	66,8	71,0	18,4	2,0
lin318-sp	0,65	97,7	115,2	15,5	14,5
lin318	0,63	95,8	115,8	15,4	2,3
pr439-sp	0,71	113,9	129,5	14,0	10,2
pr439	0,68	119,4	125,8	13,4	1,3

Tabelle 7.9: Die linke Hälfte der Tabelle zeigt die Fitness-Distance Korrelations-Koeffizienten ρ und die gemittelte Distanz der zufällig generierten und anschließend lokal optimierten Lösungen untereinander ($\overline{d_{lok}}$) und zur optimalen Lösung ($\overline{d_{opt}}$). In der rechten Hälfte wird die Wahrscheinlichkeit angegeben, mit der die Mutation (PD_{mut}) und die Rekombination (PD_{rek}) gefolgt von der lokalen Optimierung eine Lösung generieren, die sich nicht von der bzw. den Ausgangslösungen unterscheidet.

Kapitel 8

Programme und Hilfsprogramme

8.1 Generelles

Für diese Diplomarbeit wurden die Programme `vbca`, `vbcutp`, `gr2tex` und diverse Skripte zum Generieren von Statistiken, Testläufen, Konvertieren von Textformaten u.ä. implementiert. Mit dem Ende der Implementation und der anschließenden Auswahl der Parameter haben die Skripte ihre Bedeutung verloren und werden daher nicht weiter vorgestellt. `vbcutp` und `gr2tex` sind kleinere Hilfsprogramme, der Algorithmus zusammen mit der Vorverarbeitung und einer Visualisierungshilfe für Graphen ist im Programm `vbca` enthalten.

Alle Programme wurden der Vorgabe entsprechend für das Betriebssystem Linux entwickelt. Verwendet wurden die Distributionen von SuSE in den Versionen 7.x bis 8.0. Als Entwicklungsumgebung kam KDevelop [37] in den Versionen 1.x bis 2.1 zum Einsatz, die Programmdokumentation wurde mit Doxygen [18] Version 1.2.13.1 gestaltet. Die Programmierung erfolgte objektorientiert unter C++. Zum Übersetzen wurde der GNU g++ Compiler Version 2.95.3 verwendet. Verwendete Programmbibliotheken: STL (Standard Template Library), EALIB [40], ein Framework für Evolutionäre Algorithmen, in der Version 1.0 und LEDA [2], eine Sammlung von Datentypen und Algorithmen, in der Version 4.2.

8.2 vbca

`vbca` ist das Hauptprogramm dieser Arbeit. Es beinhaltet das in Kapitel 5 vorgestellte Preprocessing und den in Kapitel 6 dargelegten Algorithmus. Weiters ermöglicht es eine grafische Darstellung von Probleminstanzen. Das Programm ist modular aufgebaut und bietet die Möglichkeit, sowohl das Preprocessing als auch den Viewer separat zu verwenden beziehungsweise wiederzuverwenden. Die Implementation ist sehr umfangreich – der gut kommentierte Programmcode würde exklusive importierter Klassen ausgedruckt[1] 200 Seiten benötigen. Eine detaillierte Programmdokumentation liegt im HTML-Format bei. Generell wurde bei der Programmierung auf Übersichtlichkeit, eine gute Gliederung und hohe Effizienz geachtet. Für oft verwendete Datenstrukturen wurden Zeitmessungen durchgeführt und die sinnvollsten Alternativen gewählt.

8.2.1 Implementation

Die wichtigsten der für `vbca` realisierten Klassen lassen sich den vier Bereichen grundlegende Datenstrukturen, Vorverarbeitung, Memetischer Algorithmus und Visualisierung zuordnen. Weitere kleine Klassen, Templates und Konstrukte finden innerhalb des Programms an mehreren Stellen Verwendung, brauchen jedoch nicht extra besprochen zu werden. Für jeden der zuvor genannten Bereiche folgt einleitend ein UML-Diagramm, das die Zusammenhänge zwischen den Klassen erläutert, danach werden die Aufgaben der entsprechenden Klassen beschrieben.

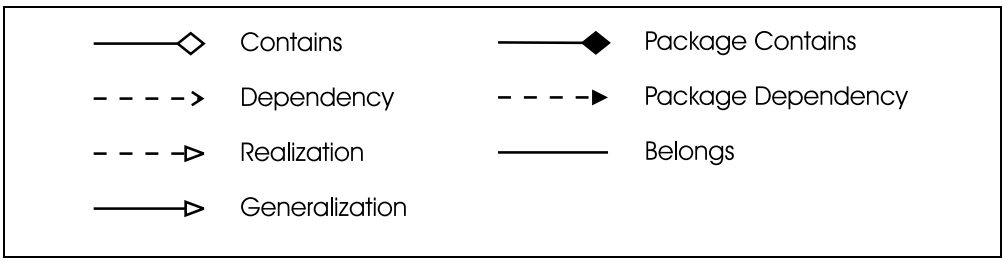


Abbildung 8.1: Legende für die nachfolgenden Diagramme.

[1] a2ps -T3 -c -1

Grundlegende Datenstrukturen

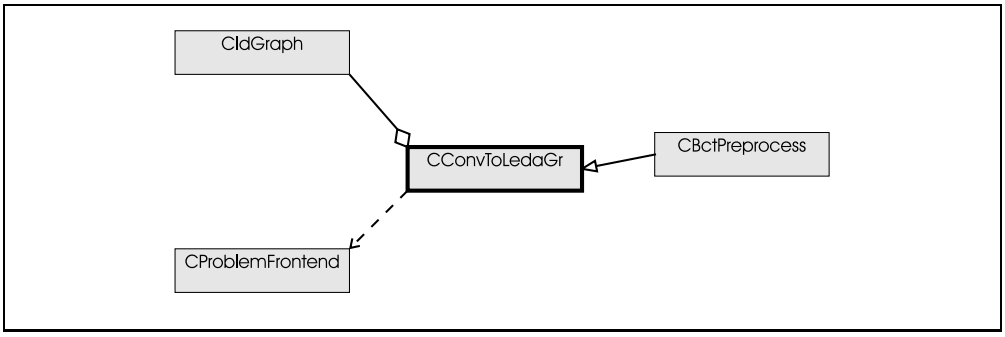


Abbildung 8.2: Diagramm für die Klasse CConvToLedaGr.

CIdGraph ist eine allgemein gehaltene Klasse und lässt sich für andere Problemstellungen gut wiederverwerten. Sie verwaltet einen Graphen, dessen Knoten und Kanten durch pro Instanz eindeutige, fortlaufende Identifikationsnummern (IDs) angesprochen werden. Im Gegensatz zu einer direkten Referenzierung der Knoten- und Kantenobjekte über Zeiger ist auf diese Weise eine für den Programmierer wesentlich leichter zu verwendende und zu überprüfende Schnittstelle zur Handhabung eines Graphen verwirklicht worden, die schwer zu findenden Fehlern entgegenwirkt. Das Kernstück dieser Klasse bildet ein Graph-Objekt aus der LEDA-Library. Eine genaue Beschreibung eines solchen Graphen ist im LEDA-Manual zu finden. Durch das Verwenden von IDs ist es auch einfach möglich, zusätzliche Informationen über Knoten und Kanten getrennt vom Graph zu verwalten und für mehrere Instanzen eines CIdGraphen zu verwenden.

CIdGraph und die davon abgeleitete Klasse CBCTIdGraph werden dazu verwendet, die strukturelle Information eines Problemgraphen respektive des davon abgeleiteten BCT wiederzugeben. Zusätzliche Informationen werden in Tabellen verwaltet, deren Elemente durch die Datenklassen CNodeInfo, die davon abgeleitete Klasse CBCTNodeInfo und CEdgeInfo gebildet werden.

Die Klasse CConvToLedaGr fasst die zuvor besprochenen Elemente zusammen. Sie liest die als Textfile vorliegende Testinstanz ein, baut entsprechende Datenstrukturen auf und stellt diese zur Abfrage bereit. Dies sind ein CIdGraph zur Darstellung des Problemgraphen, Tabellen für zusätzliche Knoten- und Kanteninformationen und eine Un-

terteilung der Kanten in die Menge der bereits im Problemgraph enthaltenen Kanten (E_0) und die Menge der Erweiterungskanten (E_A). CConvToLedaGr selbst wird nicht instanziert sondern dient als Grundlage für eine Klasse der Vorverarbeitung.

CProblem ist eine vom Institut für Computergrafik und Algorithmen entwickelte Klasse, die in der späten Phase der Entwicklung dem Projekt hinzugefügt wurde, um eine einheitliche I/O-Schnittstelle für verschiedene Algorithmen zu schaffen. Sie bietet eine alternative Möglichkeit, Testinstanzen einzulesen und Lösungen auszugeben. CProblem-Frontend baut auf dieser Klasse auf und ermöglicht eine einfachere Handhabung dieser I/O Klasse.

Vorverarbeitung

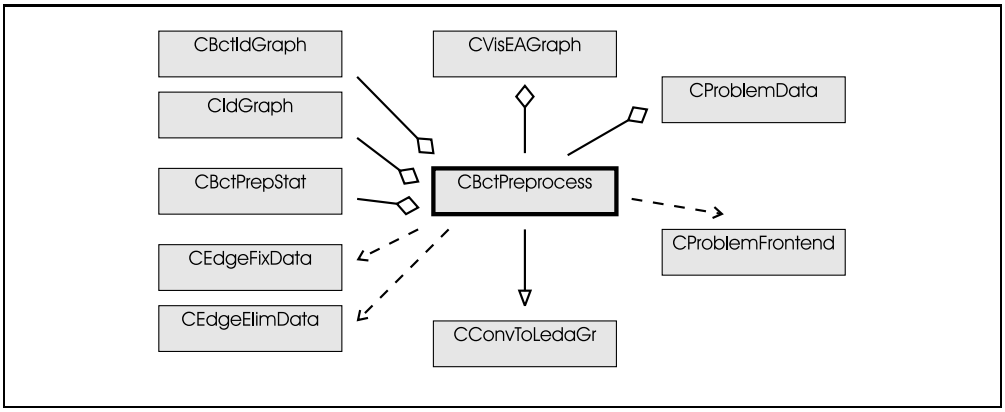


Abbildung 8.3: Diagramm für die Klasse CBctPreprocess.

CBCTPreprocess ist von CConvToLedaGr abgeleitet und erbt somit die Möglichkeit, Probleminstanzen einzulesen und die zu verwaltenden Datenstrukturen aufzubauen. Ihre Aufgabe ist, den eingelesenen Problemgraph in einen BCT umzuwandeln und die weiteren Vorverarbeitungsschritte durchzuführen. Danach wird die reduzierte Probleminstanz an den MA weitergereicht oder gegebenenfalls ein reduzierter Graph zur Verwendung für andere Algorithmen gespeichert.

Der Aufbau des BCT ist als schrittweises Umwandeln des eingelesenen Problemgraphen realisiert, wie in Kapitel 5.1.2 beschrieben. Hierdurch muss nur ein Graph im Speicher

behalten werden, und Erweiterungskanten müssen nicht explizit in einen BCT kopiert werden. Werden mehrere Knoten zu einem Block-Knoten zusammengefasst, dann werden die betroffenen Erweiterungskanten entsprechend umgelenkt, andernfalls können sie verbleiben und stellen bereits die auf den BCT übertragenen Erweiterungskanten dar. Falls eine duale Repräsentation von Graph und BCT gewünscht wird, so ist dies durch Setzen eines Parameters möglich. Die Zuordnung der Knoten zwischen dem ursprünglichen Graph und dem BCT wird in einem Hash-Array gespeichert, damit eine spätere Rückführung der vom MA erzeugten Ergebnisse auf den Problemgraphen möglich ist.

Für das Umwandeln des Graphen in einen BCT werden neben der Umstrukturierungsroutine mehrere Hilfsmethoden verwendet. Diese generieren benötigte Informationen wie zum Beispiel den Pfad zwischen zwei Knoten oder die Kennzahl einer Tiefensuche für jeden Knoten. Hierdurch wird das effiziente Auffinden des kleinsten gemeinsamen Vorgängerknotens von zwei beliebigen Knoten des BCT gewährleistet. Die weiteren Algorithmen zur Vorverarbeitung – das Eliminieren von Kanten, das Fixieren von Kanten und das Verkleinern des BCT – sind als jeweils eigenständige Methoden der Klasse realisiert. Ein getrenntes Betrachten der Effektivität der einzelnen Schritte ist somit möglich.

Weitere für die Vorverarbeitung benötigte Klassen sind CEdgeElimData und CEdgeFixData. Beide werden temporär angelegt und für das Eliminieren bzw. Fixieren von Kanten benötigt.

Die Klasse CBctPrepStat speichert Statistiken über die Ergebnisse der Vorverarbeitung.

Memetischer Algorithmus

Die von der EALIB bereitgestellte Klasse steadyStateEA verwaltet die Population der Lösungskandidaten des Algorithmus und stellt die grundlegenden Mechanismen zum Ablauf eines Steady-State Evolutionären Algorithmus zur Verfügung. Eine genaue Beschreibung dieser Klasse ist in der Beschreibung zur EALIB zu finden.

CCBEChrom ist von der virtuellen Basisklasse chromosome abgeleitet. Jede Instanz repräsentiert einen Lösungskandidaten des Algorithmus, dessen Kantenmenge wahlweise

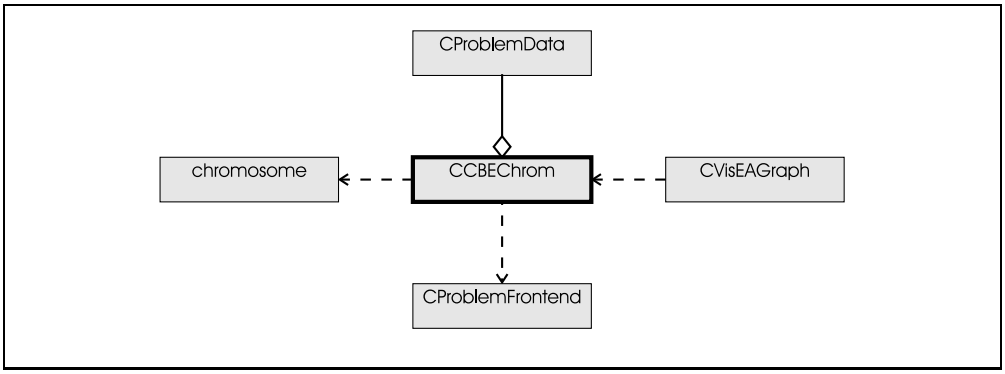


Abbildung 8.4: Diagramm für die Klasse CCBEChrom.

als Set oder Hash-Set gespeichert wird. Weiters sind neben den für die Verwaltung der Population notwendigen Vergleichs- und Zuweisungsmethoden die in Kapitel 6.2.4 beschriebenen Methoden des MA in dieser Klasse enthalten. Jeder Operator des MA ist somit als Methode von CCBEChrom implementiert, übergebene Parameter wählen unter den vorhandenen Varianten.

Ein essenzieller Teil des MA ist die Klasse CProblemData. Sie wird einmalig vor Beginn des MA für die geladenen Testgraphen konfiguriert und führt danach die Lokale Optimierung von Lösungskandidaten aus. Als statische Memberklasse von CCBEChrom wird sie bei Bedarf für den aktuell betrachteten Lösungskandidaten initialisiert und entfernt, wie in Kapitel 6.2 beschrieben, redundante Kanten. Für diesen zentralen Teil des Programms ist Effizienz besonders entscheidend. Hierauf wurde sowohl bei der Wahl der verwendeten Datenstrukturen als auch bei der Konzeption der Klasse geachtet. So wurde zum Beispiel sichergestellt, dass oft benötigte Initialisierungsvorgänge durch Speicherbereichskopien ausgeführt werden anstatt durch Einzelwertzuweisungen, oder dass auf benötigte Daten nach Möglichkeit direkt zugegriffen werden kann, ohne eine Hash-Funktion zu verwenden.

Visualisierung

Die Klasse CVisGraph nutzt ein von der LEDA-Programmbibliothek bereitgestelltes Graph-Window-Objekt, um einen CIdGraph zu veranschaulichen. Über die implemen-

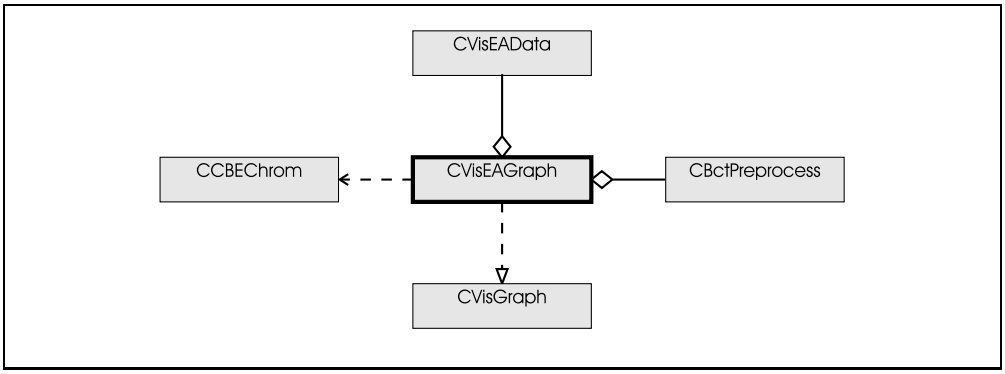


Abbildung 8.5: Diagramm für die Klasse CVisEAGraph.

tierten Methoden lassen sich Farbe und Form der einzelnen Knoten bestimmen. Weiters ist es möglich, die darzustellenden Kantenmengen zu bestimmen und diese farblich zu kennzeichnen. Die Beschriftung der abgebildeten Knoten und Kanten ist ebenfalls vorgesehen, ist jedoch nur für Instanzen mit wenigen Elementen sinnvoll.

CvisEAGraph ist eine von CVisGraph abgeleitete Klasse und dient der Darstellung von in der Population enthaltenen Lösungskandidaten im BCT. Die Lösungskandidaten können direkt an die Klasse übergeben und farblich hervorgehoben werden.

Zur Ausgabe von Mitteilungstexten wurde CMsgWin implementiert. Diese Klasse ermöglicht, kurze Nachrichten in einem Grafikfenster anzuzeigen, und wird für Demonstrationszwecke benötigt.

8.2.2 Dateien und Dateiformate

Problemgraphen

Probleminstanzen sind in Form von Textdateien gespeichert und durch die Endung „.aug“ gekennzeichnet. In der ersten Zeile der Datei wird die Knotenanzahl des Graphen spezifiziert. Danach folgen die Kanten, jeweils eine pro Zeile, im Format: `<Typ> <Startknoten> <Endknoten> <Kosten>`. `<Typ>` spezifiziert, ob die Kante aus der Menge E_0 stammt (Wert `1`) oder eine Erweiterungskante ist (Wert `0`). `<Startknoten>` und

`<Endknoten>` geben die Nummer der Knoten an, die diese Kante verbindet. Die Numerierung der Knoten beginnt bei „0“. Die Datei endet mit der Schlussmarke `0 0 0 0`.

Archivierung der Ergebnisse

`vbca` unterstützt mehrere Methoden, Ergebnisse zu archivieren. Es folgt eine kurze Übersicht.

.out Dies ist eine ausführliche und für Menschen leicht zu lesende Form, Ergebnisse eines Testlaufs festzuhalten. Parameter werden mit Namen und Wert dargestellt, Überschriften gliedern die verschiedenen Bereiche. Ein Parameter zur Laufzeit des Programmes bestimmt über den Detailgrad des Inhalts.

.sol Dieses Format ist für die Speicherung der besten gefundenen Lösung vorgesehen. Die Ausgabe erfolgt in einer vorgegebenen Form. Zuerst werden die für die gefundene Lösung gewählten Erweiterungskanten in dem Format ausgegeben, in dem auch die Probleminstanzen gespeichert sind, danach folgen weitere Daten – bei diesem Algorithmus sind es die Ergebnisse der Vorverarbeitungsschritte und, falls gewünscht, auch die von der Vorverarbeitung entfernten Kanten.

.log Diese Dateien protokollieren die Entwicklung der Population in Bezug auf ihre Bewertung über den Ablauf des MA. Die Bewertungen des besten und schlechtesten Individuums zum Zeitpunkt t sowie die durchschnittliche Bewertung der Population und die daraus resultierende Standardabweichung werden festgehalten.

.dat Die Ergebnisse mehrerer Läufe werden in je einer Tabelle pro Instanz zusammengefasst. Jeder Testlauf erweitert die Tabelle um eine Zeile und vermerkt den gefundenen Lösungswert zusammen mit weiteren Eckdaten und verwendeten Parametern zur späteren Auswertung der Ergebnisse. Die Tabellenspalten sind durch Tabulatoren getrennt.

8.2.3 Erläuterung der Optionen

Schnellhilfe

Um eine Übersicht der Parameter aufzulisten, kann man `vbca` oder `vbca -h` in der Kommandozeile eingeben. Die erste Variante listet die Parameter des Programmes sortiert nach Kategorien, die zweite liefert eine alphabetische Ordnung.

Generelle I/O Optionen

`idir` bestimmt das Verzeichnis, von dem Probleminstanzen eingelesen werden sollen. Als Standardwert ist das aktuelle Verzeichnis gesetzt.
Beispiel: `idir problemgraphen/komplex`

`ifile` setzt den Dateinamen für den einzulesenden Problemgraphen.
Beispiel: `ifile a280.aug`

`odir` bestimmt das Verzeichnis, in dem gegliederte Ergebnisdateien (.out) und Protokolldateien (.log) gespeichert werden. Als Standardwert ist das aktuelle Verzeichnis gesetzt.
Beispiel: `odir results/firstrun`

`oname` bestimmt den Namen, unter dem Ergebnisse gespeichert werden sollen. Der angegebene Name wird als Basis verwendet und durch das Anfügen der Endungen für die verschiedenen Dateien ergänzt. Wird als Name @ übergeben, so erfolgt die Ausgabe der Ergebnisse in der Konsole, genauer gesagt über die gesetzte Standardausgabe.
Beispiel: `oname a280run1`

`outext` gibt die Erweiterung an, die Resultatsdateien in ausführlicher Form kennzeichnet. Diese Erweiterung wird an den mit `ofile` spezifizierten Dateinamen angehängt. Als Standardwert wird „.out“ verwendet.
Beispiel: `outext .o`

`logext` gibt die Erweiterung an, die Protokolldateien kennzeichnet. Diese Erweiterung wird an den mit `ofile` spezifizierten Dateinamen angehängt. Als

Standardwert wird „.log“ verwendet.
Beispiel: `logext .logfile`

`soldir` bestimmt das Verzeichnis, in dem Dateien mit der besten gefundene Lösung gespeichert werden sollen. Als Standardwert wird „sol“ im aktuellen Verzeichnis verwendet.
Beispiel: `soldir results/besteLösung`

`solfname` bestimmt den Namen für die Datei mit der besten gefundenen Lösung. Als Standard wird der Name des Problemgraphen verwendet.
Beispiel: `solfname a280test`

Spezielle I/O Optionen

`detailedsol` legt den Detailgrad fest, mit dem die beste gefundene Lösung (.sol) eines Testlaufes gespeichert wird. Als Standard wird Option 1 verwendet. Gültiger Wertebereich: $[0, 1]$

- `0` Es werden die Kanten der ermittelten Lösung zusammen mit den Eckdaten für die Vorverarbeitungsschritte dieser Instanz gespeichert.
- `1` Wie oben, jedoch werden detaillierte Listen der durch die Vorverarbeitung entfernten beziehungsweise fixierten Kanten hinzugefügt.

`lchonly` legt fest, ob und wie Einträge für die Protokolldatei (.log) erstellt werden. Als Standard wird Option 2 verwendet. Gültiger Wertebereich: $[0, 2]$.

- `0` Es werden nur mit der in Option `lfreq` festgelegten Häufigkeit Einträge in die Protokolldatei vorgenommen.
- `1` Es werden nur dann Einträge mit der in `lfreq` angegebenen Häufigkeit in die Protokolldatei vorgenommen, wenn sich der Wert der besten gefundenen Lösung verbessert hat.
- `2` Es werden zusätzlich zu den durch `lfreq` festgelegten Einträgen all jene Generationen aufgenommen, in denen sich der Wert der besten gefundenen Lösung verbessert hat.

`lbuffer` bestimmt die Puffergröße für die Einträge der Protokolldatei (.log). Als Standard ist eine Größe von 10 Einträgen gewählt. Gültiger Wertebereich: $[1, 10000000]$.

`lfreq` bestimmt die Häufigkeit, mit der Einträge für die Protokolldatei (.log) erstellt werden. Als Standard ist die spezielle Größe von -1 festgelegt. Gültiger Wertebereich: $[-1, 10000000]$. Spezielle Werte:

- `-1` Einträge werden für die Generationen $0, 1, 2, 5, 10, 20, 50, 100 \ldots$ erstellt.
- `0` Es werden keine Einträge erstellt.

`logdupelim` bestimmt, ob die Anzahl an eliminierten Duplikaten, sofern diese gelöscht werden, in der Protokolldatei gespeichert werden soll. Als Standard ist Option 0 festgelegt. Gültiger Wertebereich: [0,1].

- `0` Die Anzahl eliminierter Duplikate wird gespeichert.
- `1` Die Anzahl eliminierter Duplikate wird nicht gespeichert.

`nformat` legt das Format fest, in dem Kommazahlen dargestellt werden. Die Syntax entspricht der Formatanweisung für die in der Programmiersprache C verwendete Funktion printf.

Optionen für den Batchbetrieb

`batchmode` bestimmt, ob Hilfestellungen für den Batchbetrieb des Programms aktiviert werden sollen. Als Standard wird Option 0 verwendet. Gültiger Wertebereich: [0,2].

- `0` Die Option ist deaktiviert.
- `1` Dateinamen für die Ausgabe werden automatisch generiert, indem der Name der Probleminstanz um Datum und Uhrzeit erweitert wird. Hierdurch wird auch eine Einzigartigkeit der Namen sichergestellt, solange nicht zwei Testläufe innerhalb einer Sekunde beendet werden.
- `2` Wie 1, jedoch werden sämtliche Ausgaben außer einem Eintrag in die Tabellendatei (.dat) deaktiviert.

`datline` bestimmt, ob ein Tabelleneintrag für diesen Testlauf vorgenommen werden soll. Falls keine Tabelle im Verzeichnis `odir` für diese Instanz existiert, wird eine neue angelegt. Als Standard ist Option 1 gewählt. Gültiger Wertebereich: [0,1].

- `0` Es wird kein Tabelleneintrag vorgenommen.
- `1` Ein Tabelleneintrag wird generiert.

`txtgr` bestimmt, ob ein durch das Programm bearbeiteter Problemgraph gespeichert werden soll. Abhängig vom Parameter `mode` ist dies entweder ein durch die Vorverarbeitungsschritte in Bezug auf die enthaltenen Kanten reduzierter Graph („.reduced“ bei `mode = 3`), oder G_0 zusätzlich der für die Lösung ermittelten Kanten („.best“ bei `mode = 0,1,2`). Als Standard ist Option 0 gewählt. Gültiger Wertebereich: [0,2].

- `0` Es wird kein Graph gespeichert.
- `1` Es wird ein Graph im gleichen Format wie die Probleminstanzen gespeichert.
- `2` Es wird ein Graph gespeichert, der die gleiche Art der Formatierung wie die Probleminstanzen verwendet, jedoch Kanten die von der Vorverarbeitung für eine optimale Lösung fixiert wurden, als Typ 2 spezifiziert.

Spezifische Optionen für den MA

`cdag` bestimmt, ob erzeugte Duplikate als Generation gezählt werden oder nicht. Als Standard ist Option 0 festgelegt. Gültiger Wertebereich: [0,1].

- `0` Duplikate werden nicht als Generation gewertet.
- `1` Duplikate werden als Generation gewertet.

`dupelim` bestimmt, ob Duplikate eliminiert werden. Als Standard ist Option 1 festgelegt. Gültiger Wertebereich [0,2].

- `0` Duplikate werden nicht eliminiert.

1 Durch die Rekombination erzeugte Duplikate werden eliminiert.

2 Durch die Initialisierung und Rekombination erzeugte Duplikate werden eliminiert.

`usewheap` bestimmt, ob ein Heap zum Auffinden des jeweils schlechtest bewerteten Individuums aufgebaut werden soll. Als Standard ist Option 1 gewählt. Gültiger Wertebereich: [0,1].

0 Der Heap wird nicht aufgebaut und verwaltet.

1 Der Heap wird verwendet.

`seed` bestimmt die Zahl, mit der der Zufallszahlengenerator initialisiert wird. Als Standard ist der spezielle Wert 0 festgelegt. Gültiger Wertebereich: [0,maxlong].

0 Dem Zufallszahlengenerator wird kein spezieller Wert übergeben, die Initialisierung erfolgt über eine Systemfunktion.

Generelle Strategieparameter

`popsize` bestimmt die Größe der Population für den MA. Als Standard ist eine Größe von 500 Individuen festgelegt. Gültiger Wertebereich: [2,10000000].

`pmut` bestimmt die Wahrscheinlichkeit, mit der ein durch die Rekombination erzeugtes Individuum mutiert wird. Negative Werte werden als Mutationswahrscheinlichkeit gewertet und es wird pro Individuum zufällig entschieden. Positive Werte zählen als Mutationsrate. Als Standard ist ein Wert von -0.7 gewählt. Gültiger Wertebereich: [-1.0,1.0].

`pcross` bestimmt die Wahrscheinlichkeit, mit der ein neues Individuum durch die Rekombination erzeugt wird. Als Standard ist ein Wert von 1.0 gewählt. Gültiger Wertebereich: [0.0,1.0].

`repl` bestimmt die Strategie, mit der bei einem Generationsschritt ein Individuum ersetzt wird. Die Möglichkeiten umfassen eine zufällige, deterministische oder turnierbasierte Auswahl. Als Standard ist Option 1 festgelegt. Gültiger Wertebereich: [-1000,1].

`-n` Ein Turnier mit n Teilnehmern wird durchgeführt, um das zu löschende Individuum zu bestimmen.

`0` Ein zufälliges Individuum wird gewählt und ersetzt.

`1` Das schlechtest bewertete Individuum wird ersetzt.

`tselk` bestimmt die Gruppengröße für die Selektion von Individuen. Als Standard ist ein Wert von 3 festgelegt. Gültiger Wertebereich: [1,10000].

`tcond` bestimmt das Abbruchkriterium für den MA. Als Standard ist Option 1 gewählt. Gültiger Wertebereich: [0,2].

`0` Der MA wird nach einer durch den Parameter `tgen` bestimmten Anzahl an Generationen beendet.

`1` Der MA wird beendet, wenn innerhalb der durch `tgen` angegebenen Generationsschritte keine Verbesserung des derzeit besten Individuums erreicht wird.

`2` Der MA wird beendet, wenn ein durch `tobj` spezifiziertes Ergebnis erreicht wird.

`tgen` bestimmt abhängig vom Parameter `tcond` einen Wert für das Abbruchkriterium des MA. Als Standard ist ein Wert von 10000 festgelegt. Gültiger Wertebereich: [0,100000000].

`tobj` bestimmt einen Zielwert, bei dem der MA abbricht, wenn der Parameter `tcond` auf 2 gesetzt wird. Gültiger Wertebereich: [double].

MA Methoden und Strategieparameter

`dev` bestimmt die Standardabweichung für normalverteilte Zufallszahlen außerhalb der Initialisierung. Als Standard ist ein Wert von 0.5 festgelegt.

`idev` bestimmt die Standardabweichung für normalverteilte Zufallszahlen während der Initialisierung. Als Standard ist ein Wert von 0.5 festgelegt.

`preproc` bestimmt, welche Schritte der Vorverarbeitung durchgeführt werden sollen. Als Standard ist Option 3 gewählt. Gültiger Wertebereich [0,3].

`0` Außer dem Aufbau des BCT werden keine weiteren Schritte vorgenommen.

1 Nach dem Aufbau des BCT wird nach weiteren Möglichkeiten zur Verkleinerung durch die Kantenfixierung gesucht.

2 Nach dem Aufbau des BCT sucht die Kantenelimination nach redundanten Kanten.

3 Alle Schritte zur Reduktion der Komplexität der Problemstellung werden verwendet.

`heur` bestimmt, welche Heuristik verwendet werden soll. Als Standard ist Option 1 gewählt. Gültiger Wertebereich: [0,1].

0 Die Kosten der Kanten werden als heuristisches Maß in den Methoden des MA verwendet.

1 Die Kosten pro Cut-Knoten werden verwendet.

`init` bestimmt die Methode, die bei der Initialisierung von Individuen verwendet werden soll. Als Standard ist Variante 5 festgelegt. Gültiger Wertebereich: [0,5].

0 Verwendet ausgehend von den noch nicht abgedeckten Cut-Knoten dieselbe Strategie, die für den Parameter `covcbe` festgelegt wurde.

1 Verwendet ausgehend von den noch nicht abgedeckten Cut-Knoten eine invers proportionale Auswahl zwischen je 2 zufällig gewählten Kandidaten.

2 Verwendet ausgehend von den noch nicht abgedeckten Cut-Knoten binäre Turniere, um Kanten zu selektieren.

3 Verwendet eine normalverteilte rangbasierte Auswahl über alle Kanten.

4 Verwendet eine normalverteilte Auswahl über die Reihung der Kanten nach dem aktuellen Nutzen.

5 Verwendet eine normalverteilte heuristisch getrimmte Auswahl über alle Kanten.

`covcbe` bestimmt die Vorgangsweise bei der Auswahl zum Auffüllen von Individuen, nachdem eine Kante durch die Mutation gelöscht wurde. Als

Standard ist Variante 2 festgelegt. Gültiger Wertebereich: [0,2].

- `0` Bearbeite die nicht abgedeckten Cut-Komponenten in der Reihenfolge, in der sie entdeckt werden, und wähle zufällig aus der Menge der Kandidaten.
- `1` Bearbeite die nicht abgedeckten Cut-Komponenten in explizit zufälliger Reihenfolge und wähle zufällig aus der Menge der Kandidaten.
- `2` Bearbeite die nicht abgedeckten Cut-Komponenten in explizit zufälliger Reihenfolge und wähle normalverteilt über die Rangordnung der sortierten Kandidaten.

`remred` bestimmt die Vorgangsweise der lokalen Optimierung bei der Suche nach redundanten Kanten. Als Standard ist Variante 2 festgelegt. Gültiger Wertebereich: [0,2].

- `0` Die Kanten werden in der Reihenfolge überprüft, in der sie aufscheinen.
- `1` Die Kanten werden in explizit zufälliger Reihenfolge auf Redundanz überprüft.
- `2` Die Kanten werden in umgekehrter Reihenfolge ihres heuristischen Wertes überprüft.

`cross` bestimmt die Vorgangsweise der Rekombination von 2 Individuen. Als Standard wurde Variante 2 festgelegt. Gültiger Wertebereich: [0,2].

- `0` Die Elternindividuen werden zuerst vereint, danach entfernt die lokale Optimierung redundante Kanten.
- `1` Die Schnittmenge der Elternindividuen wird auf den Nachkommen übertragen. Aus der symmetrischen Differenz werden zufällig Erweiterungskanten gewählt, bis die Lösung gültig ist.
- `2` Die Schnittmenge der Elternindividuen wird auf den Nachkommen übertragen. Aus der symmetrischen Differenz werden jeweils 2 Kanten gewählt und die bessere wird dem Nachkommen hinzugefügt.

`mut` bestimmt die Vorgangsweise, mit der eine Kante bei der Mutation gelöscht wird. Als Standard wurde Variante 1 festgelegt. Gültiger Wertebereich: [0,2].

- `0` Eine zufällige Kante wird gelöscht.
- `1` Es wird invers proportional zwischen 2 Kanten entschieden.
- `2` Die schlechtere von 2 Kanten wird gelöscht.

Optionen zum Programmablauf

`mode` bestimmt den Programmablauf von `vbca`. Als Standard wurde Option 0 gewählt. Gültiger Wertebereich: [0,15].

- `0` Der Memetische Algorithmus wird ausgeführt, und Statistiken werden im vollen Umfang generiert.[2]
- `1` Der Memetische Algorithmus wird ausgeführt, und Statistiken werden in verkleinertem Umfang generiert.[2]
- `2` Es werden vier Lösungen deterministisch erzeugt, die auf einer Auswahl von Kanten nach Kosten / Kosten pro Cut-Knoten und dem aktuellen Nutzen basieren.[2]
- `3` Statistikwerte für die Vorverarbeitungsschritte werden generiert.[2]
- `4` Stellt den BCT nach den Vorverarbeitungsschritten für die angegebene Instanz grafisch dar. Erweiterungskanten werden eingezeichnet.
- `5` Wie 4, jedoch ohne Erweiterungskanten.
- `6` Stellt den BCT vor den Vorverarbeitungsschritten inklusive aller Erweiterungskanten grafisch dar.
- `7` Wie 6, jedoch ohne Erweiterungskanten.
- `8` Zeigt drei Beispiele für die Initialisierung von Individuen.[3]
- `9` Wie 8, jedoch werden die Kanten zusätzlich nummeriert.[3]

[2] Siehe auch Parameter `txtgr` und `datline`.

[3] Die Verwendung eines kleinen Graphen ist empfohlen, aber nicht Bedingung.

`10` Zeigt drei Beispiele für die Mutation.[3]

`11` Wie 10, jedoch werden die Kanten zusätzlich nummeriert.[3]

`12` Zeigt drei Beispiele für die Rekombination.[3]

`13` Wie 12, jedoch werden die Kanten zusätzlich nummeriert.[3]

`14` Generiert zusätzliche Statistiken während der Vorverarbeitungsschritte.

`15` Wie 14 und erstellt ein logfile davon.

Debug-Optionen

Diese Parameter schalten Teile des Programms ab oder „missbrauchen“ Funktionen, um einen alternativen Weg zur Überprüfung von Ergebnissen des Programms `vbcutp` bereitzustellen. Es wird nicht empfohlen, einen dieser Parameter zu ändern, bevor man sich nicht mit dem dazugehörigen Source-Code vertraut gemacht hat.

`dbnoprep` schaltet die Vorverarbeitung ab.

`dbvalid` stellt einen sehr teuren Weg dar, eine Lösung zu validieren.

`dblocopt` stellt einen noch viel teureren Weg dar, die lokale Optimalität einer Lösung zu überprüfen.

`useCP` schaltet die Ausgabe über die Klasse CProblem ab.

8.3 vbcutp

`vbcutp` ermittelt die Anzahl von Cut-Knoten aufgeschlüsselt nach Knotengraden für einen Graphen im Problemgraphformat. Eine weitere Anwendung ist die Überprüfung von Probleminstanzen auf ihre Lösbarkeit und von ermittelten Lösungen auf ihre Gültigkeit. Das Programm baut für die eingelesene Problemstellung eine Adjazenzliste auf und verwendet nachfolgend eine Tiefensuche, um die Daten zu generieren. Für die Adjazenzliste werden jene Kanten der Probleminstanz verwendet, deren Eintrag für `<Typ>` innerhalb des durch [`etyfrom`, `etyto`] festgelegten Bereiches liegt. Siehe auch Abschnitt 8.2.2, in dem das Format der gespeicherten Graphen erläutert wird.

`etyfrom` Kanten, deren `<Typ>`-Eintrag im Bereich [`etyfrom`, `etyto`] liegt, werden eingelesen. Als Standard ist ein Wert von 0 gewählt. Gültiger Wertebereich: [integer].

`etyto` Kanten, deren `<Typ>`-Eintrag im Bereich [`etyfrom`, `etyto`] liegt, werden eingelesen. Als Standard ist ein Wert von 2 gewählt. Gültiger Wertebereich: [integer].

`ifile` setzt den Dateinamen für den einzulesenden Problemgraphen.
Beispiel: `ifile R400.aug`

`nodedeg` bestimmt, ob eine Liste der Knotengrade ausgegeben werden soll. Als Standard ist Option 0 gewählt. Gültiger Wertebereich: [0,1].

- `0` Es wird keine Liste erstellt.
- `1` Erstellt eine Liste der Knotengrade.

`obs` bestimmt, ob nach redundanten Kanten für die Problemstellung des zweifachen Zusammenhanges gesucht werden soll. Dies diente in erster Linie zur Kontrolle von ermittelten Lösungen durch das Programm `vbca` durch eine zweite verschiedene Vorgangsweise. Es wird eine einfache „leave one out“ Strategie für Kanten mit einem `<Typ>`$\neq 1$ angewandt. Als Standard ist Option 0 gewählt. Gültiger Wertebereich: [0,1].

- `0` Eine Überprüfung wird nicht vorgenommen.
- `1` Es wird nach redundanten Kanten gesucht.

`ofile` bestimmt den Dateinamen, unter dem die Ergebnisse gespeichert werden sollen. Die als Standard gewählte Option ist @. Spezielle Werte:

- `@` Die Ergebnisse werden in die Standardausgabe (Konsole) geschrieben.
- `+` Der Dateiname wird aus dem Namen des eingelesenen Graphen und der Endung „.cutp“ gebildet.

`short` bestimmt, ob die Ausgabe in gekürzter Form erfolgen soll. Als Standard ist Option 1 gewählt. Gültiger Wertebereich: [0,1].

- `0` Für gefundene Cut-Knoten wird zusätzlich ihr Name (Nummer) ausgegeben, sonst wie 1.

1 Die Ausgabe umfasst die Eckdaten für den eingelesenen Graphen, die verwendeten Kanten und die unter diesen Voraussetzungen auftretenden Cut-Knoten.

`solcheck` Kurzform, um dem Programm mitzuteilen, dass ein von `vbca` erstellter Lösungsgraph (.best) überprüft werden soll. Als Standard ist Option 0 gewählt. Gültiger Wertebereich: [0,1].

0 Deaktiviert diese Option.

1 Der eingelesene Graph wird überprüft und die Ausgabe erfolgt, sofern sie gespeichert werden soll, in eine Datei, deren Name durch das Anhängen von „.check“ an den Namen der eingelesenen Datei gebildet wird.

`startat` bestimmt den Knoten (Nummer), von dem aus die Tiefensuche starten soll. Nur die mit diesem Knoten zusammenhängende Komponente wird untersucht. Als Standard ist Knoten 0 gewählt. Gültiger Wertebereich: $[0,\ \ |V|-1]$

8.4 gr2tex

`gr2tex` erstellt eine Abbildung mit frei wählbaren Abmessungen eines Graphen im Postscript Format. Die Programmpakete LaTeX 2_ε und `dvips` werden hierzu benötigt. Beispiele der Ausgabe sind in Anhang A zu finden. Als Eingabe dient ein Graph im Problemgraphformat (.aug) und eine Datei, die die Koordinaten der Knoten enthält (.tsp). Optional kann zusätzlich eine generierte Lösung (.sol) für diesen Graphen angegeben werden.

I/O Parameter

`tspdir` bestimmt das Verzeichnis, von dem Koordinaten-Dateien (.tsp) eingelesen werden sollen. Als Standardwert ist das Verzeichnis „./tsp/“ gewählt. Dieser Parameter kann über die Datei gr2tex.config gesetzt werden, um

die Handhabung des Programmes zu vereinfachen.
Beispiel: `tspdir graphen/tsp/`

`soldir` bestimmt das Verzeichnis, von dem Lösungs-Dateien (.sol) eingelesen werden sollen. Als Standardwert ist das Verzeichnis „./sol/“ gewählt. Dieser Parameter kann über die Datei gr2tex.config gesetzt werden, um die Handhabung des Programmes zu vereinfachen.
Beispiel: `soldir lsg/`

`augdir` bestimmt das Verzeichnis, von dem aus Graphen (.aug) eingelesen werden sollen. Als Standardwert ist das Verzeichnis „./aug/“ gewählt. Dieser Parameter kann über die Datei gr2tex.config gesetzt werden, um die Handhabung des Programmes zu vereinfachen.
Beispiel: `augdir graphen/aug/`

`resdir` bestimmt das Verzeichnis, in dem die erzeugten Grafiken gespeichert werden sollen. Als Standardwert ist das aktuelle Verzeichnis gewählt. Dieser Parameter kann über die Datei gr2tex.config gesetzt werden, um die Handhabung des Programmes zu vereinfachen.
Beispiel: `resdir graphen/pspic/`

`file` setzt den Basisnamen für die einzulesenden Dateien. Dieser Name wird durch die mit den Parametern `tsp`, `sol` und `aug` spezifizierten Endungen erweitert, um die entsprechenden Dateinamen zu bilden und diese einzulesen.
Beispiel: `file a280`

`tsp` gibt die Erweiterung an, die Koordinatendateien (.tsp) kennzeichnet. Diese Erweiterung wird an den mit `file` übermittelten Namen angehängt. Als Standard wird „.tsp“ verwendet.
Beispiel: `tsp .coord`

`sol` gibt die Erweiterung an, die Lösungsdateien (.sol) kennzeichnet. Diese Erweiterung wird an den mit `file` übermittelten Namen angehängt. Als Standard wird „.sol“ verwendet.
Beispiel: `sol .solution`

`aug` gibt die Erweiterung an, die Graphendateien (.aug) kennzeichnet. Diese

Erweiterung wird an den mit `file` übermittelten Namen angehängt. Als Standard wird „.aug“ verwendet.
Beispiel: `aug .txtgraph`

`confdir` bestimmt das Verzeichnis, von dem die Konfigurationsdatei eingelesen werden soll. Als Standard ist der spezielle Parameter „-“ gewählt.
Beispiel: `confdir ../`

- Die Konfigurationsdatei wird nicht verwendet, benötigte Pfade werden über die Parameter spezifiziert.

`newconf` erstellt eine neue Konfigurationsdatei in dem mit `confdir` angegebenen Verzeichnis. Eine eventuell vorhandene Datei wird überschrieben. Die Konfigurationsdatei ist selbsterklärend. Sie kann mit einem Texteditor(z.B. kwrite) geöffnet werden. An den bezeichneten Stellen lassen sich die Pfade eintragen. Zum Erzeugen einer neuen Konfigurationsdatei wird die Option: `newconf 1` verwendet.

Grafik Parameter

`border` gibt an, wieviel Millimeter Abstand beim Zeichnen des Graphen zum Rand der Grafik gelassen werden soll. Als Standard ist ein Wert von 1 gewählt. Gültiger Wertebereich: [0,10]

`vertex` gibt den Durchmesser der gezeichneten Knoten in Millimetern an. Als Standard ist ein Wert von 1 gewählt. Gültiger Wertebereich: [0.2,5]

`labxof` gibt den horizontalen Abstand der Beschriftung des Knotens zu seinem Mittelpunkt in Millimetern an. Als Standard ist ein Wert von 1 gewählt. Gültiger Wertebereich: [0.1,5]

`labyof` gibt den vertikalen Abstand der Beschriftung des Knotens zu seinem Mittelpunkt in Millimetern an. Als Standard ist ein Wert von 0.25 gewählt. Gültiger Wertebereich: [0.1,5]

Darstellungsparameter

`ae` bestimmt die Art der Darstellung von Erweiterungskanten. Als Standard ist Option 0 gewählt. Gültiger Wertebereich: [-1,3]

- `-1` Die Erweiterungskanten werden als punktierte Linien eingezeichnet.
- `0` Die Erweiterungskanten werden nicht eingezeichnet.
- `1` Die Erweiterungskanten werden als dünne Linien eingezeichnet.
- `2` Die Erweiterungskanten werden als stärkere Linien eingezeichnet.
- `3` Die Erweiterungskanten werden als dicke Linien eingezeichnet.

`ge` bestimmt die Art der Darstellung von Kanten des zu erweiternden Graphen. Als Standard ist Option 1 gewählt. Gültiger Wertebereich: [-1,3]

- `-1` Die Kanten des Graphen werden als punktierte Linien eingezeichnet.
- `0` Die Kanten des Graphen werden nicht eingezeichnet.
- `1` Die Kanten des Graphen werden als dünne Linien eingezeichnet.
- `2` Die Kanten des Graphen werden als stärkere Linien eingezeichnet.
- `3` Die Kanten des Graphen werden als dicke Linien eingezeichnet.

`se` bestimmt die Art der Darstellung von Erweiterungskanten, die zur Lösung des Problems ausgewählt wurden. Als Standard ist Option 3 gewählt. Gültiger Wertebereich: [-1,3]

- `-1` Die zur Lösung verwendeten Erweiterungskanten werden als punktierte Linien eingezeichnet.
- `0` Die zur Lösung verwendeten Erweiterungskanten werden nicht eingezeichnet.

1 Die zur Lösung verwendeten Erweiterungskanten werden als dünne Linien eingezeichnet.

2 Die zur Lösung verwendeten Erweiterungskanten werden als stärkere Linien eingezeichnet.

3 Die zur Lösung verwendeten Erweiterungskanten werden als dicke Linien eingezeichnet.

`n` bestimmt die Art der Darstellung von Knoten. Als Standard ist Option -1 gewählt. Gültiger Wertebereich: [-1,1]

`-1` Die Knoten des Graphen werden ohne Beschriftung eingezeichnet.

`0` Die Knoten des Graphen werden nicht eingezeichnet.

`1` Die Knoten des Graphen werden beschriftet eingezeichnet.

`w` bestimmt die Breite der erzeugten Grafik in Millimetern. Als Standard ist ein Wert von 100 gewählt. Spezielle Werte:

`0` Die Breite der Grafik ergibt sich proportional zur gewählten Höhe.

`h` bestimmt die Höhe der erzeugten Grafik in Millimetern. Als Standard ist der spezielle Wert von 0 gewählt. Spezielle Werte:

`0` Die Höhe der Grafik ergibt sich proportional zur gewählten Breite.

`ps` bestimmt, in welchem Format die generierte Grafik gespeichert werden soll. Als Standard ist Option 1 gewählt. Gültiger Wertebereich: [0,1]

`0` Die generierte Grafik wird als LaTeX 2_ε (.tex) Datei gespeichert.

`1` Die generierte Grafik wird im Postscript-Format gespeichert.

Kapitel 9

Schlusswort

In dieser Diplomarbeit wurde ein Memetischer Algorithmus zur Lösung des Vertex-Biconnectivity-Augmentation-Problems entwickelt. Die Hauptmerkmale des Algorithmus sind seine effektive, deterministische Vorverarbeitung der Probleminstanzen, durch die der Suchraum in vielen Fällen beträchtlich reduziert werden konnte, die lokale Optimierungsmethode, die garantiert, dass nur lokal optimale Lösungen in der Population gespeichert und weiterverarbeitet werden, und die strenge Vererbung und Lokalität bei der verwendeten Rekombination bzw. Mutation.

Weiters spielen die eingebundenen kostenbasierten Heuristiken zur Selektion von Kanten bei der Initialisierung, Rekombination und Mutation eine wichtige Rolle und steigern die Effizienz des Verfahrens. Empirische Tests deuten darauf hin, dass der Algorithmus zuverlässig Lösungen von hoher Qualität findet, die in vielen Fällen optimal sind.

Das Verfahren skaliert gut mit der Komplexität der Probleminstanzen, was unter anderem auf die effiziente Repräsentation und Handhabe der Lösungen, und auf den relativ geringen erforderlichen Aufwand zur Überprüfung einer Lösung auf Optimalität zurückzuführen ist.

Anhang A

Die Problemgraphen

Um einen Eindruck von Struktur und Beschaffenheit der verschiedenen Probleminstanzen zu bekommen, folgen hier Darstellungen der verwendeten Graphen. Dieser Anhang gliedert sich in zwei Abschnitte. Im ersten werden die zufällig generierten Testinstanzen von Zhu gezeigt. Die Wiedergabe erfolgte mit dem Programm `vbca`, das Layout des Graphen wurde durch einen Spring-Embedding-Algorithmus bestimmt. Artikulationsknoten werden als helle Quadrate markiert. Im zweiten Abschnitt werden die euklidischen Graphen der TSPLIB visualisiert. Die Ausgabe erfolgte mit dem Programm `gr2tex` und stellt die Instanzen maßstabsgetreu dar.

Für jede Problemkategorie wird je ein Graph veranschaulicht, die Darstellung umfasst den zu erweiternden Graphen $G_0 = (V, E_0)$. Die Erweiterungskanten $E_a = E \setminus E_0$ sind der Übersichtlichkeit halber nicht eingezeichnet.

Liste der dargestellten Graphen

A.1 Darstellungen der Testinstanzen von Zhu

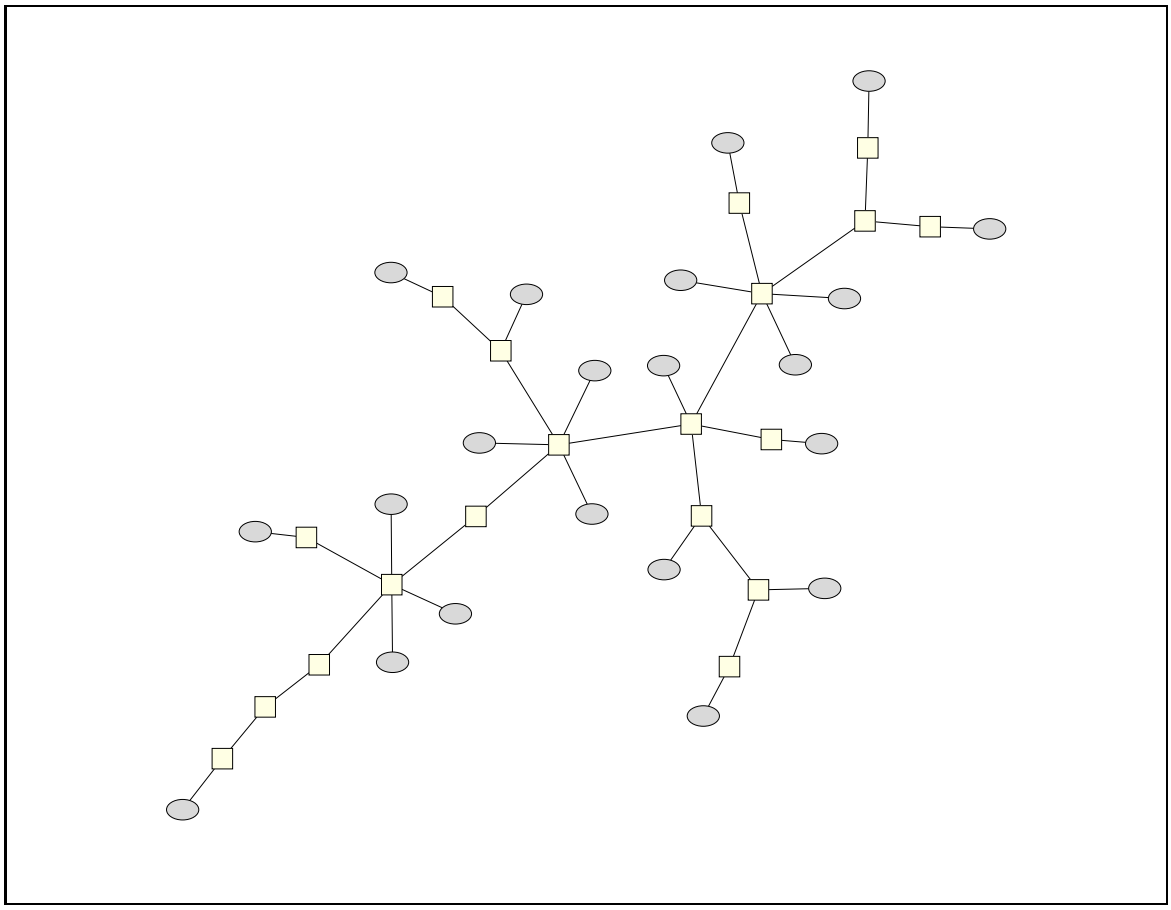


Abbildung A.1: Die Struktur des Graphen G_0 für die Probleminstanz A5. A5 besteht aus 40 Knoten und insgesamt 79 Kanten. 19 der Knoten sind Cut-Knoten. Die Gewichtung der Erweiterungskanten ist ganzzahlig aus dem Bereich $[1, 780]$ gewählt.

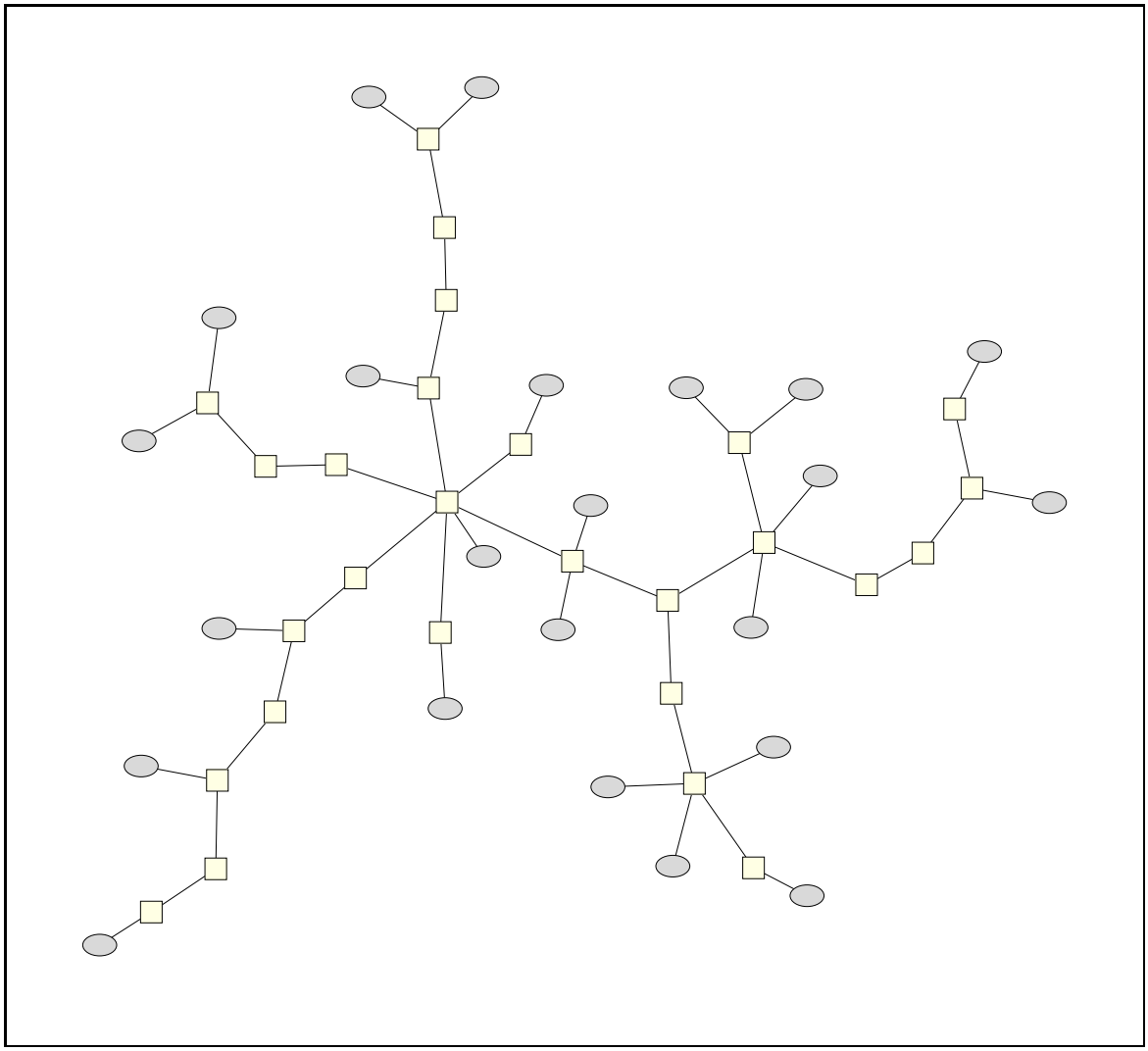


Abbildung A.2: Die Abbildung zeigt G_0 für die Probleminstanz B4.
B4 besteht aus 50 Knoten und insgesamt 112 Kanten. 27 der Knoten sind Cut-Knoten. Die Gewichtung der Erweiterungskanten ist ganzzahlig aus dem Bereich $[1, 1225]$ gewählt.

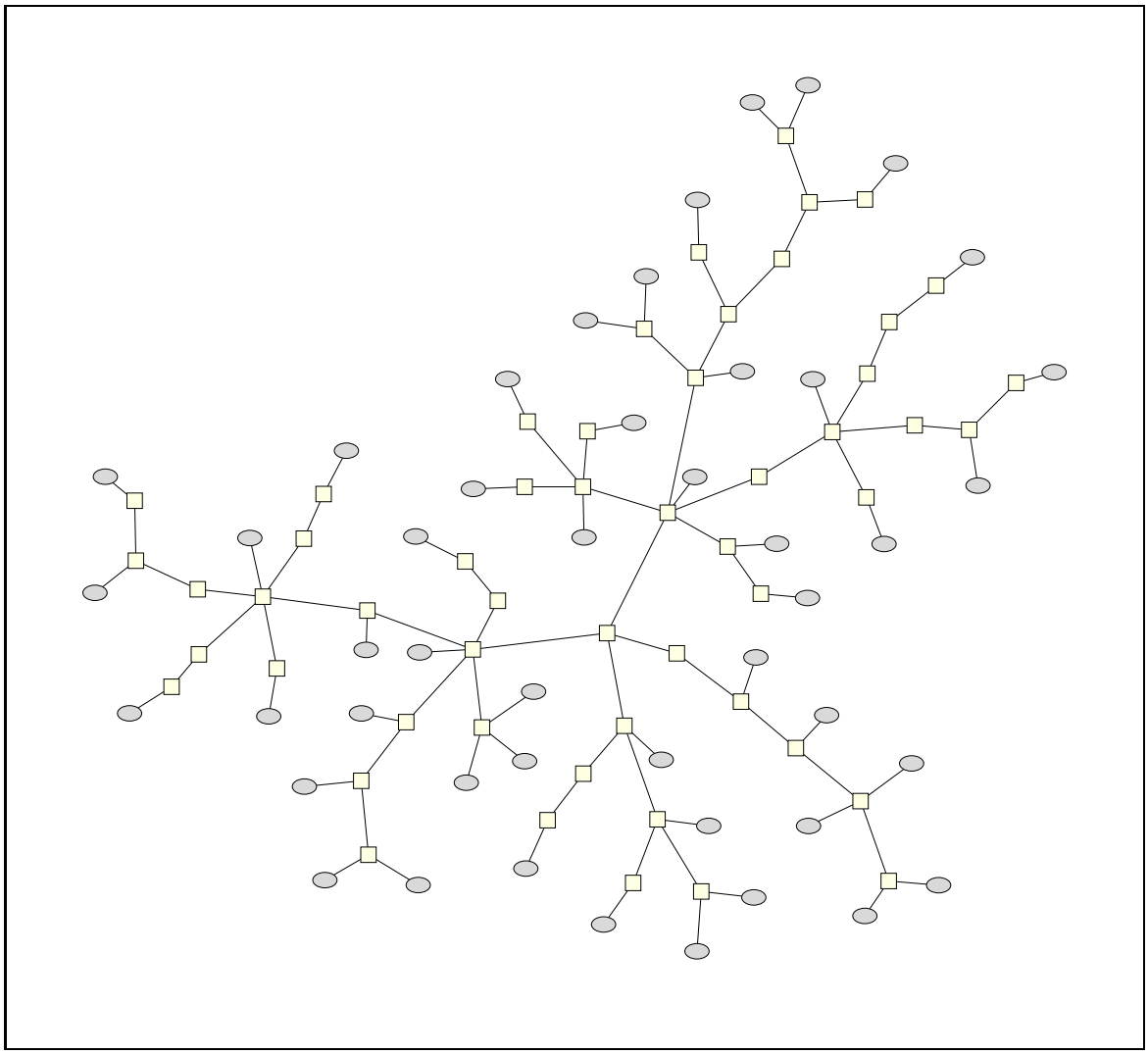


Abbildung A.3: Die Abbildung zeigt G_0 für die Probleminstanz C3.
C3 besteht aus 100 Knoten und insgesamt 248 Kanten. 53 der Knoten sind Cut-Knoten. Die Gewichtung der Erweiterungskanten ist ganzzahlig aus dem Bereich $[1, 4950]$ gewählt.

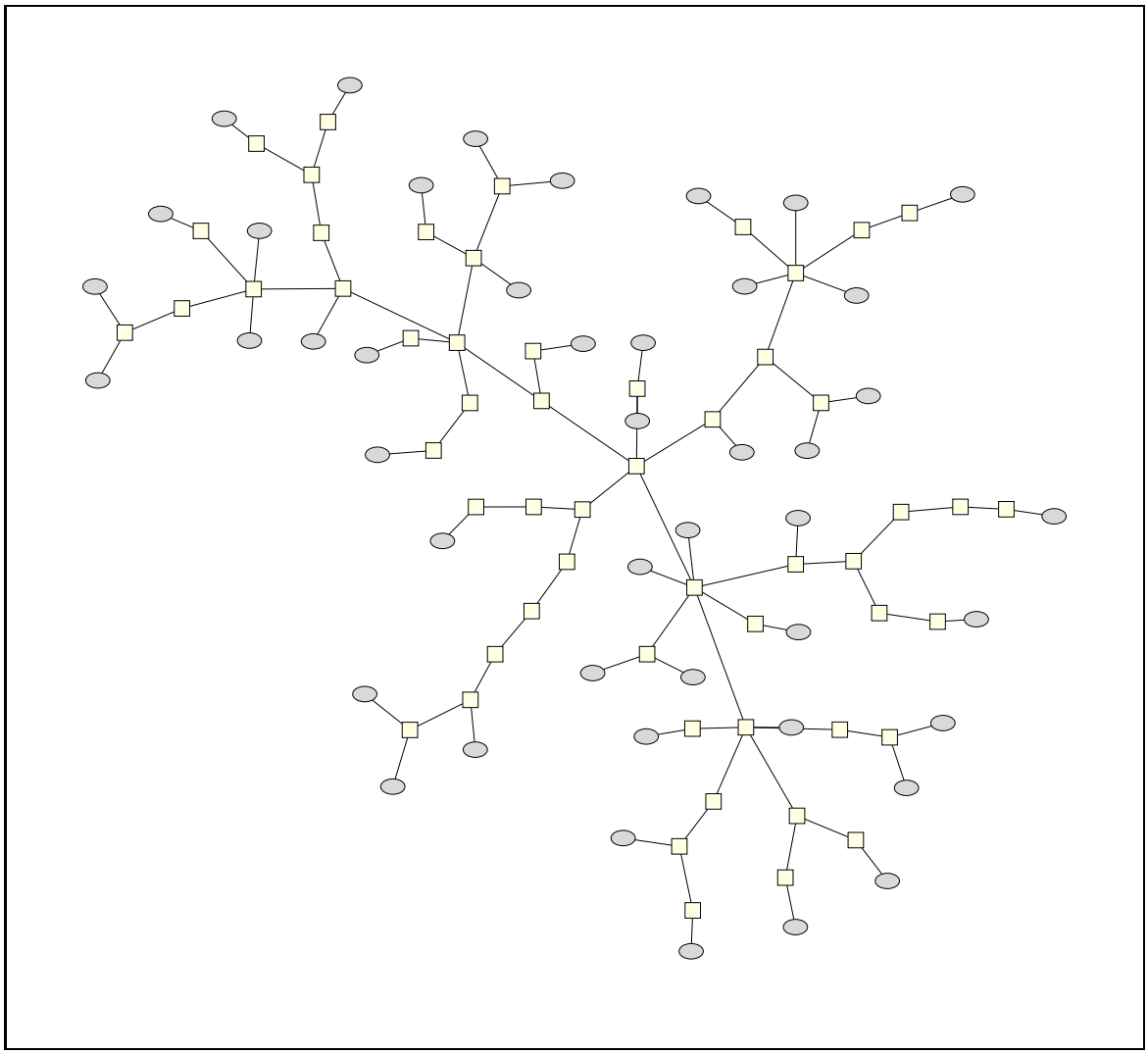


Abbildung A.4: Die Abbildung zeigt G_0 für die Probleminstanz D5.
D5 besteht aus 100 Knoten und insgesamt 497 Kanten. 55 der Knoten sind Cut-Knoten. Die Gewichtung der Erweiterungskanten ist ganzzahlig aus dem Bereich $[1, 4950]$ gewählt.

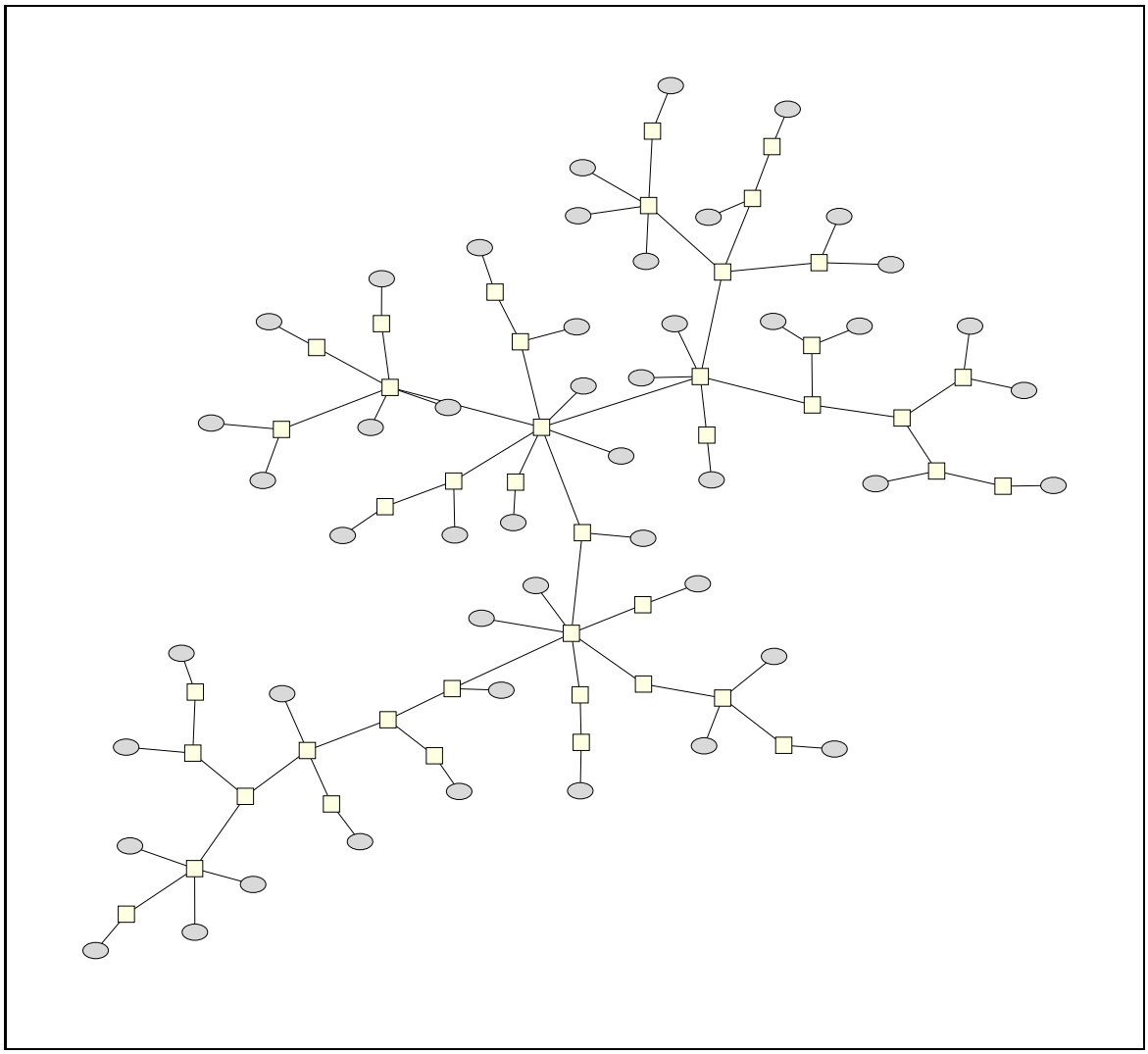


Abbildung A.5: Die Abbildung zeigt G_0 für die Probleminstanz M3.
M3 besteht aus 90 Knoten und insgesamt 438 Kanten. 42 der Knoten sind Cut-Knoten. Die Gewichtung der Erweiterungskanten ist ganzzahlig aus aus dem Bereich $[10, 1000]$ gewählt.

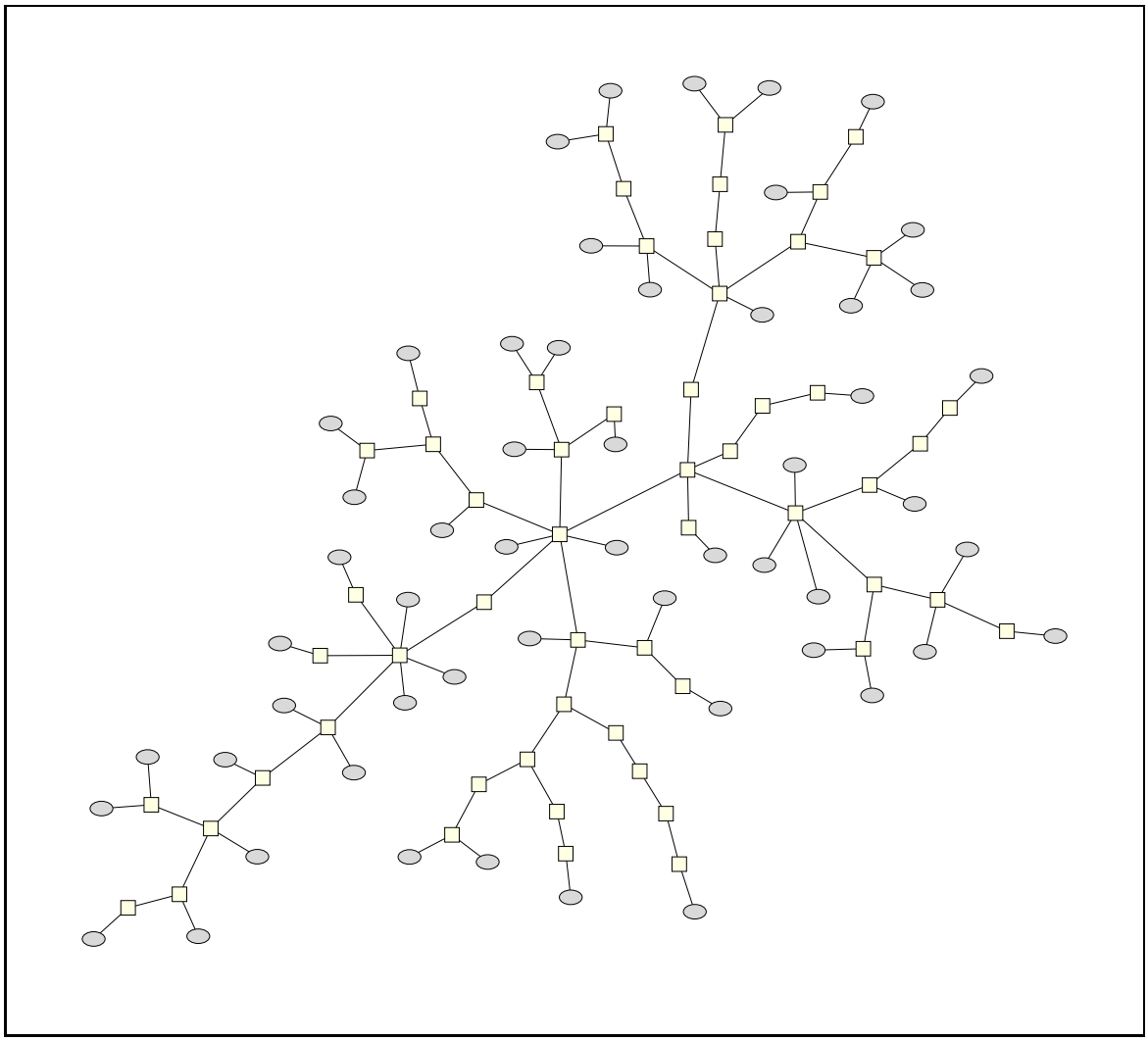


Abbildung A.6: Die Abbildung zeigt G_0 für die Probleminstanz N2.
N2 besteht aus 110 Knoten und insgesamt 1270 Kanten. 56 der Knoten sind Cut-Knoten. Die Gewichtung der Erweiterungskanten ist zufällig aus der Menge $\{10, 11, \ldots, 50\}$ gewählt.

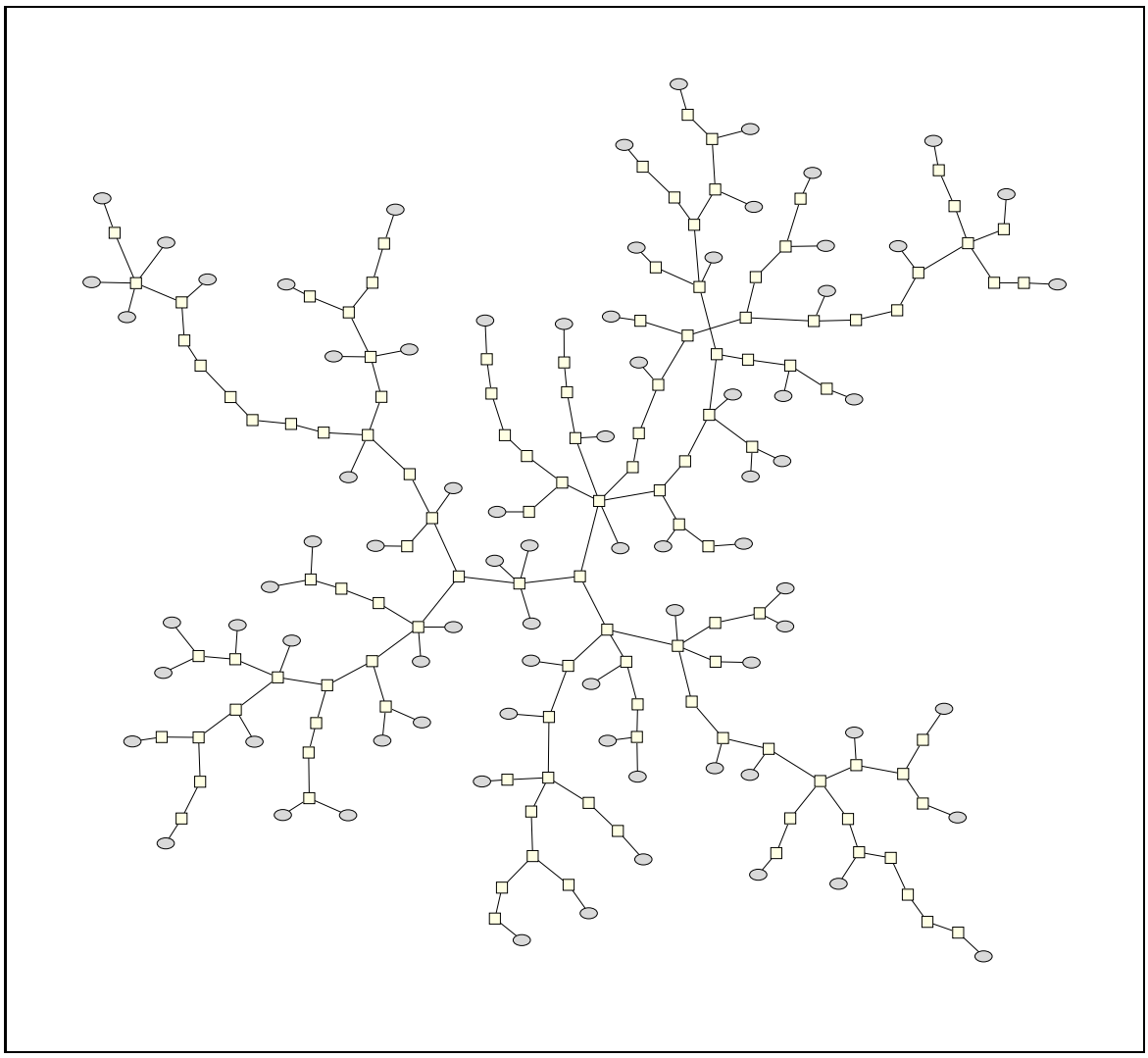


Abbildung A.7: Die Abbildung zeigt G_0 für die Probleminstanz R2.
R2 besteht aus 200 Knoten und insgesamt 9944 Kanten. 122 der Knoten sind Cut-Knoten. Die Gewichtung der Erweiterungskanten erfolgte aus dem Bereich $[5.0, 100.0]$.

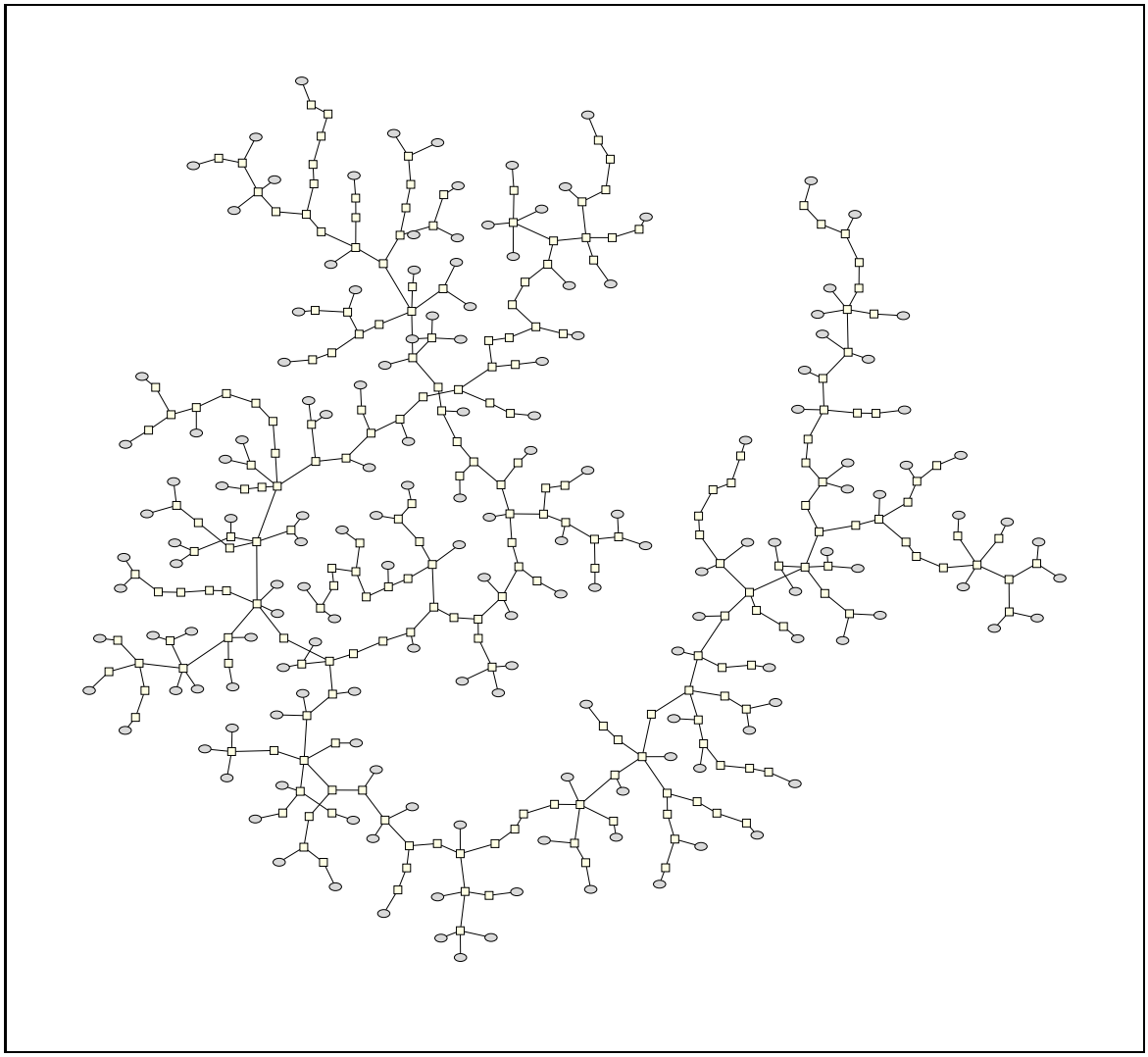


Abbildung A.8: Die Abbildung zeigt G_0 für die Probleminstanz E3.
E3 ist ein zufällig generierter euklidischer Graph und besteht aus 400 Knoten und insgesamt 8020 Kanten. 238 der Knoten sind Cut-Knoten.

A.2 Darstellung der euklidischen Graphen

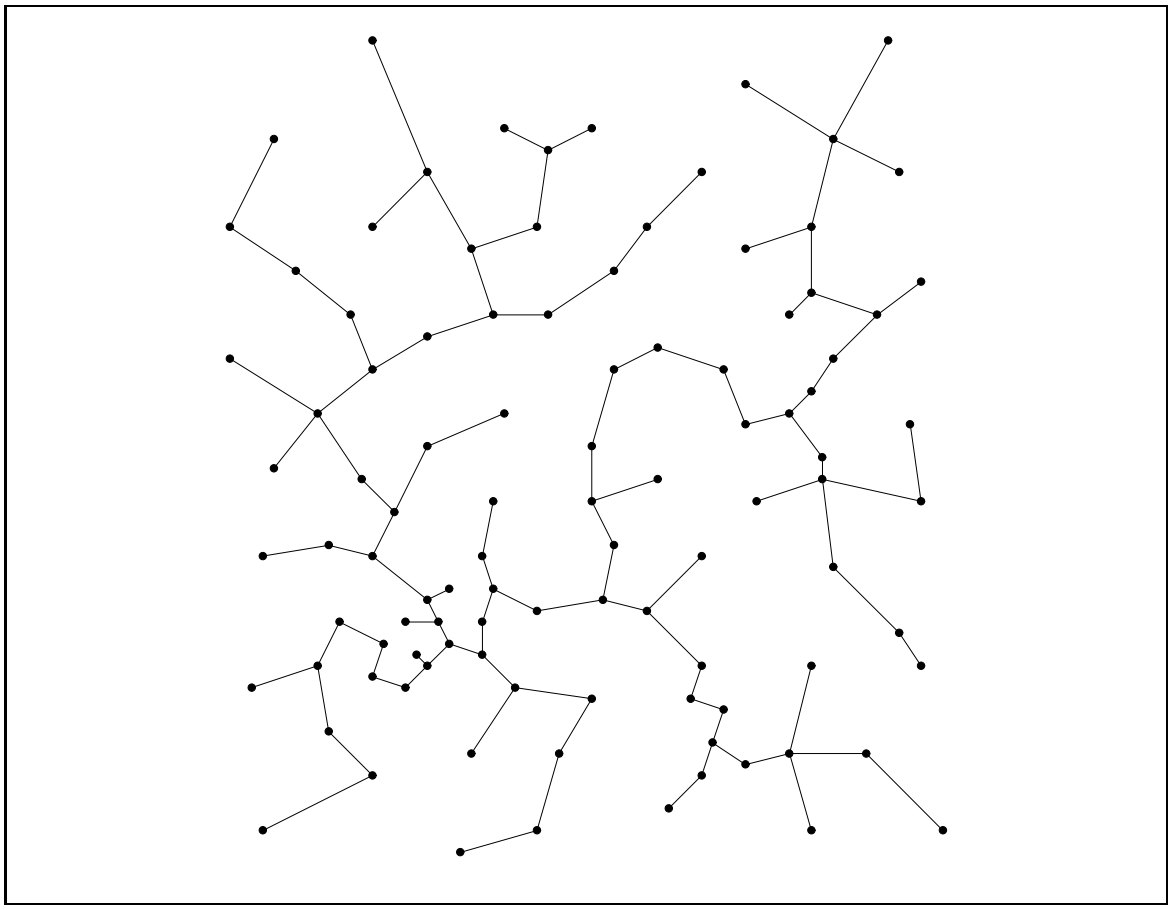


Abbildung A.9: Die Abbildung zeigt G_0 für die Probleminstanz eil101(∞). eil101(∞) ist ein kompletter Graph mit 101 Knoten und insgesamt 5050 Kanten. 68 der Knoten sind Cut-Knoten.

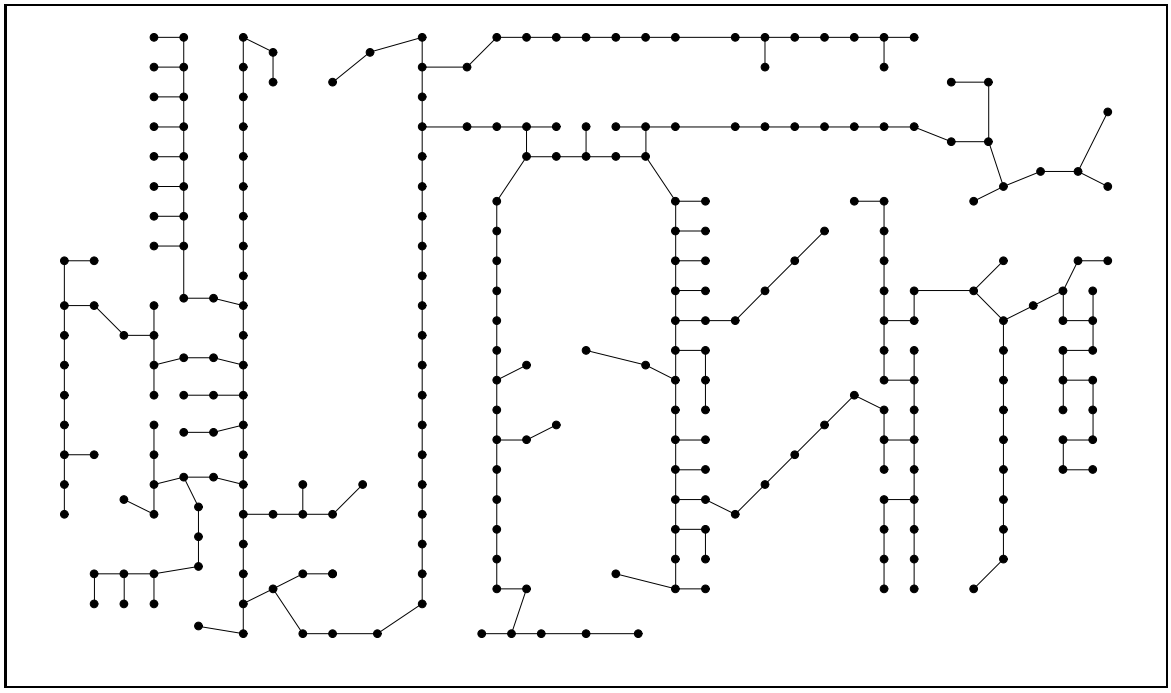


Abbildung A.10: Die Abbildung zeigt G_0 für die Probleminstanz a280(∞). a280(∞) ist ein kompletter Graph mit 280 Knoten und insgesamt 39060 Kanten. 217 der Knoten sind Cut-Knoten. Weitere Beispiele für dieselbe Knotenmenge verbinden jeden Knoten mit zumindest 50 bzw. 100 seiner topographisch nächstliegenden Nachbarknoten, um die Kantenmenge E zu generieren. E_0 ist jeweils ein in E fixierter minimaler Spannbaum.

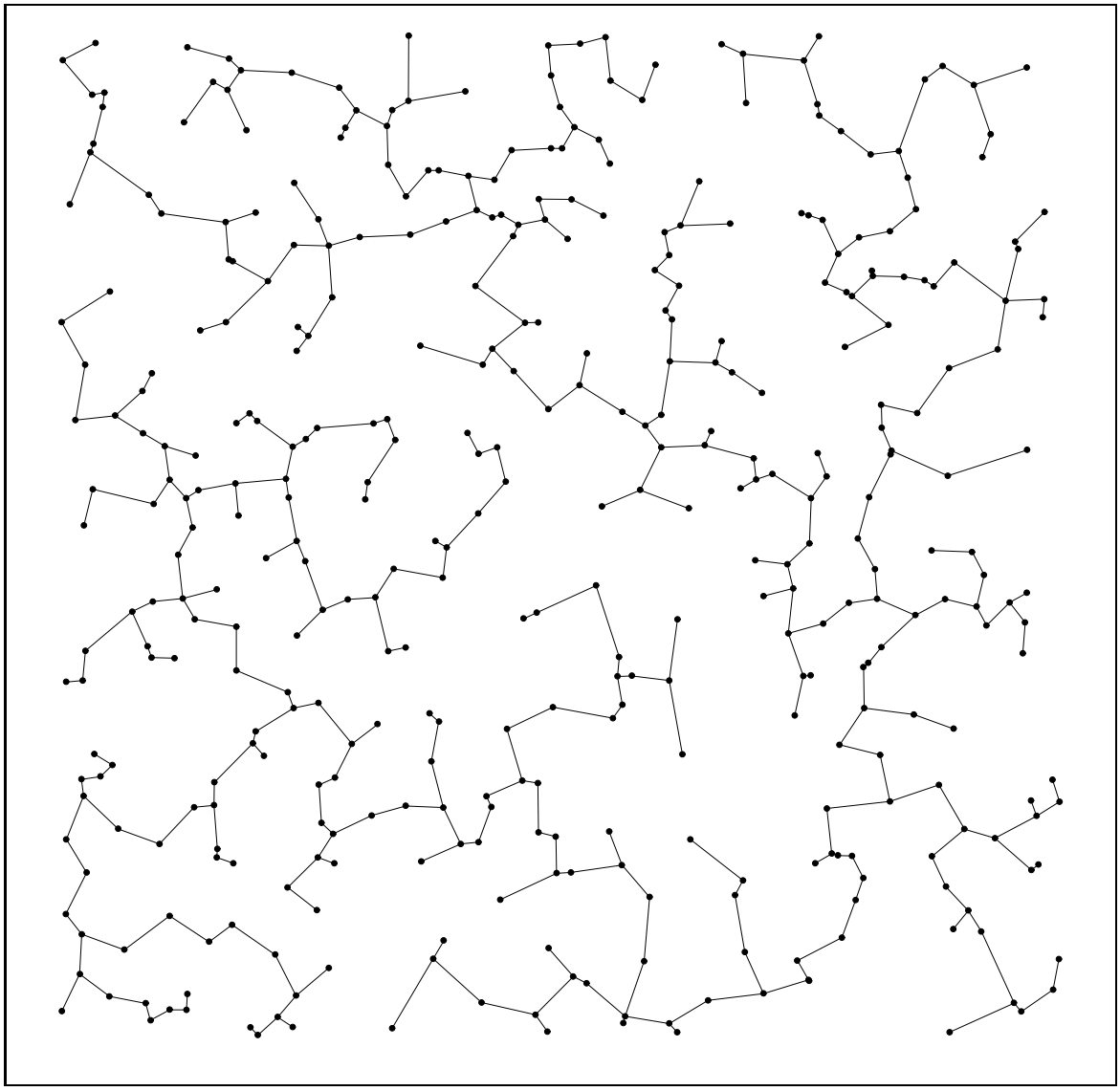


Abbildung A.11: Die Abbildung zeigt G_0 für die Probleminstanz rd400(∞), einen kompletten Graphen mit 400 Knoten und insgesamt 79800 Kanten. 304 der Knoten sind Cut-Knoten. E_0 ist ein minimaler Spannbaum.

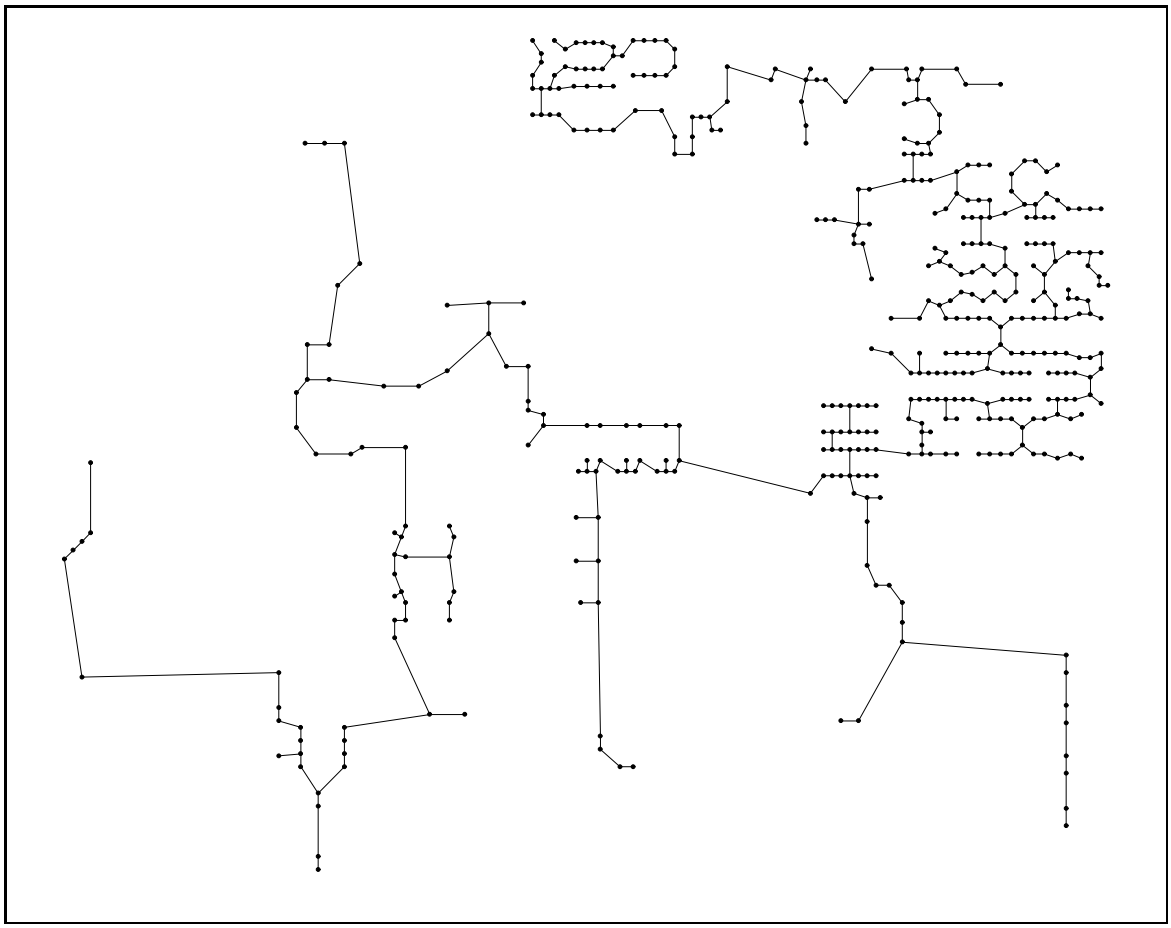


Abbildung A.12: Die Abbildung zeigt G_0 für die Probleminstanz pr439(∞), einen kompletten Graphen mit 439 Knoten und insgesamt 96141 Kanten. 363 der Knoten sind Cut-Knoten. Weitere Beispiele für dieselbe Knotenmenge verbinden jeden Knoten mit zumindest 100 bzw. 200 seiner topographisch nächstliegenden Nachbarknoten, um die Kantenmenge E zu generieren. E_0 ist jeweils ein in E fixierter minimaler Spannbaum.

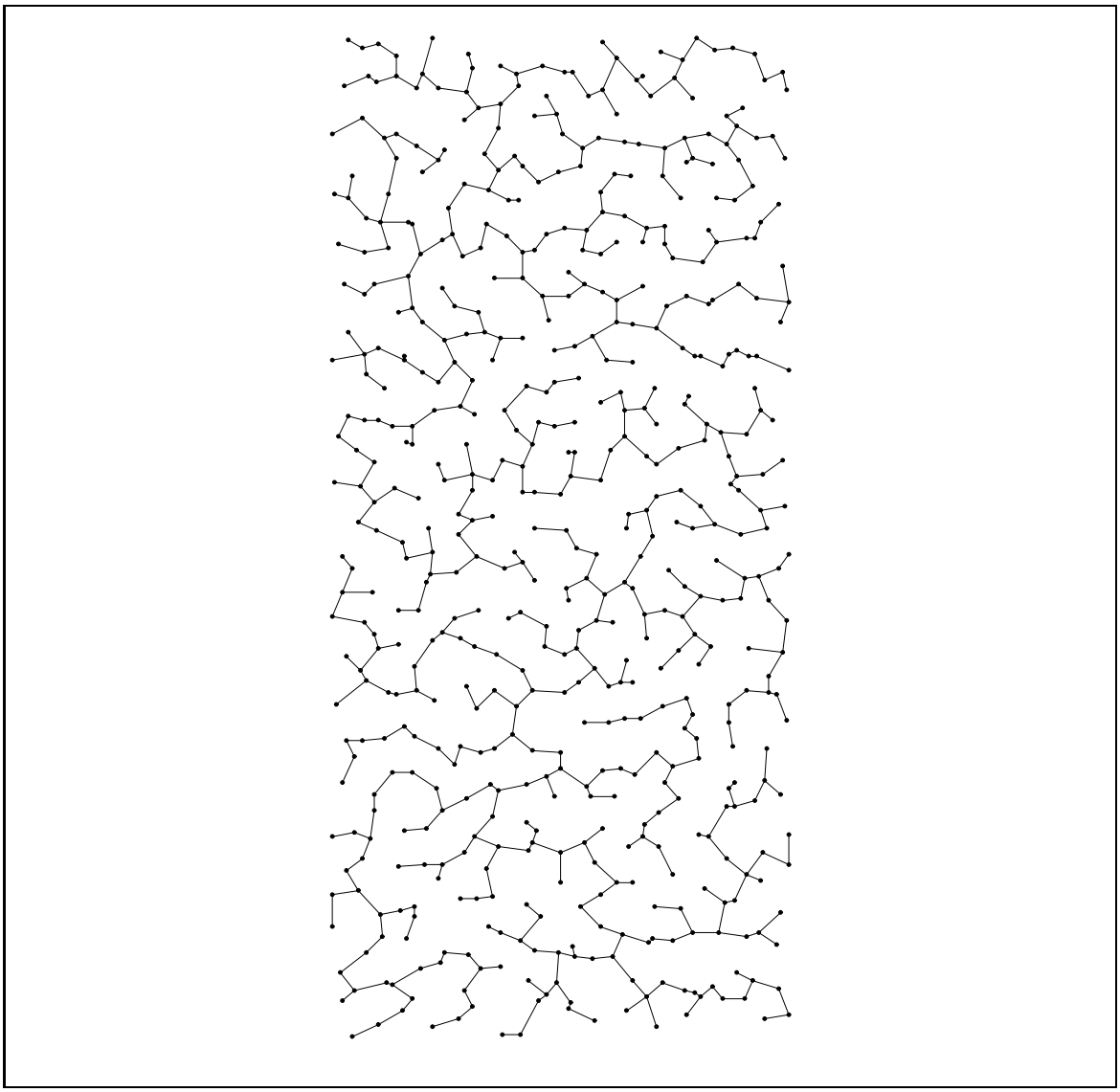


Abbildung A.13: Die Abbildung zeigt G_0 für die Probleminstanz rat575(∞), einen kompletten Graphen mit 575 Knoten und insgesamt 165025 Kanten. 436 der Knoten sind Cut-Knoten. Ein weiteres Beispiel für dieselbe Knotenmenge verbindet jeden Knoten mit zumindest 50 seiner topographisch nächstliegenden Nachbarknoten, um die Kantenmenge E zu generieren. E_0 ist jeweils ein in E fixierter minimaler Spannbaum.

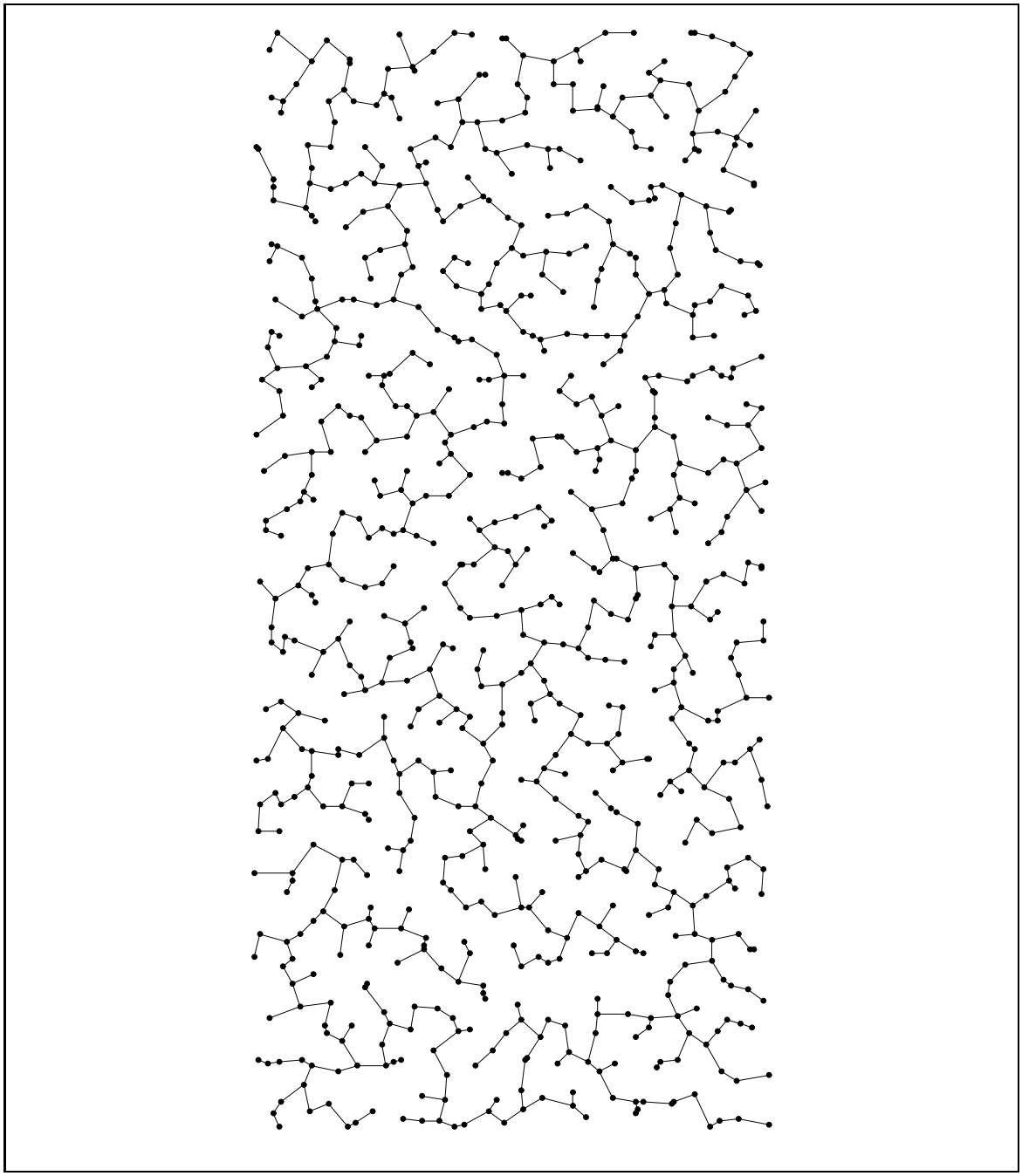


Abbildung A.14: Die Abbildung zeigt G_0 für die Probleminstanz rat783(50), bestehend aus 783 Knoten und insgesamt 21499 Kanten. 601 der Knoten sind Cut-Knoten. Ein weiteres Beispiel mit 12844 Kanten wurde für dieselbe Knotenmenge generiert.

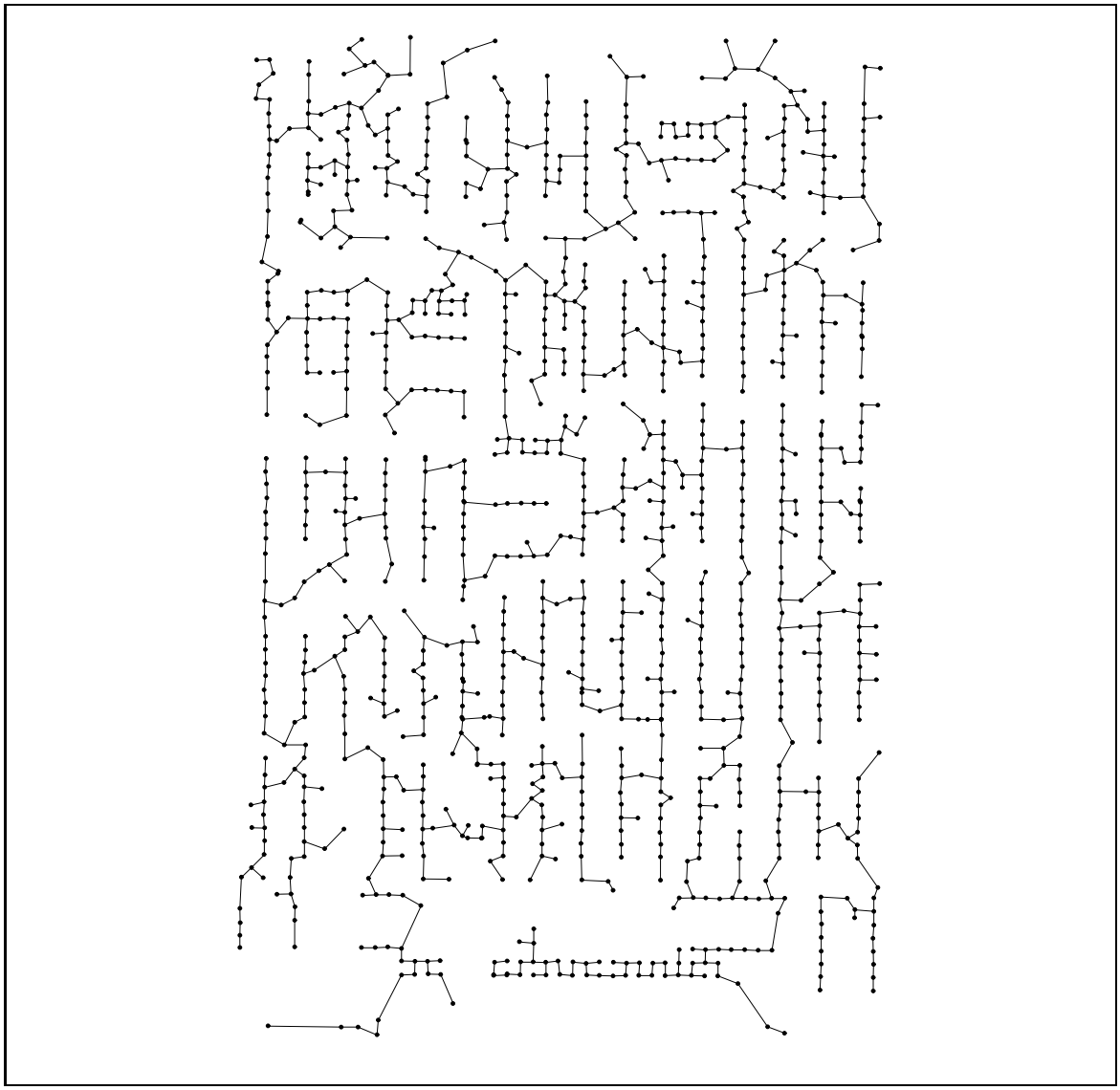


Abbildung A.15: Die Abbildung zeigt G_0 für die Probleminstanz pcb1173(30). Der Graph pcb1173(30) besteht aus 1173 Knoten und insgesamt 19493 Kanten. 953 der Knoten sind Cut-Knoten. E generiert sich aus dem Verbinden von jedem Knoten mit zumindest 30 seiner topographisch nächstliegenden Nachbarknoten. E_0 ist als minimaler Spannbaum in E fixiert.

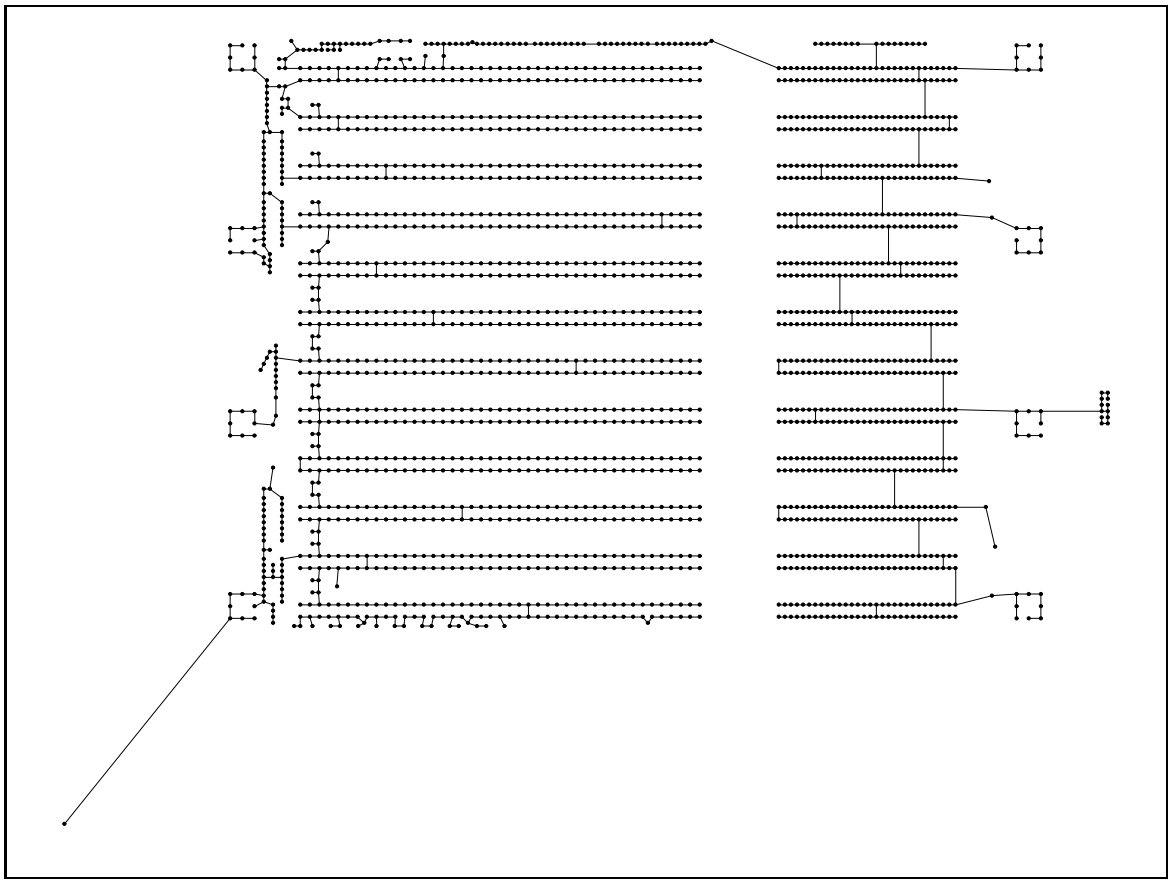


Abbildung A.16: Die Abbildung zeigt G_0 für die Probleminstanz d2103(50), einen Graphen bestehend aus 2103 Knoten und insgesamt 57732 Kanten. 1963 der Knoten sind Cut-Knoten. E generiert sich aus dem Verbinden von jedem Knoten mit zumindest 50 seiner topographisch nächstliegenden Nachbarknoten. E_0 ist als minimaler Spannbaum in E fixiert.

Anhang B

Veröffentlichung als Konferenzbeitrag

EvoCOP bezeichnet einen mittlerweile jährlich stattfindenden internationalen Workshop, der sich mit Evolutionären Algorithmen zur Lösung kombinatorischer Optimierungsprobleme befasst. EvoCOP 2002[1] wurde zusammen mit der EuroGP, der Europäischen Konferenz zum Thema Genetic Programming, in Irland abgehalten. Den Vorgaben entsprechend wurde der Inhalt dieser Arbeit auf 10 Seiten zusammengefasst. Es folgt der vom Programm-Komitee akzeptierte und in der *Lecture Notes in Computer Science (LNCS), Volume 2279* veröffentlichte Artikel [23].

[1] siehe auch http://www.ads.tuwien.ac.at/evocop2002/ und http://evonet.dcs.napier.ac.uk/eurogp2002/evocop.html

A Memetic Algorithm for Vertex-Biconnectivity Augmentation

Sandor Kersting, Günther R. Raidl, and Ivana Ljubić

Institute of Computer Graphics and Algorithms, Vienna University of Technology,
Favoritenstraße 9–11/186, 1040 Vienna, Austria
{kersting|raidl|ljubic}@ads.tuwien.ac.at

Abstract. This paper considers the problem of augmenting a given graph by a cheapest possible set of additional edges in order to make the graph vertex-biconnected. A real-world instance of this problem is the enhancement of an already established computer network to become robust against single node failures. The presented memetic algorithm includes an effective preprocessing of problem data and a fast local improvement strategy which is applied during initialization, mutation, and recombination. Only feasible, locally optimal solutions are created as candidates. Empirical results indicate the superiority of the new approach over two previous heuristics and an earlier evolutionary method.

1 Introduction

Robustness against failure is an important issue when designing a commercial computer network. It is often not acceptable that the failure of a single service node – be it a computer, router, or other device – leads to a disconnection of others. Redundant connections need to be established to provide alternative routes in case of a (temporary) break of any one node. We represent such a network by an undirected graph. It is said to be *vertex-biconnected* if at least two nodes need to be removed together with their incident edges in order to separate the graph into disconnected components. In a connected graph that is not vertex-biconnected a critical node whose removal would disconnect the graph is called *cut-point*. We say that we *cover a cut-point* when we add a set of edges to the graph which ensures that the removal of this vertex no longer disconnects the graph. Our global aim is to identify a set of edges with minimum total costs that covers all existing cut-points.

A more formal definition of the vertex-biconnectivity augmentation problem for graphs (V2AUG): Let $G = (V, E)$ be a vertex-biconnected, undirected graph with node set V and edge set E representing all possible connections. Each edge $e \in E$ has associated $cost(e) > 0$. A connected, spanning, but not vertex-biconnected subgraph $G_0 = (V, E_0)$ with $E_0 \subset E$ represents a fixed, existing network, and $E_a = E \setminus E_0$ is the set of edges that may be used for augmentation.

The objective is to determine a subset of these candidate edges $E_s \subseteq E_a$ so that the augmented graph $G_s = (V, E_0 \cup E_s)$ is vertex-biconnected and

$$cost(E_s) = \sum_{e \in E_s} cost(e) \tag{1}$$

is minimal.

The next section summarizes former approaches to this problem. Then a new memetic algorithm is presented. Section 3 explains its preprocessing, which creates needed data structures and reduces the size of the problem in general considerably by fixing or eliminating certain edges in safe ways. Section 4 describes the main algorithm, including a new local improvement algorithm for creating locally optimal offspring only. In Sect. 5 empirical results are presented and compared to two previous heuristics and a hybrid genetic algorithm. Conclusions are drawn in Sect. 6.

2 Previous Work

Eswaran and Tarjan [1] originally investigated the V2AUG problem. They showed it to be NP-hard. An exact polynomial-time algorithm could only be found for the special case when G is complete and each edge has unit costs [4].

Frederickson and Jàjà [2] provided an approximation algorithm for the general case which finds a solution within a factor 2 of the optimum. The algorithm includes a preprocessing step that transforms the fixed graph G_0 into a *block-cut tree*, see Sect. 3. Each potential augmentation edge from E_a is superimposed on the block-cut tree, and certain redundant edges are identified and eliminated. In the main part of the algorithm, the block-cut tree is directed toward an arbitrarily chosen leaf as root node, and each tree-edge is assigned zero costs. Each cut-point is substituted by star-shaped structures including new dummy-nodes in order to guarantee that strongly connecting the block-cut tree implies vertex-biconnectivity of the underlying fixed graph G_0. All superimposed augmentation edges are also directed. A minimum out-branching algorithm as described by Gabow et al. [3] is then applied to the block-cut tree including the superimposed augmentation edges to identify the solution's ede-set E_s. The computational effort of the algorithm is $O(|V|^2)$.

An improved variant of this approximation algorithm has been developed by Khuller and Thurimella [5]. It exhibits a time complexity of only $O(|E| + |V| \log |V|)$, but has the same approximation factor of 2.

An iterative approach based on Khuller and Thurimella's algorithm has been proposed by Zhu et al. [10]. In each step, a *drop*-heuristic measures the gain of each augmentation edge if it would be included in a final solution. This is achieved by calling the branching algorithm for each edge once with its cost set to zero and once with its original cost. The edge with the highest gain is then fixed, and its cost are permanently set to zero. The process is repeated until the obtained branching has zero total costs. Furthermore, the whole algorithm is applied with each leaf of the block-cut tree becoming once the root, and the overall

cheapest solution is the final one. Although the theoretical approximation factor remains 2, practical results are usually much better than when applying Khuller and Thurimella's algorithm. However, time requirements are raised significantly.

A straight-forward hybrid genetic algorithm for the V2AUG problem has been proposed by Ljubić and Kratica [6]. This algorithm is based on a binary encoding in which each bit corresponds to an edge in E_a. Standard uniform crossover and bit-flip mutation are applied. Infeasible solutions are repaired by a greedy algorithm which temporarily removes cut-points one by one and searches for suitable augmentation edges that reconnect separated components.

Another, "lighter" kind of connectivity property is edge-biconnectivity, which means that a graph remains connected after the removal of any single edge. While vertex-biconnectivity implies edge-connectivity, the reverse is not true. Similar algorithms as for V2AUG have been applied to the edge-biconnectivity augmentation problem (E2AUG). From the algorithmic point-of-view, E2AUG is easier to deal with, since it does not require the special block-cut tree.

Recently, Raidl and Ljubić described in [7, 9] an effective evolutionary algorithm for E2AUG. A compact edge set encoding and special initialization and variation operators that include a local improvement heuristic are applied. In this way, the space of locally optimal solutions is searched only. The approach belongs to the broader class of so-called *local-search-based memetic algorithms* [8].

Based on this algorithm for E2AUG, the memetic algorithm for V2AUG presented in this article has been developed. Major differences lie in the underlying data structures (e.g. the now necessary block-cut tree), the preprocessing, and the local improvement algorithm. While it is relatively easy to check and eventually establish the coverage of a single critical edge in case of E2AUG, this is significantly harder to achieve for a node in the V2AUG-case, especially in an efficient way: A fixed edge can always be covered by a single augmentation edge, and it is obvious which augmentation edges are able to cover the fixed edge. On the other side, a combination of multiple augmentation edges is in general necessary to cover a cut-point completely.

3 Preprocessing

During preprocessing, a block-cut tree is derived from the fixed graph G_0 according to [1], and other supporting data structures are created. They are all needed for an efficient implementation of the main algorithm. Furthermore, several deterministic rules are applied in order to reduce E_a in a safe way. The following paragraphs describe these mechanisms in detail.

3.1 The Block-Cut Tree

A block-cut tree $T = (V_T, E_T)$ with node set V_T and edge set E_T is an undirected tree that represents the connections between already vertex-biconnected components (called *blocks*) and cut-points of the underlying fixed graph G_0.

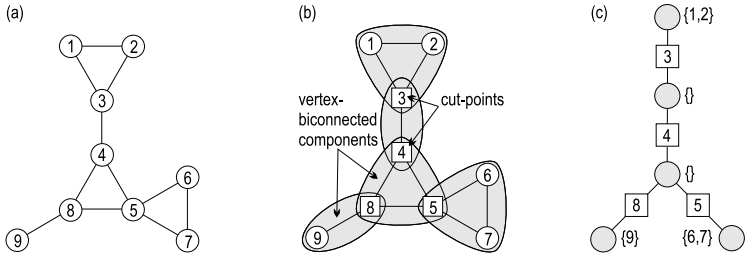


Fig. 1. The derivation of a block-cut tree: (a) given graph G_0, (b) identified blocks (shaded areas) and cut-points (square nodes), and (c) the block-cut tree.

Two types of nodes form V_T: cut-nodes and block-nodes. Each cut-point in G_0 is represented by a corresponding cut-node in V_T, each maximal vertex-biconnected block in G_0 by a unique block-node in V_T. A block-node is associated with all nodes of the represented block in G_0 that are no cut-points. If the represented block consists of cut-points only, the block-node is not associated with any node from V_0.

A cut-node and a block-node are connected by an edge in E_T iff the corresponding cut-point is part of the block in G_0. Thus, cut-nodes and block-nodes always alternate on any path in T. The resulting structure is always a tree, since otherwise, the nodes forming a cycle can be shrinked into a single, larger block. Figure 1 illustrates the derivation of the block-cut tree.

After T has been derived from G_0, all potential augmentation edges in E_a are superimposed on T forming a new edge-set E_A: For each edge $(u, v) \in E_a$, a corresponding edge (u', v') is created with $u', v' \in V_T$ being the nodes that are associated with u, respectively v. The mapping from E_A to E_a is stored in order to be finally able to derive the original edges of an identified solution. Note that $G_A = (V_T, E_T \cup E_A)$ may be a multi-graph containing self-loops and multiple edges between two nodes; however, the reductions described in Sect. 3.3 will make this graph simple.

3.2 When is a Cut-Point Covered?

A block-cut tree's edge $e \in E_T$ is said to be covered by an augmentation edge $e_A = (u, v) \in E_A$ iff e is part of the unique path in T connecting u with v. In order to cover a cut-node $v_c \in V_T$ completely, all its incident edges need to be covered, but this is in general not a sufficient condition.

If v_c would be removed from T, the tree will fall into k disconnected components $C_1^{v_c}, \ldots, C_k^{v_c}$, where k is the degree of v_c; we call them cut-components of v_c. We say an augmentation edge $e_A = (u, v) \in E_A$ *contributes in covering the cut-node* v_c, iff *two* edges incident to v_c are covered by e_A. Such an augmentation edge is obviously not incident to v_c and unites two cut-components $C_i^{v_c}$ and $C_j^{v_c}$. To cover v_c completely, exactly $k - 1$ augmentation edges are needed, and they must unite all components $C_1^{v_c}, \ldots, C_k^{v_c}$ into one.

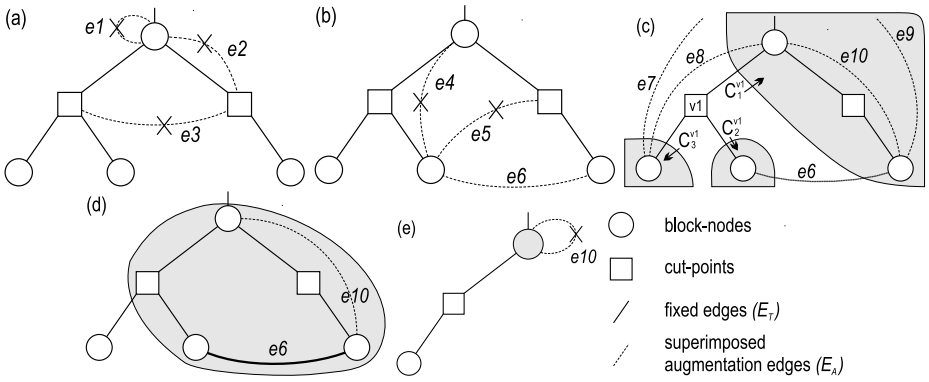


Fig. 2. Examples for preprocessing: (a) Edges that do not contribute in covering a cut-point are removed. (b) When $cost(e6) \leq cost(e4)$ and $cost(e6) \leq cost(e5)$, $e4$ and $e5$ are discarded. (c) As $e6$ is the only edge that connects C_2^{v1} to any other cut-component of $v1$, it is fixed; (d) the cycle caused by fixing $e6$ is shrinked into a new block; (e) $e10$ becomes a self-loop and is finally also discarded.

For any cut-node v_c, let $A(v_c) \subseteq E_A$ be the set of all augmentation edges that contribute in covering v_c by uniting two of its cut-components. Furthermore, for each $e_A \in E_A$, let $R(e_A)$ be the set of all cut-nodes to whose covering e_A contributes. The proposed memetic algorithm explicitly computes and stores all sets $A(v_c)$ for all cut-nodes and the sets $R(e_A)$ for all augmentation edges as supporting data structures for its main part.

Later, we need to check efficiently if a certain cut-node is covered by a subset of augmentation edges $S \subset E_A$. This check is in general performed in $O(|S|)$ time with the aid of a union-find data structure. However, in most cases the degree of the cut-node is less than four, and then it is sufficient just to check whether each cut-component of the considered cut-node is connected to any other cut-component.

3.3 Reducing the Search Space

From E_A we discard all edges that do not contribute in covering any cut-node ($R(e_A) = \{\}$). In particular, edges forming self-loops or edges connecting a cut-node with an adjacent block-node or with another cut-node adjacent to the same block-node are removed in this way; see Fig. 2(a). Furthermore, from multiple edges connecting the same nodes in T, only one with minimum weight is retained. In this way, $G_A = (V_T, E_T \cup E_A)$ becomes a simple graph.

In addition to these simple reductions, we apply the following more sophisticated steps which are partly adopted from [9].

Edge Elimination: If there are two edges $e_A, e'_A \in E_A$, $cost(e_A) \leq cost(e'_A)$, and e_A covers all those edges which are covered by e'_A (in addition to others), e'_A is obsolete and can be discarded. All such edges can be identified in $O(|V|^2)$ time

as a byproduct from a dynamic programming algorithm that computes distance values needed for the algorithm from Frederickson and Jájá [2]; see Fig. 2(b).

Fixing of Edges: An edge $e_A \in E_A$ must be included in any feasible solution to the V2AUG problem, when it represents the only possibility to connect a cut-node's cut-component $C_i^{v_c}$ to any other cut-component of v_c. In more detail, we process for each cut-node v_c its set $A(v_c)$ and look for such edges, which are then fixed by moving them from E_A to E_T; see Fig. 2(c). The corresponding original augmentation edges from E_a are permanently marked to be included in any future solution.

Shrinking: By fixing an edge, a cycle is introduced in T. This cycle forms a new vertex-biconnected component that can be shrinked into a single block-node, see Fig. 2(d) and 2(e). After shrinking all cycles, all changes in T are reflected to the supporting data structures. Owing to these changes, more edges may become available for elimination. Therefore, all reduction steps are repeated until no further shrinking is possible.

4 The Memetic Algorithm

The main part of the new approach is a steady-state evolutionary algorithm, in which in each iteration one new candidate solution is always created by selecting two parents in k-ary tournaments with replacement, recombining them and applying mutation. Such a solution replaces the worst solution in the population with one exception: To maintain a minimum diversity, a new candidate that resembles a solution already contained in the population is discarded.

A solution is represented by directly storing the set of its augmentation edges $S \subseteq E_A$ in the form of a hash-table. In this way, only $O(|S|) = O(|V|)$ space is needed, since $|S| < |V|$ in any locally optimal solution, in general becoming $|S| \ll |V|$ with increasing number of nodes, and an edge can be added, deleted or checked for existence in constant time.

Local Improvement: For the creation of initial solutions and the variation operators that derive new solutions, the following local improvement method plays a central role. From a feasible solution S, it removes redundant edges until the solution becomes locally optimal in the sense that no further edge can be removed without including others or making the solution infeasible.

For the cut-components of each cut-node, it is first determined how often each of them is connected to any other by the edges in S. Edges that provide the only connection for any cut-component must always be included in a feasible solution and are therefore not redundant.

The remaining edges in S are then processed one-by-one in decreasing cost-order. All cut-nodes in whose coverage a certain augmentation edge $e \in S$ participates ($R(e)$) are checked if they remain covered when e is removed, see Sect. 3.2. If this is the case, this edge is actually redundant and removed from S.

In the worst case, the computational effort of this local improvement may be $O(|V|^3)$, however, it is much lower on average.

Initialization: A member of the initial population is created by randomly selecting edges from E_A without replacement and including each in the initially empty edge-set S if it is not redundant. This process stops when all cut-nodes are covered.

The selection of edges for inclusion is biased toward cheaper edges by sorting E_A according to costs, and choosing an edge via the following random-rank:

$$rank = \lfloor |\mathcal{N}(0,s)| \cdot |V| \rfloor \bmod |E_A|\,, \tag{2}$$

$\mathcal{N}(0,s)$ is a normally distributed random variable with zero mean and standard deviation s, a strategy parameter that determines the strength of biasing.

Recombination: This operator was designed with the aim to provide highest possible heritability. First, edges common in both parents S_1 and S_2 are always adopted: $S = S_1 \cap S_2$. Then, while not all cut-nodes are covered, an edge is randomly selected from the set of remaining parental edges $((S_1 \cup S_2) \setminus (S_2 \cap S_1))$ and included in the offspring S if it provides a new cover. To emphasize the inclusion of low-cost edges, they are selected via binary tournaments with replacement. As final step, local improvement is called.

Mutation: Having created a new offspring via recombination, mutation is applied with a certain probability in order to introduce new edges that were not available in the parents. From S, one edge is selected randomly and removed. This makes one or more cut-nodes uncovered. These cut-nodes are identified and newly covered in random order: For each cut-node v_c, the edges from $A(v_c)$ that would actually help in covering v_c anew are determined. From this set, edges are repeatedly chosen at random and included in S until v_c is completely covered. Finally, local improvement is applied again.

The selection of the edge to be removed is biased toward more expensive edges: A pair of edges is drawn at random and a binary tournament selection with replacement decides among them.

5 Empirical Results

To compare the presented approach with other algorithms we have used test instances of different size and structure. Since shrinking can always reduce the problem of augmenting a general connected graph G_0 to the problem of augmenting a tree, G_0 is always a spanning tree in these test instances. Table 1 shows the number of nodes, the number of augmentation edges, and the number of cut-points (CP) before and after applying the memetic algorithm's preprocessing.

The first eight instances have been created with Zhu's generator [10] and were already used in [6, 7, 9]. The remaining ones are derived from Euclidean instances of Reinelt's TSP-library[1] in the following way: G is the graph containing all nodes

[1] www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95

Table 1. Problem instances and results of the memetic algorithm's preprocessing.

instance	$\|V\|$	$\|E_a\|$	$cost(e) \in$	$\mathrm{CP}(G_0)$	$\|V_T\|$	$\|E_A\|$	$\mathrm{CP}(T)$
B1	60	55	$\{1, 2, ..., 1770\}$	34	1	0	0
C3	100	149	$\{1, 2, ..., 4950\}$	53	59	66	33
M1	70	290	$\{0, 10, ..., 1000\}$	37	70	227	37
M2	80	327	$\{0, 10, ..., 1000\}$	41	80	242	41
M3	90	349	$\{0, 10, ..., 1000\}$	42	90	262	42
N1	100	1104	$\{10, 11, ..., 50\}$	50	100	687	50
N2	110	1161	$\{10, 11, ..., 50\}$	56	110	734	56
R1	200	9715	$\{1, 2, ..., 100\}$	115	200	3995	115
a280 (50)	280	7654	TSP-lib Eucl.	218	280	1561	218
a280 (100)	280	15769	TSP-lib Eucl.	220	280	3322	220
a280 (∞)	280	38781	TSP-lib Eucl.	217	280	10182	217
rd400 (100)	400	22610	TSP-lib Eucl.	306	400	4959	306
rd400 (200)	400	46444	TSP-lib Eucl.	306	400	9368	306
pr439 (100)	439	26865	TSP-lib Eucl.	364	439	6095	364
pr439 (200)	439	55111	TSP-lib Eucl.	362	439	11890	362
pr439 (∞)	439	95703	TSP-lib Eucl.	362	439	19574	362
rat575 (∞)	575	164451	TSP-lib Eucl.	436	575	34514	436
pcb1173 (30)	1173	18321	TSP-lib Eucl.	953	1173	4492	953
d2103 (50)	2103	55630	TSP-lib Eucl.	1963	2103	6657	1963

Table 2. Results of the heuristics from Khuller and Thurimella (KT), Zhu et al. (ZKR), the genetic algorithm from Ljubić and Kratica (LK), and the memetic algorithm (MA).

		KT	ZKR	LK		MA				
instance	C^*	*gap*	*gap*	*gap*	*evals*	*gap*	σ	t [s]	*evals*	SR [%]
B1	15512.0*	8.6	0.0	0.0	900	0.0	0.0	< 1	1	100.0
C3	59129.0*	10.4	1.1	0.0	7300	0.0	0.0	< 1	906	100.0
M1	2940.0*	8.2	0.0	0.0	2700	0.0	0.0	2	2509	100.0
M2	4600.0*	5.9	0.0	0.0	8700	0.0	0.0	2	1667	100.0
M3	4980.0*	6.2	0.4	0.0	8100	0.0	0.0	4	2387	100.0
N1	390.0*	31.5	4.1	4.9	27800	0.0	0.0	10	4671	100.0
N2	429.0*	39.2	4.0	2.3	90000	0.0	0.0	12	7777	100.0
R1	121.4*	16.8	1.9	-	-	0.1	0.2	169	27738	36.7
a280 (50)	474.0*	25.1	-	-	-	0.0	0.1	15	8831	80.0
a280 (100)	473.0*	29.4	-	-	-	0.1	0.4	27	13568	93.3
a280 (∞)	489.0*	21.9	-	-	-	0.9	0.6	77	16445	3.3
rd400 (100)	4001.0	28.9	-	-	-	0.2	0.2	138	36694	20.0
rd400 (200)	3990.0	28.6	-	-	-	0.6	0.5	263	43521	3.3
pr439 (100)	27907.0*	20.5	-	-	-	0.7	0.5	181	29286	3.3
pr439 (200)	28289.0	19.1	-	-	-	1.1	0.5	296	43157	6.7
pr439 (∞)	27810.0	20.5	-	-	-	2.8	1.1	470	42512	3.3
rat575 (∞)	1546.0	33.4	-	-	-	1.3	0.6	1259	28668	3.3
pcb1173 (30)	11464.0	28.1	-	-	-	0.2	0.1	3202	49737	13.3
d2103 (50)	7333.0	9.6	-	-	-	0.0	0.0	14388	17974	63.3

of the TSP-instance and edges for each node to its nearest k neighbors, where k is the number shown in parentheses in Table 1; $k = \infty$ represents the complete graph. Edge costs are always the Euclidean distances rounded to nearest integer values. From G, a minimum spanning tree is derived and fixed as G_0.

Results of preprocessing document that a fixing of edges is only possible in shallower graphs like B1 and C3, where the number of cut-points could be dramatically reduced. In case of dense graphs, edge-elimination was highly effective. On average, the number of augmentation edges could be reduced to about a quarter.

The following setup was used for the memetic algorithm as it proved to be robust in preliminary tests. Population size: 800; group size for tournament selection: 5; parameter s for biasing initialization toward cheaper edges: 2.5; mutation probability 0.7. Each run was terminated when no new best solution could be identified during the last 10,000 iterations.

We compare the memetic algorithm, called MA, to the heuristics from Khuller and Thurimella [5] (KT), Zhu et al. [10] (ZKR), and the hybrid genetic algorithm from Ljubić and Kratica [6] (LK). For smaller instances, we were able to derive optimum solution values by a not yet published branch-and-cut approach. Table 2 shows in column C^* these optimum values marked by '*' or otherwise best-known solution values. For the heuristic approaches, the qualities of final solutions are reported as percentage gaps with respect to C^*: $gap = (cost(S) - C^*)/C^* \cdot 100\%$.

KT was run once for each leaf-node becoming the root of branching, and the best obtained gaps are shown. ZKR could only by applied to smaller instances owing to its high computational effort. The same is true for LK, for which the shown gaps are adopted from [6]; they represent best values obtained from 10 runs per instance. MA's gaps are averaged over 30 runs for all instances, and σ shows the gaps' standard deviations. t gives the CPU-times on a PentiumIII/800MHz PC and *evals* the number of evaluated solutions until the finally best solutions had been identified. The success rate SR is the percentage of MA's runs that yielded optimum or best-known solutions.

It can be seen that KT performed generally worst. For the smaller instances, where results of ZKR and LK are available, MA found optimum solutions in any run. Furthermore, MA scaled well to larger instances. In all our test cases, it could identify solutions with gaps less than 3% with high reliability, as shown by the small deviations. The running times and needed numbers of evaluations increase only moderately with the problem size. Figure 3 shows two exemplary solutions.

6 Conclusions

The main features of the proposed memetic algorithm for the vertex-biconnectivity augmentation problem are: The effective deterministic preprocessing which reduces the search space in many cases dramatically, the local improvement procedure which guarantees local optimality of any created candidate solution, and

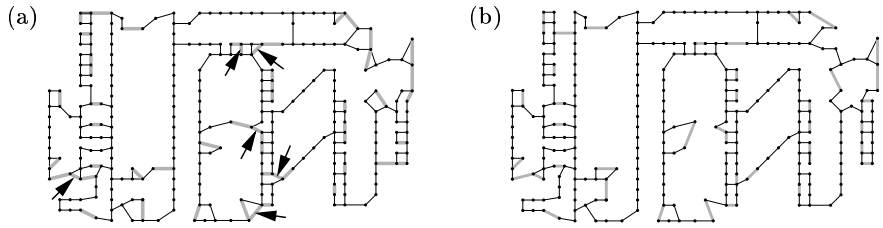


Fig. 3. Solutions to the Euclidean problem instance a280 (100) found by (a) Khuller and Thurimella's heuristic (gap: 29.4%, $|S| = 53$) and (b) the memetic algorithm (optimal, $|S| = 40$). The solutions' augmentation edges are shown in gray. In (a), the arrows mark obviously redundant edges.

the strong heritability and locality of the proposed recombination, respectively mutation. Furthermore, the local cost-based heuristics in the edge-selections of initialization, recombination, and mutation play a significant role.

Empirical tests indicate that the algorithm calculates solutions of high quality which are optimal in many cases. In particular the approach scales well to large problem instances owing to its relatively low computational effort for the creation and local improvement of one candidate solution.

References

1. K. P. Eswaran and R. E. Tarjan. Augmentation problems. *SIAM Journal on Computing*, 5(4):653–665, 1976.
2. G. N. Frederickson and J. Jájá. Approximation algorithms for several graph augmentation problems. *SIAM Journal on Computing*, 10(2):270–283, 1981.
3. H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.
4. T.-S. Hsu and V. Ramachandran. On finding a minimum augmentation to biconnect a graph. *SIAM Journal on Computing*, pages 889–912, 1993.
5. S. Khuller and R. Thurimella. Approximation algorithms for graph augmentation. *Journal of Algorithms*, 14(2):214–225, 1993.
6. I. Ljubić and J. Kratica. A genetic algorithm for the biconnectivity augmentation problem. In C. Fonseca, J.-H. Kim, and A. Smith, editors, *Proceedings of the 2000 IEEE Congress on Evolutionary Computation*, pages 89–96. IEEE Press, 2000.
7. I. Ljubić and G. R. Raidl. An evolutionary algorithm with hill-climbing for the edge-biconnectivity augmentation problem. In E. J. Boers, S. Cagnoni, J. Gottlieb, E. Hart, P. L. Lanzi, G. R. Raidl, R. E. Smith, and H. Tijink, editors, *Applications of Evolutionary Computation*, volume 2037 of *LNCS*, pages 20–29. Springer, 2001.
8. P. Moscato. Memetic algorithms: A short introduction. In D. Corne et al., editors, *New Ideas in Optimization*, pages 219–234. McGraw Hill, 1999.
9. G. R. Raidl and I. Ljubić. Evolutionary local search for the edge-biconnectivity augmentation problem. *to appear in Information Processing Letters*, 2001.
10. A. Zhu, S. Khuller, and B. Raghavachari. A uniform framework for approximating weighted connectivity problems. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 937–938, 1999.

Literaturverzeichnis

[1] Aho, A. V., J. E. Hopcroft und J. D. Ullman: *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[2] Algorithmic Solutions Software GmbH: *LEDA – Library of Efficient Data Types and Algorithms*, 1998–2003.

[3] Baron, G. und P. Kirschenhofer: *Einführung in die Mathematik für Informatiker Band 3*. Springer-Verlag, 1989.

[4] Bäck, T.: *Evolutionary Algorithms. SIGBIO Newsletter, 12(2):26–31, June 1992. ga:Back92d.*, 1992.

[5] Bäck, T.: *Selective Pressure in Evolutionary Algorithms: A Characterization of Selection Mechanisms*, 1994.

[6] Blickle, T.: *Tournament Selection*. In: Bäck, T., D. B. Fogel und Z. Michalewicz (Hrsg.): *Handbook of Evolutionary Computation*, S. C2.3:1–C2.3:4. Oxford University Press, New York, 1997.

[7] Blickle, T. und L. Thiele: *A Comparison of Selection Schemes Used in Genetic Algorithms*. Techn. Ber. 11, Gloriastrasse 35, 8092 Zurich, Switzerland, 1995.

[8] Breymann, U.: *Designing Components with the C++ STL*. Addison Wesley, 1998.

[9] Ellison, R., D. Fisher, R. Linger, H. Lipson, T. Longstaff und N. Mead: *Survivable Network Systems: An Emerging Discipline*. Techn. Ber. CMU/SEI-97-

TR-013, Pittsburgh,PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[10] Eswaran, K. P. und R. E. Tarjan: *Augmentation Problems*. SIAM Journal on Computing, 5(4):653–665, 1976.

[11] Feddersen, S., F. Heinzmann und E. Schöneburg: *Genetische Algorithmen und Evolutionsstrategien: Eine Einführung in Theorie und Praxis der simulierten Evolution*. Addison Wesley, 1994.

[12] Fonseca, C., J.-H. Kim und A. Smith (Hrsg.): *Proceedings of the 2000 IEEE Congress on Evolutionary Computation*. IEEE Press, 2000.

[13] Frederickson, G. N. und J. JáJá: *Approximation Algorithms for Several Graph Augmentation Problems*. SIAM Journal on Computing, 10(2):270–283, 1981.

[14] Fukunaga, A. S.: *Restart Scheduling for Genetic Algorithms*. Lecture Notes in Computer Science, 1498:357–, 1998.

[15] Gabow, H. N., Z. Galil, T. Spencer und R. E. Tarjan: *Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs*. Combinatorica, 6(2):109–122, 1986.

[16] Goldberg, D. E.: *Genetic Algorithms in Search, Optimization, and Learning*. Addison-Wesley, Reading, Massachusetts, 1989.

[17] Grötschel, M., C. L. Monma und M. Stoer: *Design of Survivable Networks*. In: Monma, C., G. Nemhauser, M. O. Ball und T. L. Magnanti (Hrsg.): *Handbooks in Operations Research and Management Science. Vol. 7: Network Models*, S. 617–672, 1995.

[18] Heesch, D.: *Doxygen – A Documentation System for C++, C, Java and IDL*, 1997–2003.

[19] Hojjat, A. und S. L. Hung: *Machine Learning, Neural Networks, Genetic Algorithms and Fuzzy Systems*. John Wiley and Sons, 1995.

[20] Holland, J.: *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

[21] Hsu, T.-S. und V. Ramachandran: *Finding a Smallest Augmentation to Biconnect a Graph*. SIAM Journal on Computing, 22(5):889–912, 1993.

[22] Jones, J. und S. Forrest: *Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms*. In: Eshelman, L. J. (Hrsg.): *Proceedings of the Sixth International Conference on Genetic Algorithms*, S. 184–192. Morgan Kaufmann, 1995.

[23] Kersting, S., G. R. Raidl und I. Ljubić: *A Memetic Algorithm for Vertex-Biconnectivity Augmentation*. In: Cagnoni, S., J. Gottlieb, E. Hart und others (Hrsg.): *Applications of Evolutionary Computing: EvoWorkshops 2002*, Bd. 2279 d. Reihe *LNCS*, S. 102–111. Springer, 2002.

[24] Khuller, S. und R. Thurimella: *Approximation Algorithms for Graph Augmentation*. Journal of Algorithms, 14(2):214–225, 1993.

[25] Laprie, J. C.: *Dependability: Basic Concepts and Terminology in English, French, German and Japanese*. Springer Verlag, 1992.

[26] Ljubić, I. und J. Kratica: *A Genetic Algorithm for the Biconnectivity Augmentation Problem*. In: Fonseca, C. et al. [12], S. 89–96.

[27] Ljubić, I. und G. R. Raidl: *A Memetic Algorithm for Minimum-Cost Vertex-Biconnectivity Augmentation of Graphs*. To appear in the Journal of Heuristics, 2003.

[28] Ljubić, I. und G. R. Raidl: *An Evolutionary Algorithm with Hill-Climbing for the Edge-Biconnectivity Augmentation Problem*. In: Boers, E. J. W., S. Cagnoni, J. Gottlieb und others (Hrsg.): *Applications of Evolutionary Computing: EvoWorkshops 2001*, Bd. 2037 d. Reihe *LNCS*, S. 20–29. Springer, 2001.

[29] Ljubić, I., G. R. Raidl und J. Kratica: *A Hybrid GA for the Edge-Biconnectivity Augmentation Problem*. In: Deb, K., G. Rudolph, X. Yao und

H.-P. Schwefel (Hrsg.): *Proceedings of the 2000 Parallel Problem Solving from Nature VI Conference*, Bd. 1917 d. Reihe *LNCS*, S. 641–650. Springer, 2000.

[30] Luke, S.: *When Short Runs Beat Long Runs.* In: Spector, L., E. D. Goodman, A. Wu, W. B. Langdon, H. M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon und E. Burke (Hrsg.): *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, S. 74–80, San Francisco, California, USA, 7-11 2001. M. Kaufmann.

[31] Merz, P. und B. Freisleben: *Fitness Landscapes and Memetic Algorithm Design.* In: *New Ideas in Optimization*, S. 245–260. McGraw-Hill, Berkshire, England, 1999.

[32] Merz, P. und B. Freisleben: *Fitness Landscape Analysis and Memetic Algorithms for the Quadratic Assignment Problem.* IEEE Transactions on Evolutionary Computation, 4(4):337–352, 2000.

[33] Michalewicz, Z.: *Genetic Algorithms + Data Structures = Evolution Programs.* Springer, Berlin, 1996.

[34] Mitchel, T.: *Machine Learning.* McGraw Hill, 1997.

[35] Moscato, P.: *Memetic Algorithms: A Short Introduction.* In: Corne, D., M. Dorigo und F. Glover (Hrsg.): *New Ideas in Optimization*, S. 219–234. McGraw Hill, Berkshire, England, 1999.

[36] Neumann, P. G.: *Practical Architectures for Survivable Systems and Networks.* Techn. Ber., SRI International Computer Science Laboratory, 2000.

[37] Nolden, R., S. Meier, W. Tasin, B. Gehrmann, J. Nordin, J. Olsson, S. Heidrich, S. Bartel, J. Birch und F. Brettschneider: *KDevelop – An Easy to Use C/C++ IDE for Unix*, 1998–2003.

[38] Ottmann, T. und P. Widmayer: *Algorithmen und Datenstrukturen.* Spektrum, 1996.

[39] Raidl, G. R.: *An Efficient Evolutionary Algorithm for the Degree-Constrained Minimum Spanning Tree Problem.* In: Fonseca, C. et al. [12], S. 104–111.

[40] Raidl, G. R.: *EALIB Vers. 1.1 – Evolutionary Algorithm Library*, 2003. Institute of Computer Graphics and Algorithms, Vienna University of Technology, Vienna, Austria.

[41] Raidl, G. R. und J. Gottlieb: *On the Importance of Phenotypic Duplicate Elimination in Decoder-Based Evolutionary Algorithms*. In: Brave, S. und A. S. Wu (Hrsg.): *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, S. 204–211, Orlando, FL, 1999.

[42] Raidl, G. R. und I. Ljubić: *Evolutionary Local Search for the Edge-Biconnectivity Augmentation Problem*. Information Processing Letters, 82(1):39–45, 2002.

[43] Rechenberg, I.: *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.*. Frommann-Holzboog Verlag, Stuttgart, 1973.

[44] Reinelt, G.: *TSPLIB - A Traveling Salesman Problem Library*. ORSA Journal on Computing, 3(4):376–384, 1991.

[45] Schwefel, H. P.: *Numerical Optimization of Computer Models*. Wiley, Chichester, 1981.

[46] Sedgewick, R.: *Algorithmen*. Addison-Wesley, 1991.

[47] Tarjan, R. E.: *Depth First Search and Linear Graph Algorithms*. SIAM Journal of Computing, 1:146–160, 1972.

[48] Zhu, A., S. Khuller und B. Raghavachari: *A Uniform Framework for Approximating Weighted Connectivity Problems*. In: *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, S. 937–938, 1999.