technische universität
dortmund

Human Guided Optimization

Sophia Kardung

Algorithm Engineering Report
TR10-1-005
Juni 2010
ISSN 1864-4503

Fakultät für Informatik
Algorithm Engineering (LS 11)
44221 Dortmund / Germany
http://ls11-www.cs.uni-dortmund.de/

technische universität
dortmund

Diplomarbeit

Human Guided Optimization

Sophia Kardung

12. April 2010

Betreuer:
Frau Prof. Dr. Petra Mutzel
Herr Dr. Markus Chimani

Fakultät für Informatik
Algorithm Engineering (Ls11)
Technische Universität Dortmund
http://ls11-www.cs.uni-dortmund.de

Abstract

In dieser Diplomarbeit wird ein Programm namens *HuGO* entwickelt, mit dessen Hilfe Optimierungsprobleme benutzergesteuert optimiert werden können. Die Beteiligung des Benutzers am Prozess der Optimierung kann es ermöglichen, dass durch menschliche Intuition schnell gute Lösungen gefunden werden. Außerdem besitzt der Benutzer die Möglichkeit die Lösung nach seinem Wissen anzupassen und dadurch Restriktionen und Präferenzen einzubringen, die nicht in die Problemeingabe codiert werden können. Die Benutzerinteraktion besteht aus einfachen Aktionen auf einer graphischen Oberfläche, die das Problem möglichst übersichtlich darstellt.

HuGO ist modular aufgebaut, so dass es einfach möglich ist Plugins für beliebige Optimierungsprobleme zu realisieren. In dieser Arbeit wurde ein Plugin für die k-Schichten-Kreuzungsminimierung und für das Rucksackproblem entworfen. Der Optimierungsalgorithmus bietet zwei alternative Suchvarianten an. Die Erste ist eine einfach Greedy-Suche, während die zweite Suchvariante nach dem Simulated Annealing-Prinzip arbeitet und einen Tabu-Ansatz integriert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel wird das Thema *Human Guided Optimization* eingeführt und motiviert. Anschließend wird ein kurzer Überblick über den Aufbau dieser Arbeit gegeben. Im letzten Abschnitt werden einige Grundlagen, erläutert, die im späteren Verlauf dieser Arbeit gebraucht werden.

1.1 Human Guided Optimization

Die Optimierung von schwierigen Problemen ist eines der wichtigsten Gebiete in der Informatik. Es gibt die verschiedensten exakten und heuristischen Ansätze zur Lösung von Optimierungsproblemen. Viele sind speziell für ein bestimmtes Problem entwickelt worden und optimieren dieses. Andere Algorithmen, wie zum Beispiel allgemeine Suchheuristiken, können auf jedem Optimierungsproblem ausgeführt werden.

In dieser Diplomarbeit möchte ich mich jedoch mit einer Idee beschäftigen, die in der Literatur noch nicht ausgiebig behandelt wurde, der *Human Guided Optimization*. Ein Ansatz der benutzergesteuerten Optimierung ist in sehr vielen Algorithmen zu finden. Oft muss oder kann der Benutzer einige Parameter, die die Optimierung lenken, setzen. Insbesondere bei Suchheuristiken kann der Benutzer oft eine Reihe von Parametern angeben und dadurch die Optimierung erheblich verbessern, wenn die Parameter geeignet gewählt wurden. Auch durch die Wahl eines Stopkriteriums, wird ein gewisser Einfluss auf die Optimierung genommen. Ziel ist es allerdings direkten Einfluss auf die Optimierung zu nehmen, um nicht nur die Güte oder die Laufzeit beeinflussen zu können, sondern die Lösung direkt. Um dieses Ziel zu erreichen, kann dem Benutzer des Optimierungsalgorithmus die Möglichkeit gegeben werden, Lösungen zu verändern und die Optimierung zu fokussieren.

In dieser Diplomarbeit wird eine solche Steuerung der Optimierung durch Veränderungen an Lösungen durch den Benutzer realisiert. Nach dem Vorbild des in Kapitel 2.1 beschriebenen Programms *HuGS*, wird ein Framework mit dem Namen *HuGO* entwickelt, mit

dessen Hilfe eine erweiterte Steuerung der Optimierung möglich ist. Zusätzlich können die Parameter des Optimierungsalgorithmus gesetzt werden. Ein Ablaufschema einer benutzergesteuerten Optimierung ist in Abbildung 1.1 dargestellt.

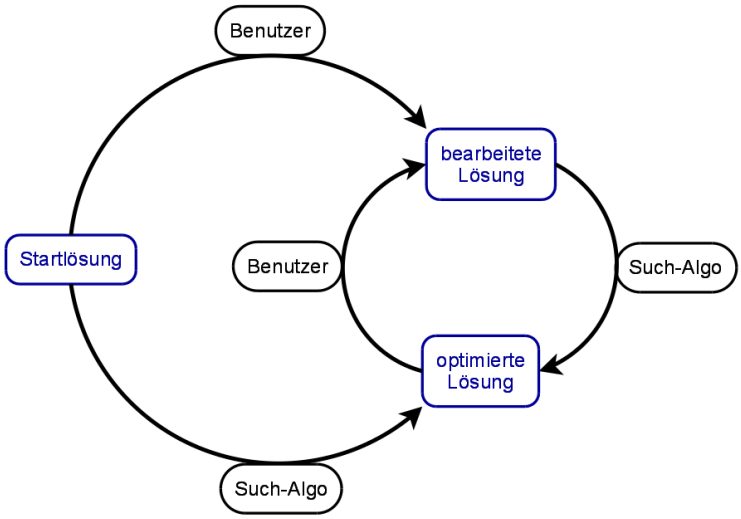

Abbildung 1.1: Ablaufschema einer Optimierung in *HuGO*

Es gibt einige andere problemspezifische Programme, die diesen Ansatz realisiert haben. Das in Kapitel 2.2 vorgestellte Programm *Foldit*, optimiert beispielsweise die Faltung von Proteinen durch Benutzersteuerung. Das in Kapitel 2 kurz erwähnte Programm *GLIDE* realisiert eine Benutzersteuerung durch interaktives Setzen von Restriktionen. Ein anderes Beispiel ist das Programm *DCAPS*, dass mit Hilfe von Benutzerinteraktionen ein Ablaufplan von Operationen zur Beladung eines Shuttles berechnet.

Es ist nicht Ziel dieser Diplomarbeit für ein spezielles Problem eine benutzergesteuerte Optimierung zu entwickeln. Statt dessen soll, wie oben beschrieben, ein Framework realisiert werden, in das Plugins für beliebige Optimierungsprobleme einfach integriert werden können. Zwei Plugins, die das Problem der k-Schichten-Kreuzungsminimierung (*CrossMin*) und das Rucksackproblem (*BackPack*) optimieren, werden in dieses Framework integriert.

1.2 Motivation

Es gibt eine große Menge an Suchalgorithmen für die verschiedenen NP-schweren Optimierungsprobleme. Viele sind speziell für ein Optimierungsproblem implementiert, andere sind allgemeine Suchheuristiken. Ebenso gibt es diverse exakte Algorithmen, die unter Ausnutzung verschiedener Eigenschaften versuchen praktisch eine gute Laufzeit zu erreichen.

Diese Algorithmen haben jedoch eines gemeinsam, der Anwender muss sein gesamtes Wissen, über das zu lösende Problem, in der Problemeingabe codieren. Möchte er den Algorithmus nicht anpassen ist es oft nicht möglich Restriktionen einzuhalten. So wird oft nur eine vereinfachte Version des eigentlichen Problems optimiert. Die Optimierung kann in vielen Suchheuristiken durch Parameter beeinflusst werden, die der Benutzer ändern kann. In der Regel ist allerdings eine gute Kenntnis der Heuristik und deren Arbeitsweise nötig um die Parameter sinnvoll setzen zu können. Ein anderes Problem ist die Beschreibung der Kostenfunktion. Alle diese Algorithmen benötigen eine genaue Kostenfunktion, die die Güte einer Lösung bestimmt und die optimiert wird. In der Praxis ist es bei komplexen Problemen oft nicht möglich eine Kostenfunktion exakt anzugeben, so dass sie der Realität entspricht. Der Algorithmus optimiert dadurch nicht zum realen Optimum hin, sondern nur zu dem, das die Kostenfunktion vorgibt. Ist die Kostenfunktion schlecht gewählt, wird selbst der beste Algorithmus keine, für den Benutzer akzeptablen Lösungen, finden.

Die Idee der *Human Guided Optimization* setzt genau an diesen Problemen an. Durch die direkte Beteiligung des Benutzers am Optimierungsprozess können die beschriebenen Probleme umgangen werden. Restriktionen und Präferenzen, die nicht in der Problemeingabe codiert werden konnten, können direkt in die Lösungen integriert werden. Außerdem kann, durch eine direkte Steuerung der Optimierung in die gewünschten Richtungen, bei einer ungünstigen Kostenfunktion eine für den Benutzer akzeptable Lösung berechnet werden.

Weiterhin hat sich gezeigt, dass Benutzer, die mit einem Problem vertraut sind, bei guter visueller Darstellung, oft Strukturen entdecken um eine Lösung gezielt zu verbessern. Hier haben sie die Möglichkeit diese Verbesserung direkt zu integrieren und anschließend mit der automatischen Optimierung fortzufahren. So ist es möglich die Suche erheblich zu beschleunigen. Durch den modularen Aufbau des Frameworks von *HuGO*, ist es außerdem vergleichbar einfach ein neues Plugin für ein Optimierungsproblem zu schreiben.

1.3 Aufbau der Arbeit

Diese Arbeit ist in sechs Kapitel unterteilt. Im ersten Kapitel, der Einleitung, werden die Grundlagen zu dieser Arbeit gelegt. Das Konzept der *Human Guided Optimization* wird vorgestellt und motiviert. Die beiden Optimierungsprobleme, k-Schichten-Kreuzungsminimierung und das Rucksackproblem, die als Plugins implementiert werden, werden definiert. Weiterhin wird der Simulated Annealing Algorithmus vorgestellt, der im Optimierungsalgorithmus von *HuGO* realisiert wird. Im nächsten Kapitel werden zwei Programme vorgestellt, die ebenfalls das Konzept der *Human Guided Optimization* realisieren. Das erste vorgestellte Programm *HuGS* lieferte die Grundidee dieser Diplomarbeit, während das zweite Programm *Foldit* ein sehr gut entwickeltes Programm ist, welches zeigt das Human Guided Optimization praktisch relevant sein kann. Im dritten Kapitel wird, dass in

dieser Diplomarbeit entwickelte Programm *HuGO*, vorgestellt. Zuerst wird der Aufbau des allgemeine Frameworks beschrieben. Das darauf folgende Kapitel stellt das Handbuch für *HuGO* dar, indem alle Funktionen, die der Benutzer ausführen kann, erläutert werden. Dort werden auch die pluginspezifischen Funktionen für die beiden Plugins beschrieben. Im ersten Teil der fünften Kapitels wird die allgemeine Entwicklung eines Plugins am Beispiel des realisierten *BackPack*-Plugins, welches das Rucksackproblem optimiert, beschrieben. Im zweiten Teil wird die Implementierung des CrossMin-Plugins und seine erweiterten Funktionen beschrieben. Im letzten Kapitel findet sich das Fazit zu dieser Diplomarbeit.

1.4 Grundlagen

In den nächsten zwei Abschnitten werden zwei Optimierungsprobleme vorgestellt, die k-Schichten-Kreuzungsminimierung und das Rucksackproblem. Für diese beiden Probleme wurden in *HuGO* Plugins realisiert, die diese optimieren. In den folgenden Abschnitten werden diese Optimierungsprobleme allgemein definiert und es werden einige bekannte Lösungsmöglichkeiten erläutert. Eine genaue Beschreibung der Realisierung der entsprechenden Plugins für *HuGO*, ist für die k-Schichten-Kreuzungsminimierungin in Kapitel 5.2 und für das Rucksackproblem in Kapitel 5.1 zu finden. Im dritten Abschnitt findet sich eine Beschreibung des heuristischen Optimierungsverfahres Simulated Annealing, das in dem in dieser Diplomarbeit entwickelten Programm *HuGO* als Optimierungsalgorithmus, zum Einsatz kommt.

1.4.1 k-Schichten-Kreuzungsminimierung

Das Problem der k-Schichten-Kreuzungsminimierung ist ein Optimierungsproblem, dessen Ziel es ist, die Anzahl der Kreuzungen zwischen den Kanten eines Graphen zu minimieren. Gegeben ist ein Graph $G = (V, E) = (V_1, V_2, ..., V_l, E)$ mit $V = V_1 \cup V_2 \cup ... \cup V_l$ dessen Knoten auf l Schichten angeordnet sind. Dabei gilt $V_1 \cap V_2 \cap ... \cap V_l = \emptyset$. Die Knoten der Schicht V_i sind auf einer horizontalen Linie mit Y-Position $k - i$ angeordnet und Kanten verbinden zwei Knoten durch eine gerade Linie.

Eine Lösung einer k-Schichten-Kreuzungsminimierungsinstanz kann durch l Permutationen $\pi_1, ..., \pi_l$ der Knoten in jeder Schicht repräsentiert werden. Knoten können die Positionen in ihrer Schicht dabei beliebig ändern. Ziel ist es für jede Schicht eine Permutation der Knoten zu berechnen, so dass die Anzahl der Kantenkreuzungen minimal ist. Die k-Schichten-Kreuzungsminimierung ist NP-schwierig. [15]

Motivation

Die Minimierung von Kreuzungen ist einer der wichtigsten Schritte beim automatischen Zeichnen von Graphen. Automatisches Graphenzeichnen ist in einigen Bereichen von

großer Bedeutung. In der Softwareentwicklung kommen zum Beispiel immer wieder große Diagramme vor, die sich oft ändern und es kostet sehr viel Zeit, diese manuell anzuordnen. In Bereichen wie Projektmanagement oder den Sozialwissenschaften sind ebenfalls oft große Graphen mit vielen Informationen zu finden. Um die Informationen und Zusammenhänge aus einem großen Graphen erkennen zu können, ist eine gute übersichtliche Zeichnung außerordentlich wichtig. Eine kleine Anzahl an Kreuzungen trägt dabei maßgeblich zu einer guten Zeichnung bei, denn Kreuzungen erschweren das Erkennen von Verbindungen sehr. [6]

Bekannte Verfahren

In den meisten praktischen Verfahren wird die k-Schichten-Kreuzungsminimierung auf eine Reihe von Optimierungen mit $k = 2$ reduziert. Dazu wird erst einmal eine Preprocessingphase ausgeführt, in der zusätzliche sogenannte Dummyknoten eingefügt werden. Für jede Schicht L_i werden Dummyknoten in den Kanten, die L_i nur kreuzen ohne einen Knoten auf dieser Schicht zu bilden, eingefügt. Anschließend gilt für jede Kante (u, v) $u \in V_i$ und $v \in V_j$ mit $i - j = \|1\|$ und $0 < i, j \leq l$. Alle Kanten verbinden also Knoten, die sich auf direkt untereinander liegenden Schichten befinden. Nach diesem Preprocessing beginnt das Iterieren. Dazu werden für alle $i = 1, ..., k - 1$ die Schichten L_i und $L_i + 1$ betrachtet, L_i wird fixiert und die Knotenpermutation von $L_i + 1$ wird optimiert. Anschließend wird die Reihenfolge von i umgekehrt und die Schichten werden rückwärts durchlaufen, wobei jeweils die untere Schicht fixiert und die obere permutiert wird. Diese Durchläufe werden solange abwechselnd wiederholt, bis sich die Knotenpositionen nicht mehr ändern.

Da das Problem der k-Schichten-Kreuzungsminimierung mit einer fixierten Schicht und für $k = 2$ NP-schwierig ist, werden dafür oft Heuristiken eingesetzt. Im folgenden beschäftigen wir uns also mit der 2-Schichten-Kreuzungsminimierung mit einer fixierten Schicht. Es wird eine der Heuristiken vorgestellt, die bei Experimenten sehr gut abgeschnitten hat und eine sehr kleine Laufzeit hat, die Barycenter Heuristik. Diese Heuristik weist jedem Knoten einen Barycenterwert zu, der der durchschnittliche Wert der Positionen seiner adjazenten Nachbarn ist. Anschließend werden die Knoten nach ihrem Barycenterwert sortiert und danach auf der Schicht positioniert. Sei n die Anzahl der Knoten auf der zu permutierenden Schicht, gilt $n \leq 4$ berechnet die Barycenter Heuristik die optimale Lösung. Die Barycenter Heuristik kann in Laufzeit $O(|E| + |V_2| log |V_2|)$ mit $|V_2|$ als Anzahl der Knoten in der zu permutierenden Schicht, realisiert werden. [15]

Für die 2-Schichten Kreuzungsminimierung mit einer fixierten Schicht gibt es einen exakten Algorithmus, der bis zu einer Größe von ca. 60 Knoten pro Schicht noch in kurzer Zeit berechenbar ist. Dazu wird das Problem in eine Linar Ordering Probleminstanz transformiert, welche mit einem Branch-and-Cut Verfahren für bis zu 60 Elemente innerhalb

einer Sekunde exakt lösbar ist. Dieses Verfahren wurde 1984 von Grötschel, Jünger und Reinelt entwickelt und 1995 neu implementiert. [7]

Betrachtet man nun die Analysen die in [15] durchgeführt wurden, die dieser exakte Algorithmus bei der k-Schichten-Kreuzungsminimierung durch Iterieren nach dem oben beschriebenen Prinzip liefert, stellt man fest, dass die Barycenter Heuristik besonders für dünne Graphen teilweise besser abschneidet als das exakte Verfahren. Außerdem fällt auf, dass besonders bei dünnen Graphen, die Lösungen aller Verfahren sehr weit vom Optimum entfernt sind.

Eine Heuristik für die k-Schichten-Kreuzungsminimierung, die nicht auf der 2-Schichten Kreuzungsminimierung aufbaut, wurde 1999 von Matuszewski, Schönfeld und Molitor entwickelt und 2001 von Günther, Schönfeld, Becker und Molitor verbessert, um die Laufzeit zu reduzieren.[6]

1.4.2 Rucksackproblem

Bei diesem Problem ist ein Rucksack gegeben, der ein bestimmtes Gewicht tragen kann. Ziel ist es, aus einer gegebenen Menge an Gegenständen, eine Auswahl zu treffen, die in den Rucksack passt und größtmöglichen Wert hat. Formal wird das maximale Gewicht des Rucksacks mit M bezeichnet. Die Anzahl der Gegenstände wird mit n bezeichnet und sie werden von 1 bis n nummeriert. Der i-te Gegenstand besitzt ein Gewicht w_i und einen Wert v_i, außerdem definiert die binäre Variable x_i, ob der Gegenstand sich im Rucksack befindet oder nicht.

$$x_i = \begin{cases} 1 & \text{wenn sich der Gegenstand } i \text{ im Rucksack befindet} \\ 0 & \text{sonst} \end{cases}$$

Für jede Lösung muss die Bedingung

$$\sum_{i=1}^{n} w_i v_i \leq M$$

gelten, während

$$\sum_{i=1}^{n} x_i v_i]$$

maximiert wird.
Das Rucksackproblem ist NP-vollständig, denn das Rucksackproblem ist ein Spezialfall des SUBSET-SUM Problems, wenn für alle Elemente gilt $w_i = v_i$. Da das SUBSET-SUM Problem NP-vollständig ist, ist auch das Rucksackproblem NP-vollständig. [13]

Motivation

Praktische Anwendungen für das Rucksackproblem zu finden ist nicht schwer, da sie zahlreich vorhanden sind. Ein Händler auf dem Markt kann zum Beispiel nur ein sehr begrenztes Warenangebot ausstellen. Er muss also eine Zusammenstellung an Waren finden,

die seinen Profit maximieren. Jeder Artikel nimmt einen bestimmten Platz in der Warenauslage ein und bringt bei Verkauf einen gewissen Profit. Die Berechnung des Rucksackproblems mit den genannten Eingaben, würden den Profit des Händlers maximieren. Allerdings unter der Voraussetzung, dass alle Waren verkauft werden. Die Eingabe sollte hier so manipuliert sein, dass nicht von jeder Ware beliebige Stückzahlen viele ausgewählt werden dürfen, sonst könnte das berechnete Angebot sehr eintönig ausfallen.

In einem anderen Fall könnte eine Person das Ziel haben, ihr Vermögen oder Teile ihres Vermögens, zu investieren. Ihr steht ein Betrag von M zur Verfügung, der investiert werden kann und es gibt n verschiedene Möglichkeiten den Betrag zu investieren. Jede Investitionsmöglichkeit bietet einen bestimmten Profit und benötigt eine gewisse Menge Geld als Anlagesumme. Wird das Rucksackproblem mit den genannten Eingaben berechnet, liefert es die besten Investitionen die für diese Person möglich sind. Auch in diesem Beispiel sind nicht alle Faktoren, wie z.B. das Risiko und die Laufzeit der Investitionen, berücksichtigt. [13]

In einem letzten Beispiel möchte eine Hilfsorganisation ein Frachtschiff in eine weit entfernte Krisenregion verschiffen. Aufgrund der Kosten kann nur ein Frachtschiff verschickt werden. Hilfsgüter sind allerdings in ausreichender Menge im heimischen Lager vorhanden. Nun muss aus den n vorhandenen Hilfsgütern eine Menge ausgewählt werden, die in das Frachtschiff mit maximaler Kapazität M passt und den Menschen in der Krisenregion maximale Hilfe bietet. Jede Hilfsware hat eine gewisse Größe und bringt den Menschen einen bestimmten Nutzen. Die Optimierung des Rucksackproblems auf diesen Daten liefert die Menge an Hilfsgütern, die den Menschen am meisten Hilfe liefert.

Die Beispiele für das Rucksackproblem sind vielseitig, aber es fällt auf, dass in den meisten praktischen Anwendungen sehr viele Faktoren berücksichtigt werden müssen, das Grundproblem bleibt aber gleich. Das Rucksackproblem stellt weiterhin ein Teilproblem einiger anderer Algorithmen dar.

Bekannte Verfahren

Das Rucksackproblem ist eines der am besten untersuchten Optimierungsprobleme und es gibt diverse Heuristiken und exakte Algorithmen für dieses Problem. Weiterhin gibt es eine Reihe von Varianten des klassischen oben definierten Rucksackproblems für die es wiederum eine Reihe von Algorithmen gibt. Eine der einfachsten Heuristiken für das oben definierte Rucksackproblem ist, der im folgenden beschriebene, Greedy-Algorithmus. Die Gegenstände werden absteigend nach ihrem Nutzen sortiert. Der Nutzen des Gegenstands n_i ist definiert als $b_i = \frac{v_i}{w_i}$. Angefangen mit dem Gegenstand mit größtem Nutzen, werden solange Gegenstände in den Rucksack gepackt, bis der nächste Gegenstand in der sortierten Liste das maximale Gewicht des Rucksacks überschreiten würde. Dieser Algorithmus hat eine Laufzeit von $O(n \log n)$, die aus dem Sortieren der Gegenstände resultiert. Außerdem

ist die Lösung des Greedy-Algorithmus mindestens halb so gut wie die optimale Lösung. Der Algorithmus ist damit eine 2-Approximation. [13]

Im folgenden wird der exakte Algorithmus von Nemhauser und Ullmann, der 1969 entwickelt wurde, vorgestellt. Dieser Algorithmus baut auf Pareto-optimalen Punkten im Suchraum auf. Unter einem Pareto-optimalen Punkt versteht man einen Punkt, der nicht von anderen Punkten dominiert wird. Vergleicht man zwei Punkte im Suchraum, so dominiert einer den anderen, wenn dieser in allen Kriterien besser ist. Im Rucksackproblem ist ein Punkt im Suchraum definiert durch das Gewicht und den Wert der Teilmenge an Gegenständen, die eine Lösung definieren und die der Punkt repräsentiert. Der Algorithmus von Nemhauser und Ullmann berechnet die Menge aller Pareto-optimalen Punkte und wählt anschließend aus dieser Menge den Punkt mit höchstem M nicht überschreitenden Gewicht aus. Dieser repräsentiert die optimale Lösung.
Zur Berechnung der optimalen Lösung wird unter den Punkten (1)-(4) die Menge aller Pareto-optimalen Punkte berechnet. Anschließend wird in dieser Menge das Optimum gesucht.

1. Erzeuge eine Liste L_0, die den Punkt$(0, 0)$ enthält, also den leeren Rucksack repräsentiert.

2. Erzeuge nun aus der Liste L_i eine Liste L'_i, in der alle Punkte aus L_i, verschoben um das Gewicht $w_i + 1$ und den Wert $v_i + 1$ der $i + 1$-ten Gegenstands, enthalten sind.

3. Erzeuge die Liste $L_i + 1$ durch Verschmelzung der Listen L_i und L'_i, wobei Punkte entfernt werden, die von Punkten aus der jeweils anderen Liste dominiert werden.

4. Bis die Liste L_n erzeugt wurde, weiter bei (2).

5. Durchsuche die Liste L_n nach dem Punkt p mit höchstem Gewicht, das M nicht überschreitet aus.

6. Gib den Punkt p als optimale Lösung zurück.

Die Laufzeit ist im Worst Case allerdings exponentiell, denn im ungünstigsten Fall können alle Punkte des Suchraums Pareto-optimal sein. In der Praxis macht die Anzahl der Pareto-optimalen Punkte in der Regel nur einen Bruchteil der gesamten Punkte des Suchraums aus. So ist es in der Praxis möglich, Instanzen mit mehreren tausend Elementen in kurzer Zeit zu lösen. [3]

1.4.3 Simulated Annealing

Um NP-vollständige Probleme zu lösen können sogenannte randomisierte Suchalgorithmen eingesetzt werden, die den Suchraum, der aus einer exponentiellen Anzahl an möglichen Lösungen besteht, durchsuchen. Randomisierte Suchalgorithmen sind in der Regel nicht

schneller und besser als problemspezifische Algorithmen, die die Strukturen der Probleme ausnutzen, aber sie liefern schnell und ohne das große problemspezifische Anpassungen nötig sind akzeptable Ergebnisse. In der Praxis werden oft Algorithmen mit genau diesen Eigenschaften gesucht. Simulated Annealing ist eine randomisierte Suchheuristik, in der in jedem Schritt aus der aktuellen Lösung randomisiert eine Neue erzeugt wird, die mit einer gewissen Wahrscheinlichkeit akzeptiert wird.

Es gibt allerdings für polynomielle Laufzeiten keinerlei allgemeine Gütegarantien. Simulated Annealing kann, bis es das globale Optimum gefunden hat, sogar eine wesentlich längere Laufzeit benötigen als das vollständige Durchsuchen des Suchraums benötigen würde. In der Praxis hat sich allerdings gezeigt, dass diese Strategie sehr oft schnell zu brauchbaren Ergebnissen kommt. [8] [19]

Grundlagen

Einer der einfachsten randomisierten Suchalgorithmen ist die einfache, zufällige Suche. Hier werden zufällig neue Lösungen erzeugt und die beste gefundene Lösung wird bei Beenden des Algorithmus zurückgegeben. Theoretisch hat die zufällige Suche natürlich exponentielle Laufzeit, da sie einen exponentiellen Suchraum betrachten muss und auch praktisch ist diese Methode oft nicht effektiv.

Ein weiter entwickelter Suchalgorithmus ist der sogenannte Metropolis Algorithmus der im Jahr 1953 von Metropolis et al. entwickelt wurde. Sei $0,1^n$ der Suchraum und f eine Funktion, die jedem Punkt im Suchraum einen Wert zuordnet, der minimiert werden soll, außerdem sei T ein gegebener Temperaturparameter. Der Ablauf des Algorithmus sieht wie folgt aus:

1. Erzeuge eine Startlösung $s \in 0,1^n$
2. Erzeuge durch eine lokale Änderung eine neue Lösung s' aus s
3. Falls $f(s') \geq f(s)$ setze $s = s'$
   falls $f(s') < f(s)$ setze $s = s'$, mit Wahrscheinlichkeit $\exp(\frac{f(s)-f(s')}{T})$
4. Weiter bei (2.), solange bis Abbruchkriterium erfüllt ist.

Theoretisch ist die Laufzeit des Metropolis Algorithmus exponentiell, doch in der Praxis hat sich gezeigt, dass er, wenn die Temperatur geeignet gesetzt ist, durchaus schnell gute Lösungen erreicht werden. Neben der geeigneten Temperatur muss für jedes Optimierungsproblem definiert werden, wie die lokale Änderung der Lösung in Schritt 2 definiert ist. Standardmäßig wird ein zufälliges Bit der Bitfolge s, die die Lösung repräsentiert, geflippt. [19]

Motivation

Die Grundidee der Simulated Annealing Suchstrategie stammt aus der Physik. Dort werden geschmolzene Metalle zu Feststoffen abgekühlt. Wenn diese Abkühlung ausreichend langsam statt findet, erreicht der Feststoff einen Zustand mit minimalen Energiegehalt. Während des Abkühlungsprozesses können sich die Teilchen erst noch fast beliebig bewegen, mit Abnahme der Temperatur wird diese Bewegungsfreiheit immer weiter eingeschränkt, bis es zum Stillstand kommt. Im Verlauf der Abkühlung ändert sich der Energiegehalt ständig und kann sich auch verschlechtern. Der Simulated Annealing Algorithmus in der Informatik funktioniert sehr ähnlich. Abhängig von einer Temperatur, die im Laufe der Zeit sinkt, wird die Probleminstanz ständig verändert. Um so niedriger die Temperatur wird, um so geringer ist die Wahrscheinlichkeit das Änderungen, die die Probleminstanz verschlechtern, zugelassen werden. Durch dieses Konzept erhofft man sich, lokale Optima im Lösungsraum durch Verschlechterungen wieder verlassen zu können und am Ende der Optimierung, das globale Optimum zu erreichen oder ihm mindestens sehr nahe zu kommen. [5]

Details

In der Simulated Annealing Variante des Metropolis Algorithmus ist die Temperatur T nicht konstant, sondern verändert sich während der Optimierung. Dadurch wird, die im vorherigen Abschnitt beschriebene, Abkühlung realisiert. Die Abkühlung wird formal durch die folgenden Parameter definiert:

- Eine Temperaturfunktion $T(n)$, die die Temperatur für den n-ten Schritt der Optimierung liefert, muss definiert werden. Durch $T(n)$ wird definiert, wie die Temperatur in jedem Schritt geändert wird.
- Die initiale Temperatur $T(1)$ mit der die Optimierung startet, muss gesetzt werden.
- Die Funktion $N(t)$ liefert für die Temperatur t die Anzahl der Iterationen, die ohne Temperaturänderung stattfinden. Diese Funktion muss definiert werden.
- Ein Stopkriterium für den Optimierung muss gesetzt werden.

Betrachten wir als erstes die Temperaturfunktion $T(n)$. Eine einfache Möglichkeit ist eine lineare Funktion, bei der die Temperatur nach jedem Schritt um einen konstanten Wert gesenkt wird. Eine lineare Temperaturfunktion kann für eine Konstante c folgendermaßen definiert werden $T(n)_{lin} = T(n-1) - c$. Eine andere Möglichkeit ist eine exponentielle Senkung der Temperatur. Die Temperaturfunktion $T(n)_{exp} = T(n-1)f$ mit einen konstanten Faktor f mit $0 \leq f \leq 1$ realisiert dies. Durch die Funktion $T(n)_{exp}$ sinkt die Temperatur zu Beginn schneller als am Ende. So wird einerseits ein zu langes Verweilen bei einer hohen Temperatur verhindert und andererseits wird die Temperatursenkung

in Richtung Nullpunkt immer langsamer. Untersuchungen haben gezeigt, dass ein langes Verweilen bei hoher Temperatur nicht zu besseren Ergebnissen führt, da annähernd jede Veränderung zugelassen wird und die Suche einer einfachen zufälligen Suche gleicht. Es gibt noch eine Reihe weiterer Temperaturfunktionen, auf die hier allerdings nicht weiter eingegangen wird, da sie im Verlauf dieser Arbeit nicht genutzt werden.

Nach der Temperaturfunktion betrachten wird nun die Funktion $N(t)$, die für jeden Wert von T, bestimmt wie viele Iterationen die Temperatur unverändert bleibt. Auch hier gibt es verschiedene Ansätze. Oft wird $N(t)$ allerdings konstant auf einen Wert gesetzt. Die initiale Temperatur, sowie das Stopkriterium werden, in dem in dieser Diplomarbeit entwickelten Programm, vom Anwender gesetzt, da diese Werte problemspezifisch sind. [8]

Kapitel 2

Benutzergesteuerte Optimierungsprogramme

In diesem Kapitel werden einige benutzergesteuerte Optimierungsprogramme vorgestellt. Zuerst betrachten wir ein Programm, dass ein eigentlich sehr gut erforschtes Problem optimiert. Im sogenannten *DATA-CHASER Automated Planner/Scheduler (DCAPS)* Programm wird ein Schedulingproblem optimiert. Genauer wird ein Scheduling erstellt, das die Datenübertragung von Operationen auf ein Space Shuttle regelt. Dieses Problem ist sehr viel komplexer als das normale Schedulingproblem und ist deshalb, mit den bekannten Methoden nur schwer lösbar. Die Operationen sind meistens sehr lang sehr komplex und es gibt diverse Abhängigkeiten untereinander. Weiterhin müssen die erschwerten Bedingungen durch sehr eingeschränkte Energie und Rechenleistung beachten werden. *DCAPS* ist ein interaktives Programm bei dem der Benutzer das berechnete Scheduling verändern kann. Anschließend repariert der Algorithmus Fehler, die dabei entstanden sein können. Ebenso ist eine weitere Optimierung und die Eingabe von Restriktionen möglich. *DCAPS* ist ein sehr problemspezifisches Programm, das zeigt, dass die Idee der benutzergesteuerten Optimierung gerade bei schwierigen Problemen durchaus erfolgversprechend ist. [4]

Nun kommen wir zu einem anderen Programm das interaktives Zeichnen von kleinen Graphen unterstützt. Im sogenannte *Graph Layout Interactive Diagram Editor (GLIDE)* kann der Benutzer Restriktionen für die einzelnen Komponenten interaktiv anpassen. Außerdem sind eine Reihe von sogenannten *Visual Organization Features (VOFs)* vorgegeben. *VOFs* sind Makros zur Zeichnung von kleinen Teilkomponenten eines Graphen. Diese werden zusammen mit den Restriktionen von einem Algorithmus erzwungen und so entsteht die Darstellung eines Graphen. Außerdem kann der Benutzer die Komponenten eines Graphen manuell verschieben. [16]

Nachdem wir diese zwei Programme nur kurz betrachtet haben, beschäftigen wir uns in den folgenden zwei Abschnitten ausführlich mit den beiden benutzergesteuerten Optimierungsprogrammen *HuGS* und *Foldit*.

2.1 HuGS

Human Guided Search (HuGS) ist ein Programm zur interaktiven Optimierung von Problemen. Die Optimierung wird visualisiert und kann durch den Benutzer gesteuert werden. HuGS ist modular aufgebaut, es besteht einerseits aus einem allgemeinen Framework und andererseits aus einem Plugin, welches das Optimierungsproblem definiert. Es werden vier Plugins für verschiedene Optimierungsprobleme vorgestellt, die mit Hilfe von *HuGS* optimiert werden können. Weiterhin wird die Möglichkeit neue Plugins zu realisieren erläutert. Die Software stand der Forschung einige Zeit durch eine freie Forschungslizenz zur Verfügung, doch zur Zeit ist HuGS nicht mehr frei verfügbar. Die Idee dieser Diplomarbeit zum Thema *Human Guided Optimization* ist durch *HuGS* entstanden.

2.1.1 Grundstruktur

Die Grundstruktur eines Plugins in *HuGS* ist aus folgenden Komponenten konstruiert. Aus der Eingabe wird ein *Probleminstanz* erstellt, die zu optimieren ist. Jede Probleminstanz besteht unter anderem aus einer Menge sogenannter *Elemente*. Ziel der benutzergesteuerten Suche ist es, die beste *Lösung* für das Problem zu finden. Dies stellt die Ausgabe der Optimierung dar. Zur Durchsuchung des Suchraums werden bisher gefundene Lösungen durch sogenannte *Moves* verändert, so dass neue Lösungen entstehen. Ein *Move* ist also eine Veränderung der Lösung, die eine Teilmenge der Elemente der Probleminstanz verändert und so zu einer neuen Lösung führt.

Diese Strukturen muss jedes Plugin implementieren, das in das Framework von *HuGS* integriert werden soll. Die Realisierung der richtigen Moves für ein Plugin ist dabei entscheidend für die Effektivität des Optimierungsalgorithmus. Durch eine Sequenz von Moves muss es möglich sein jeden Punkt im Suchraum zu erreichen, außerdem muss ein einzelner Move effizient erzeugt und ausgeführt werden können.

2.1.2 Beispielanwendungen

Für die aktuelle Version von *HuGS* wurden vier Beispielanwendungen realisiert, um die interaktive Optimierung zu testen. Diese werden in den folgenden Abschnitten kurz vorgestellt. Außerdem wurde für die erste Version von *HuGS* mit *SimpleSearch* Variante ein Plugin für das sogenannte *Capacitated Vehicle Routing with Time Windows (CVRTW)* entwickelt, dieses soll hier aber nicht weiter erläutert werden. Neben den vier Anwendungen die in den folgenden Abschnitten beschrieben werden, wurden im Laufe der Zeit von verschiedenen Personen noch weitere Plugins entwickelt. Diese sollen hier nur kurz erwähnt werden. Das *Packing*-Plugin ist ein 2-dimensionales BinPacking Problem, bei dem Rechtecke, so angeordnet werden müssen, das sie eine gewisse Breite nicht überschreiten. Im *Labeling*-Plugin müssen Beschriftungen von Punkten möglichst ohne Überlappungen an-

geordnet werden. Eine Scheduling-Variante in der zwei Faktoren optimiert werden, ist im Plugin *Paintshop* realisiert. Dieses Plugin zeigt, dass multikriterielle Optimierung für benutzergesteuerte Optimierungsalgorithmen kein großes Problem darstellt, denn der Benutzer kann die Gewichtung der verschiedenen Faktoren steuern. Ein weites Plugin namens *Transport*, zeigt, dass *HuGS* auch für sehr große und komplexe Probleme geeignet sei kann. Im *Transport*-Plugin wird eine Variante des Vehicle Routing Problems optimiert. Es wurde eine Instanz erzeugt, der Flugpläne für den Transport von Waren über einen Zeitraum von einem Jahr berechnet. [11]

Crossing

Die erste Beispielanwendung ist das sogenannte *Crossing*-Plugin. Es entspricht dem in Kapitel 1.4.1 vorgestellten Problem der k-Schichten-Kreuzungsminimierung.

Die Definition, der im vorherigen Abschnitt vorgestellten Grundstruktur, ist hier denkbar einfach. Jedes Element stellt einen Knoten dar, der sich auf einer bestimmten Schicht im Graphen befindet. In der Probleminstanz sind alle Knoten des Graphen enthalten und können durch Kanten verbunden werden. Eine Lösung einer Probleminstanz ergibt sich durch die Positionierung der Knoten auf ihrer jeweiligen Schicht, woraus eine Kreuzungsanzahl resultiert. Um Lösungen zu ändern, gibt es einen Move, der die Positionen zweier Knoten auf einer Schicht tauscht.

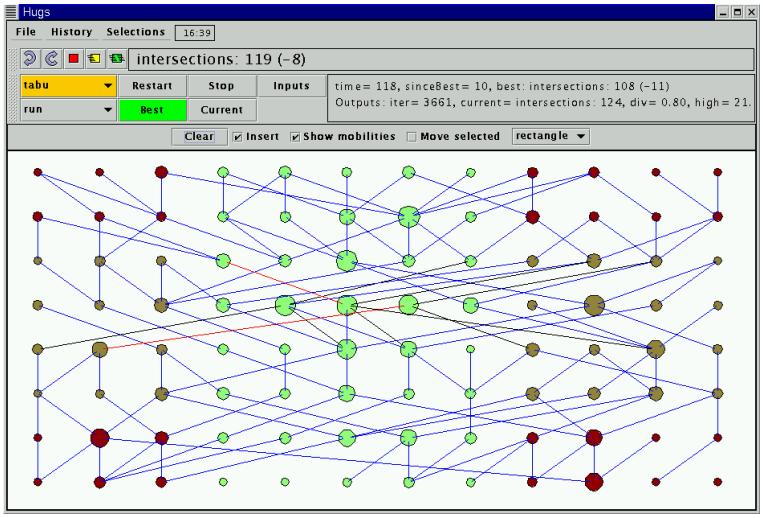

Abbildung 2.1: *HuGS* - *Crossing*-Plugin [9]

Delivery

Das zweite Problem, das sogenannte *Delivery* Problem, ist eine Abwandlung des Traveling-Salesman Problems, indem eine Route über alle Orte minimiert wird. In dieser Variante

wird die Voraussetzung, dass alle Orte besucht werden müssen, gestrichen und in der Problemeingabe wird eine Wegbeschränkung angegeben, die nicht überschritten werden darf. Ziel ist es, so viele Orte wie möglich zu besuchen, ohne die Wegbeschränkung zu überschreiten. Außerdem ist es möglich die einzelnen Orte unterschiedlich zu gewichten. Jedem Ort, der einen Kunden darstellt, wird dazu eine Anzahl an Paketen zugewiesen, die dort ausgeliefert werden müssen. Weiterhin wird der benötigte Weg minimiert, wobei ein ausgeliefertes Paket mehr, immer besser ist als ein kürzerer Weg, solange die Wegbeschränkung eingehalten wird.

Die im vorangegangenen Abschnitt erläuterte Grundstruktur, wird wie folgt auf das *Delivery* Problem übertragen. Jeder Kunde wird durch ein Element repräsentiert, und befindet sich an einer bestimmten Position auf der Karte. Eine Probleminstanz besteht aus der Menge von Kunden die beliefert werden können, sowie einer Wegbeschränkung. Eine Lösung für eine Probleminstanz besteht aus einer Sequenz von Kunden, die Teilmenge der Menge an Kunden der Probleminstanz ist. Diese Sequenz stellt eine Route dar, nach deren Reihenfolge die Kunden beliefert werden. Um Lösungen zu verändern gibt es zwei verschiedene Typen von Moves. Ein SwapMove kann zwei Kunden auf der Route tauschen. Wird auf der aktuellen Route Kunde x vor Kunde y besucht, ist diese Reihenfolge nach der Ausführung eines SwapMoves auf x und y umgekehrt. Die Positionen in der Reihenfolge der anderen Kunden ändern sich dabei nicht. In einem anderen Typ Move werden Kunden aus der Route entfernt bzw. hinzugefügt. In Abbildung 2.2 ist eine Probleminstanz des *Delivery*-Plugins in *HuGS* zu sehen.

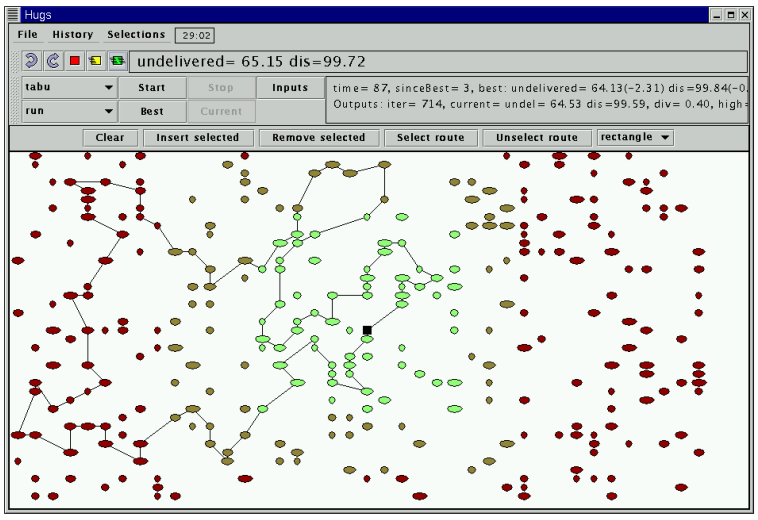

Abbildung 2.2: *HuGS* - *Delivery*-Plugin [9]

Jobshop

Das *Jobshop* Problem ist eine Scheduling Variante. Es müssen n Jobs auf m Maschinen ausgeführt werden, während jeder Job in m Aufgaben unterteilt ist, die in einer bestimmten Reihenfolge ausgeführt werden müssen. Jede Aufgabe eines Jobs muss auf einer anderen der m Maschinen ausgeführt werden, jede der m Maschinen führt also genau eine Aufgabe jedes Jobs aus. Außerdem kann jede Maschine nur eine Aufgabe zur selben Zeit erledigen. Gesucht ist ein Scheduling, dass einen Ablaufplan für jede Maschine enthält, so dass sich keine zwei Aufgaben auf einer Maschine überschneiden und alle Jobs vollständig erfüllt werden. Die Zeit die dieses Scheduling benötigt, bis die letzte Aufgabe abgeschlossen ist, wird minimiert.

Nun betrachten wir wieder die Definition der Grundstruktur für dieses Optimierungsproblem in *HuGS*. Eine Probleminstanz beinhaltet eine Liste aller Maschinen, eine Liste aller Jobs, die jeweils aus einer Menge an Aufgaben in einer bestimmten Reihenfolge bestehen und eine Zeitangabe für jede Aufgabe. Die Aufgaben stellen die Elemente dieses Plugins dar, die in den Jobs zusammengefasst sind. Eine Lösung für eine Probleminstanz besteht aus einem Ablaufplan für jede Maschine. Zur Änderung von Lösungen ist ein sogenannter *InsertionMove* implementiert. Ein *InsertionMove* schiebt eine Aufgabe auf einer Maschine an eine frühere oder spätere Ausführungsposition. Um einen freien Platz für die zu ändernde Aufgabe zu schaffen, werden Aufgaben, die die neue Position blockieren, ebenfalls verschoben. Im automatischen Suchalgorithmus werden nur Verschiebungen von Aufgaben um einen Platz, also ein Tausch der Position mit benachbarten Aufgaben, ausgeführt. Der Benutzer kann manuell jedoch jede Aufgabe an einen beliebigen Platz im Ablaufplan der entsprechenden Maschine schieben.

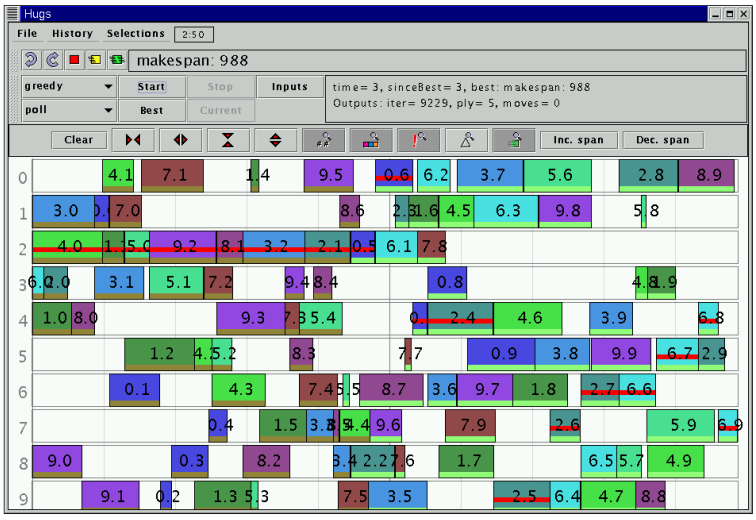

Abbildung 2.3: *HuGS* - *Jobshop*-Plugin [9]

Protein

Diese Beispielanwendung optimiert das sogenannte *Protein* Problem, welches eine Vereinfachung des Proteinfaltungsproblems darstellt. Das Proteinfaltungsproblem wird in Abschnitt 2.2 genauer erläutert, außerdem wird ein Programm vorgestellt, dass dieses Problem optimiert. Die vereinfachte Version, die in *HuGS* betrachtet wird, erhält als Eingabe eine Sequenz, die aus wasserlöslichen und fettlöslichen Elementen besteht. Ziel ist es, diese Sequenz in einem 2-dimensionalen-Raster so auszurichten, dass die Anzahl der Paare von fettlöslichen Elementen maximiert wird. Dabei dürfen sich Elemente nicht überlappen und die Sequenz muss erhalten bleiben. Dieser Zustand entspricht dem minimal energetischen Zustand des repräsentierten Protein.

Um die vereinfachte Version des Proteinfaltungsproblems in *HuGS* zu realisieren, wird folgende Grundstruktur erstellt. Eine Probleminstanz besteht aus einer Sequenz von Aminosäuren. Die Elemente sind die einzelnen Aminosäuren, die im 2-dimensionalen Raster angeordnet werden müssen. Die Lösung einer Probleminstanz, ist ein 2-dimensionales Raster, in dem die Position jeder Aminosäure der Sequenz gespeichert ist. Die Reihenfolge der Sequenz muss dabei erhalten bleiben. Zur Veränderung von Lösungen wurde ein sogenannter *PullMove* entwickelt. Ein *PullMove* repositioniert zwei in der Sequenz benachbarte Aminosäuren um, indem sie auf benachbarte freie Rasterpositionen geschoben werden. Dabei können in der Sequenz der Aminosäuren Lücken entstehen, die durch weitere *PullMoves* wieder geschlossen werden müssen. Im ungünstigsten Fall müssen alle Aminosäuren repositioniert werden, um eine entstanden Lücke wieder zu schließen. In der Regel sind allerdings nur einige *PullMoves* nötig. [12]

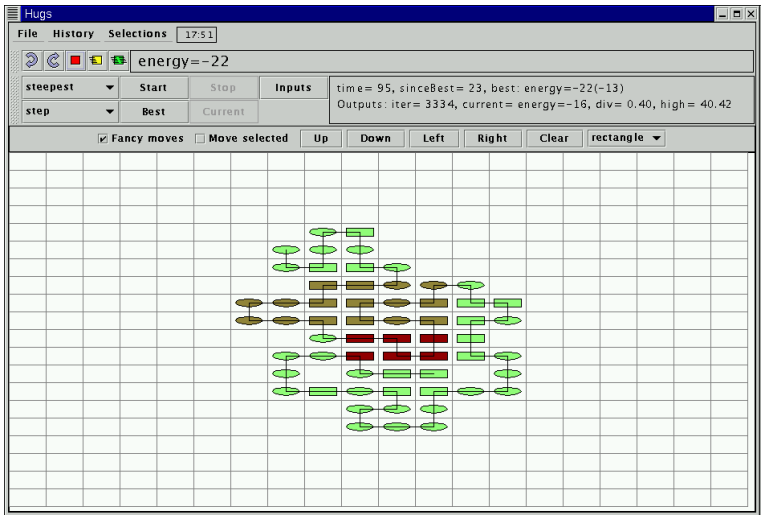

Abbildung 2.4: *HuGS* - *Protein*-Plugin [9]

2.1.3 Mobilities

Um die Möglichkeiten der Steuerung des Suchalgorithmus durch den Benutzer zu erweitern, wurde die Idee entwickelt, dies durch sogenannte Mobilities zu realisieren. Jedem Element eines Problems wird eine Mobilität zugewiesen, die niedrig, mittel oder hoch sein kann. Die drei Möglichkeiten werden für den Benutzer visuell durch Färbung der Elemente in den Farben rot, gelb und grün dargestellt und sind somit intuitiv und übersichtlich erkennbar. Elemente mit hoher Mobility werden durch Moves beliebig verändert, während Elemente mit mittlerer Mobility nur eingeschränkt verändert werden können. Eingeschränkte Veränderung bedeutet, dass Moves, die nur Elemente mit mittlerer Mobility verändern, nicht ausgeführt werden dürfen. Elemente mit mittlerer Mobility können nur geändert werden, wenn sie in Verbindung mit einem Element mit hoher Mobility in einem Move enthalten sind. Elemente mit niedriger Mobility werden durch den Suchalgorithmus nicht verändert. Durch Setzen der Mobilities kann die Optimierung von Benutzerseite aus gesteuert werden, da der Suchraum durch gezieltes Setzen der Mobilities erheblich verkleinert werden kann. So fokussiert sich der Suchalgorithmus in der Optimierung auf den Suchraum der erreichbar ist.

In *HuGO* wird das Konzept der Mobilities weitestgehend übernommen und mit einigen Zusatzfunktionen ausgestattet. Genaueres zur Realisierung der Mobilities in *HuGO* ist in Kapitel 3.2.2 zu finden.

2.1.4 Manuelle Änderung von Lösungen

Eine weitere Möglichkeit zur Steuerung der Optimierung ist das manuelle Ändern einer Lösung. Der Benutzer passt die aktuelle Lösungen durch ändern der Elemente nach seinen Ideen an und startet den Suchalgorithmus anschließend neu. Diese manuellen Änderungen sind insbesondere dann sinnvoll, wenn der Suchalgorithmus sich in einem lokalen Optimum befindend und dieses nur schwer wieder verlassen kann. Durch den Neustart mit einer veränderten Lösung wird das lokale Optimum verlassen und der Suchalgorithmus kann unter Umständen neue bessere Lösungen finden. Die genaue Definition einer manuellen Änderung einer Lösung ist pluginspezifisch. Im oben beschriebenen *Delivery*-Plugin besteht das manuelle Ändern aus hinzufügen und entfernen von Lösungen aus der Route. Im *Crossing*-Plugin können Knoten durch Verschieben innerhalb einer Schicht ihre Position ändern. Die manuellen Änderungen können durch Setzen der Mobilities sehr gut ergänzt werden, denn so kann einerseits verhindert werden, dass der Suchalgorithmus manuelle Änderungen, die die Lösung verschlechtert haben, sofort wieder rückgängig macht. Andererseits kann der Benutzer gezielt Restriktionen und Präferenzen, die aus der Realität über das Problem bekannt sind und die in der Eingabe nicht codiert werden konnten, einfügen. Eine mögliche Präferenz im *Delivery*-Plugin wäre zum Beispiel, dass eine Reihe von Kunden besonders wichtig sind, obwohl sie in der aktuellen Routenplanung vielleicht

keinen großen Wert haben. In Anbetracht zukünftiger Aufträge, ist es für den Benutzer trotzdem sehr wichtig, dass diese Kunden beliefert werden. Manuelles Hinzufügen dieser Kunden zur Route und anschließendes Setzen auf niedrige Mobility garantiert, dass diese Kunden immer beliefert werden.

Je nach Plugin kann es möglich sein ungültige Lösungen durch manuelle Moves zu erzeugen. So kann zum Beispiel die Route im *Delivery*-Plugin durch manuelle Moves länger werden, als die Wegbeschränkung es erlaubt. Wird der Optimierungsalgorithmus dann gestartet, erzeugt er allerdings nach Möglichkeit wieder eine gültige Lösung.

Um die Effektivität und Komfortabilität beim Bearbeiten von Lösungen zu gewährleisten, enthält HuGS noch zwei weitere Funktionen. Zum einen ist es jederzeit möglich Änderungen durch wiederholtes Betätigen der *Undo*-Funktion rückgängig zu machen. Außerdem wurde ein Verlauf implementiert, in dem alle möglichen berechneten Lösungen gespeichert werden. So kann der Benutzer verschiedene Ansätze zur Optimierung probieren, ohne das aktuelle Lösungen gelöscht werden.

2.1.5 Suchalgorithmus

In der ersten Version von *HuGS*, wird als Suchalgorithmus eine einfache Greedy-Suche implementiert. Diese bildet in jedem Schritt eine Liste aller Moves, die zu allen benachbarten Lösungen führen und wählt die erste aus, die die Lösung verbessert. Eine zweite Variante, namens *steepest-descent*, durchläuft die Liste der erzeugten Moves einmal komplett und wählt den Move aus, der die Lösung am meisten verbessert. Es kann dabei unterschiedliche Arten von Moves geben und der Benutzer kann auswählen welche erzeugt werden sollen. Die Definition der Moves ist allerdings pluginspezifisch. In dieser Version ist es die Aufgabe des Benutzers die Suche aus lokalen Optima zu leiten, denn mit diesen beiden Varianten wird ein einmal erreichtes lokales Optimum nicht wieder verlassen, da keinerlei verschlechternde Änderungen zugelassen werden. Diese erste Version wurde *Human Guided Simple Search (HuGSS)* genannt.

In einer zweiten Version wurde der Algorithmus *Human Guided Tabu Search* entwickelt. Die Tabusuche ist ein heuristischer Ansatz zur Durchsuchung eines großen Suchraums auf dem eine Nachbarschaftsbeziehung definiert ist. Um zu verhindern, dass die Suche im Kreis läuft, wird ein Tabuliste von Moves gespeichert, die nicht ausgeführt werden dürfen. Nach einer bestimmten Anzahl an Schritten werden die Moves wieder aus der Tabuliste entfernt. Die Tabusuche in HuGS geht dabei wie folgt vor: Es wird eine Liste aller legalen Moves erstellt, um den besten Move auszusuchen. Der ausgesuchte Move kann die Lösung auch verschlechtern. Unter legalen Moves werden alle Moves verstanden, die weder die Lösung ungültig machen, noch Elemente bewegen deren Mobility es nicht erlaubt. Der ausgesuchte Move wird auf der aktuellen Lösung ausgeführt und falls dies eine beste Lösung liefert, wird diese gespeichert. Nun werden die Mobilities aller Elemente aktualisiert. Dabei wird

allen Elementen die durch den letzten Move verändert wurden, eine mittlere Mobility zugewiesen. So wird verhindert, dass die ausgeführte Veränderung im nächsten Schritt wieder rückgängig gemacht wird. Nach einer bestimmten Anzahl an Schritten, wird den Elementen wieder eine hohe Mobility zugewiesen. In HuGS wird die Tabuliste also nicht explizit gespeichert, sondern implizit durch das Vergeben von Mobilities. Ein weiterer Mechanismus namens *Diversify* wurde realisiert, um den Suchraum möglichst weitreichend zu durchsuchen und nicht nur in eine beschränkte benachbarte Teilmenge des Suchraums. Um dies zu erreichen, müssen bevorzugt Elemente ausgewählt werden, die bisher selten verändert wurden. Dazu wird Elementen, die bisher sehr häufig verändert wurden, eine mittlere Mobility zugewiesen. Sowohl für die Tabu-Funktion als auch für die *Diversify*-Funktion, gibt es je einen Parameter zur Steuerung, der vom Benutzer gesetzt werden kann.

Ein Problem an diesem Vorgehen ist, dass in jedem Schritt eine sehr große Menge an Moves erzeugt und durchsucht werden muss. Dadurch wird viel Laufzeit in Anspruch genommen, um die Menge der Moves zu erzeugen, aus der nur einer ausgeführt wird. Durch Realisierung der Tabu und *Diversify*-Funktion mit Hilfe von Mobilities, ist das Vorgehen des Suchalgorithmus für den Benutzer leicht nachzuvollziehen. Allerdings führt das ständige Wechseln der Mobilities der Elemente auch zu einer gewissen Unübersichtlichkeit. Es ist so nur schwerer möglich manuell Mobilities gezielt zu verteilen, um die Suche zu fokussieren und Präferenzen und Restriktionen einzubauen und insbesondere die Folgen dieser manuell gesetzten Mobilities zu erkennen.

2.1.6 Pluginerstellung

Um ein neues Plugin für *HuGS* zu erstellen, muss die in Kapitel 2.1.1 beschriebene Grundstruktur und eine graphische Darstellung implementiert werden. Der Rest des Frameworks ist modular aufgebaut, so dass es für alle Plugins, die die Schnittstellen korrekt implementieren, funktionsfähig ist. Die zu implementierenden Komponenten, sowie die Schnittstellen, die beachtet werden müssen, werden im Folgenden grob erläutert. Zuerst einmal muss die Grunddatenstruktur des Plugins implementiert werden. Dabei muss eine Probleminstanz definiert werden, sowie die daran enthaltenen Elemente. Weiterhin muss die Lösung einer Probleminstanz definiert werden. Um zwei Lösungen im Suchalgorithmus vergleichen zu können, muss die Klasse *Score* implementiert werden, die eine Methode *isBetter* besitzt, in der zwei Lösungen bzgl. ihrer Güte verglichen werden. Außerdem muss mindestens ein Move definiert werden, sowie die Erzeugung der Moves, die ebenfalls pluginspezifisch ist.

Zur visuellen Darstellung des Plugins muss eine graphische Oberfläche implementiert werden. Diese wird modular eingebaut und muss folgende drei Eigenschaften implementieren.

- Die grafische Oberfläche des Plugins muss dem Framework manuelle Änderungen, die der Benutzer an der aktuellen Lösungsinstanz vornimmt, in Form von Moves mitteilen.
- Es muss eine Update-Funktion implementiert werden, die die Änderungen durch Moves im Suchalgorithmus, sowie Änderungen der Mobilities, aktualisiert.
- Der Benutzer muss Elemente auswählen und abwählen können und diese müssen dem Framework mitgeteilt werden. Dadurch wird die Änderung der Mobilities realisiert, die im Framework stattfindet.

Es empfiehlt sich eine Markierung der, durch Moves veränderten, Elemente einzuführen, damit der Benutzer nach jedem Schritt des Suchalgorithmus die vorgenommene Änderung auf einen Blick erkennen kann. Ohne diese Markierung ist es schon bei kleinen Instanzen fast unmöglich die genauen Änderungen nachzuvollziehen.

Die graphische Darstellung der Elemente und ihrer Verknüpfungen ist für den Erfolg eines Plugins außerordentlich wichtig. Bei einer unübersichtlichen visuellen Darstellung des Problems wird der Benutzer nicht in der Lage sein, den Optimierungsalgorithmus sinnvoll zu steuern. [10]

2.1.7 Experimente

Von den Entwicklern von *HuGS* wurden einige Experimente durchgeführt, um die Effizienz von benutzergesteuerter Optimierung zu testen. Bei Experimenten mit der ersten *Human Guided Simple Search* Variante für das *CVRTW* Problem zeigte sich deutlich, dass eine Optimierung mit dem Suchalgorithmus ohne Benutzersteuerung sehr viel schlechter ausfiel als ein benutzergesteuerter Optimierungslauf. Da der Suchalgorithmus in dieser Variante eine einfache Greedy-Suche ist, sind diese Ergebnisse nicht allzu überraschend. Im anschließenden Vergleich mit gängigen Suchheuristiken landete die benutzergesteuerte Optimierung mit HUGSS allerdings auch im guten Mittelfeld. Bei diesen Versuchen optimierten jeweils vier mit HuGS trainierte Personen für eine bestimmte Zeit eine Probleminstanz. Anschließend wurden die erreichten Werte mit denen der Suchheuristiken verglichen. [1]

Die 2002 entwickelte Tabusuche für HuGS wurden ebenfalls in einer Reihe von Versuchen getestet. Die ungesteuerte Tabusuche wurde mit verschiedenen Werten des Parameters für die *Diversify*-Funktion getestet. Dabei stellte sich heraus, dass durchschnittlich ein mittlerer Wert die besten Ergebnisse liefert. Im *Crossing*-Plugin brachte allerdings ein höherer Wert bessere Ergebnisse. Die Einstellungen dieses Wertes sollten also pluginspezifisch vorgenommen werden. Außerdem wurde die ungesteuerte Greedysuche mit der ungesteuerten Tabusuche mit einer geeigneten *Diversity* verglichen. Dabei schnitt die Tabusuche, wie erwartet, wesentlich besser ab. Ein Vergleich der ungesteuerten Tabusuche

mit Standardalgorithmen für das *Crossing* Problem zeigte, dass die Tabusuche durchschnittlich nur eine geringfügig schlechtere Güte erreichte.

Um zu zeigen, dass die eigentliche Idee der Human Guided Optimization erfolgreich ist, wurde die ungesteuerte Tabusuche mit der benutzergesteuerten Optimierung mit Hilfe der Tabusuche verglichen. Dazu wurde die Zeit, die die ungesteuerte Tabusuche braucht, um die selbe Güte zu erreichen, wie eine 10-minütigen benutzergesteuerte Optimierung, bestimmt. Getestet wurden die Plugins *Crossing* und *Delivery*. Das Ergebnis zeigt, dass für beide Plugins durchschnittlich über eine Stunde ungesteuerte Tabusuche nötig ist um die gleiche Güte zu erreichen. Weiterhin zeigte sich, dass sogar 10 Minuten benutzergesteuerte Optimierung mit der Greedysuche erst von durchschnittlich 29 Minuten ungesteuerter Tabusuche geschlagen werden.

Die Experimente haben gezeigt, dass das Konzept der *Human Guided Optimization* funktioniert und so oft schneller bessere Ergebnisse gefunden werden. Wenn man dazu die Möglichkeit des einfachen Einbaus von Präferenzen des Benutzers bedenkt, kann die Human Guided Optimization ein durchaus sehr sinnvolles Werkzeug sein. [9]

2.2 Foldit

Foldit ist ein Programm zur Optimierung von Proteinstrukturen durch den Benutzer mit Hilfe einiger algorithmischer Methoden, die aufgerufen werden können. Es wird im Internet frei angeboten und zielt darauf ab, dass die Benutzer auf spielerische Art und Weise Proteinfaltungsinstanzen lösen. Ursprünglich wurde ein Programm namens *Rosetta@home* entwickelt, das Proteinfaltungsprobleme automatisch optimiert. Da diese Optimierung sehr laufzeitintensiv ist, kann man die Wissenschaftler unterstützen, indem man *Rosetta@home* auf seinem privaten Rechner im Hintergrund laufen lässt. So können die Wissenschaftler auf die Rechenleistung von sehr vielen freiwilligen Benutzern zurückgreifen.

Einige Benutzer beobachteten dabei die Optimierungsabläufe des Algorithmus und fragten daraufhin die Entwickler, wieso der Optimierungsalgorithmus oft so ineffektiv vorgeht und teilten mit, dass sie selbst dabei anders vorgehen würden. So entstand die Idee, dem Benutzer die Möglichkeit zu geben in die Optimierung einzugreifen und es wurde das Programm *Foldit* entwickelt. Es zeigte sich, dass der Benutzer mit Hilfe einiger automatischer Optimierungsverfahren, die Proteinfaltung sehr gut unterstützen kann. Eine genauere Beschreibung des Problems der Proteinfaltung und des Programms *Foldit*, ist in den nächsten Abschnitten zu finden.

2.2.1 Proteinfaltung

Proteine sind Makromoleküle, die aus Aminosäuren aufgebaut sind und die verschiedenste Aufgaben im menschlichen Körper übernehmen. Aus den 20 Aminosäuren sind durch

die unterschiedliche Anordnungen tausende an Proteinen konstruierbar. Jede Aminosäure besteht aus einer Basis und einem Rest . Durch den Rest unterscheiden sich die einzelnen Aminosäuren voneinander, denn die Basis jeder Aminosäure ist identisch. Proteine bestehen immer aus einer langen Kette miteinander verbundener Aminosäuren. Dabei kann man zwischen dem Grundgerüst der aneinandergereihten Aminosäuren und den Seitenketten unterscheiden. Das Grundgerüst besteht aus den eben erwähnten Basen der Aminosäuren, während die Seitenketten den Rest der jeweiligen Aminosäure darstellen. Diese Aminosäureketten liegen im Körper allerdings nicht in einer geraden Kette vor, sondern sind auf eine ganz bestimmte Art und Weise im Raum angeordnet und in dieser Anordnung liegt der Schlüssel zur Funktion eines Proteins. Diese Anordnung eines Proteins nennt man Proteinfaltung und sie wird mit *Foldit* berechnet. Da die Faltung sehr vieler Proteine noch unbekannt ist, besteht großes Interesse an möglichst gut gefalteten Proteinmodellen.

Die Proteinfaltung ist wichtig um die Funktionsweise eines Proteins nachvollziehen zu können, um so künstliche Proteine zur Krankheitsbekämpfung herzustellen zu können und Krankheiten besser zu verstehen. Der HI-Virus beispielsweise besteht aus mehreren Proteinen, also einem Polyprotein, und bildet zusätzlich zwei Proteine, die bei der Replikation des Virus in den Zellen helfen. Je besser die genaue Funktionsweise dieses Virus verstanden wird, desto eher ist es möglich, eine Therapie zu entwickeln, die beispielsweise diese beiden Helferproteine schädigt und so die Replikation des HI-Virus verhindert.

Eine Möglichkeit die noch unbekannte Faltung eines Proteins zu bestimmen ist die Alignierung. Dazu betrachten wir die Struktur eines Proteins genauer. Wie bereits beschrieben, besteht ein Protein aus Aminosäuren, die in einer Kette angeordnet sind. Die einfachste Möglichkeit das Protein zu beschreiben ist, die Aminosäuren der Reihe nach in einer Sequenz aufzulisten. Diese Sequenz wird Primärstruktur genannt. Schon anhand der Primärstruktur eines Proteins kann probiert werden seine Faltung zu bestimmen. Dazu werden die Primärstrukturen verschiedener Proteine mit bekannter Faltung, mit der Primärstruktur des zu faltenden Proteins verglichen. Dieser Vergleich geschieht durch die sogenannte Alignierung. Dabei können Gaps (Lücken) in die Sequenzen eingefügt werden, um die Ähnlichkeit zu erhöhen. Um so mehr gleiche Aminosäuren sich an der selben Position in den beiden Sequenzen befinden, um so ähnlicher sind sich die zwei Proteine. Eine andere Möglichkeit zur Beschreibung eines Proteins ist die Sekundärstruktur. Hier werden die Aminosäuren 2-dimensional auf einem Raster angeordnet. In dem Plugin *Protein* von *HuGS*, das in Kapitel 2.1.2 beschrieben wurde, wird zum Beispiel die Sekundärstruktur eines Proteins unter vereinfachten Bedingungen optimiert.

Die Proteinfaltung ist abhängig von den verschiedenen Komponenten des Proteins, die verschiedene Eigenschaften haben. Es gibt beispielsweise hydrophobe Komponenten, die sich eher im inneren Bereich des gefalteten Proteins befinden, um so der, das Protein umgebenen Flüssigkeit, auszuweichen. Hydrophile Komponenten dagegen befinden sich

eher auf der äußeren Seite des gefalteten Proteins und brauchen größeren Abstand zu anderen Komponenten. Weiterhin gibt es einige Komponentenpaare die möglichst nah nebeneinander liegen sollten, während einige Paare weit von einander entfernt sein sollten. Dies sind nur einige Beispiele für Präferenzen der Proteine. Es gibt noch sehr viele weitere Strukturen die Restriktionen und Präferenzen bzgl. ihrer Position im Raum betreffend haben. Anhand dieser Bedingungen bestimmt *Foldit* eine Punktzahl, die beschreibt, wie gut eine Proteinfaltung ist. Ziel ist es diese Punktzahl zu maximieren.

2.2.2 Rosetta@home

Rosetta@home optimiert die Faltung von Proteinen automatisch und nutzt dazu die Rechenleistung der Computer der Benutzer. Es lädt die aktuell zu optimierende Probleminstanz aus dem Internet und optimiert die Proteinfaltung. Ziel ist es die Faltung des gegebenen Proteins vorherzusagen. Dies reale Faltung entspricht dabei der Faltung mit geringster Energie. Die Energie einer Faltung wird von den verschiedenen, im letzten Abschnitt erläuterten Präferenzen und Restriktionen der Aminosäurekomponenten bestimmt. Der Ablauf der Optimierung ist dabei folgender:

1. Starte mit einer geraden ungefalteten Kette.
2. Positioniere einen Teil der Kette um und erzeuge so eine neue Faltung.
3. Berechne die Energie der erzeugten Faltung.
4. Akzeptiere diese Positionierung oder verwerfe sie, abhängig von der Energiebilanz.
5. Wiederhole Schritt 2 bis 4 so lange, bis jeder Teil der Kette vielfach bewegt wurde.

Dieser Ablauf ist in zwei Phasen unterteilt. In der ersten Phase wird das Protein vereinfacht dargestellt und es finden Änderungen der groben Struktur statt. In der zweiten Phase finden nur noch kleine Veränderungen an dem exakt dargestellten Protein statt. Zu jedem Protein finden mehrere Versuche dieses Ablaufes statt, da die Veränderungen randomisiert berechnet werden. Jeder Lauf kann ein anderes und vielleicht besseres Ergebnis liefern. Die energieminimalste Lösung wird jeweils zurückgeschickt, um zentral die Beste aller Lösungen auszuwählen, die dann als Vorhersage der Proteinfaltung gilt.

Der Benutzer von *Rosetta@home* hat die Möglichkeit die aktuell vom Algorithmus berechneten Lösungen zu betrachten, aber es gibt keinerlei Möglichkeiten Einfluss auf die Optimierung zu nehmen. Um dies zu ermöglichen wurde, das im nächsten Abschnitt beschriebene Programm *Foldit*, realisiert. In Abbildung 2.5 ist die Darstellung der Proteinfaltung in *Rosetta@home* zu sehen. [17]

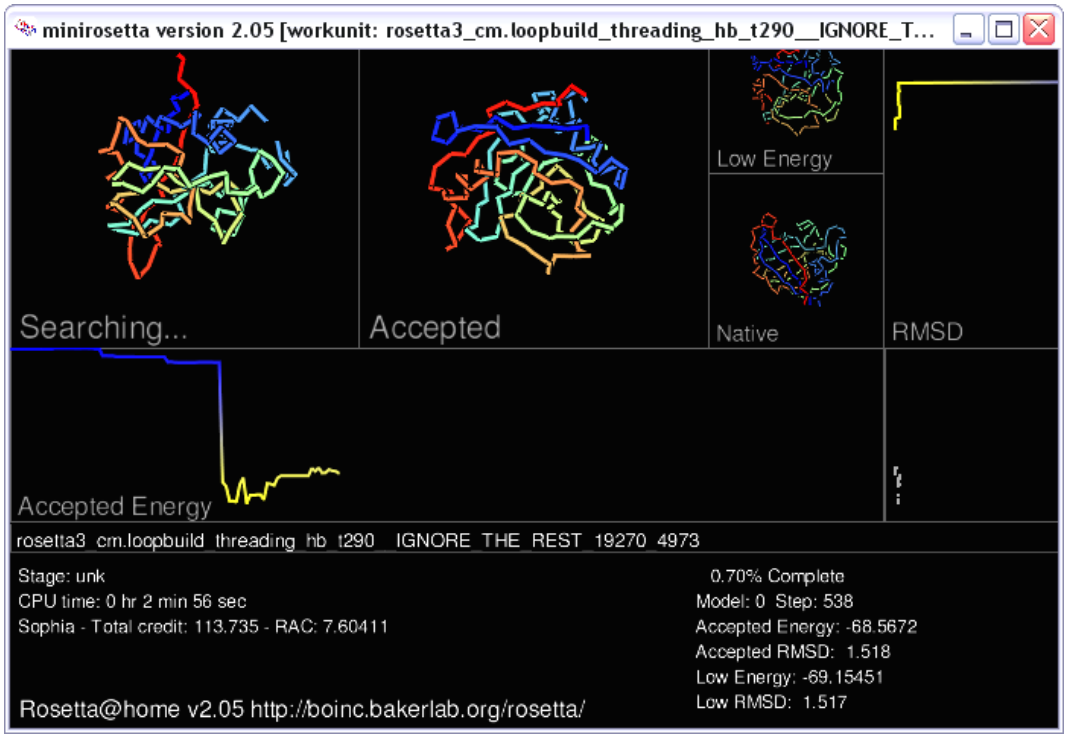

Abbildung 2.5: Beispiel einer Proteinfaltung in *Rosetta@home*

2.2.3 Foldit

Foldit bietet dem Benutzer die Möglichkeit aktiv in die Proteinfaltung einzugreifen. Die wichtigsten Funktionen zur manuellen und automatischen Optimierung sind im nächsten Abschnitt beschrieben. Das Programm ist als Spiel konzipiert, um möglichst viele Benutzer zu animieren, Proteinfaltungsinstanzen zu optimieren. Um die Motivation der Spieler lange zu erhalten, wurde ein Punktesystem entwickelt aus dem Bestenlisten erzeugt werden. Außerdem werden diverse Funktionen zur Entwicklung einer Community zur Verfügung gestellt. Darunter zum Beispiel ein Forum, eine Chatfunktion im Spiel, ein Blog und eine Facebook Fanseite, weiterhin ist es möglich Duelle gegen andere Benutzer zu spielen.

Die besten von den Benutzern optimierten Faltungen werden von Entwicklern von *Foldit* gesammelt und ausgewertet. Außerdem untersuchen die Wissenschaftler die Vorgehensweise der Benutzer bei der Optimierung. Ziel ist es so neue Erkenntnisse zu gewinnen, durch die die automatische Optimierung unter Umständen verbessert werden kann. Dazu werden regelmäßig neue Proteininstanzen zur Verfügung gestellt, die dann über einen gewissen Zeitraum optimiert werden können. In Abbildung 2.6 ist ein Proteinfaltungsinstanz in *Foldit* zu sehen.

[18]

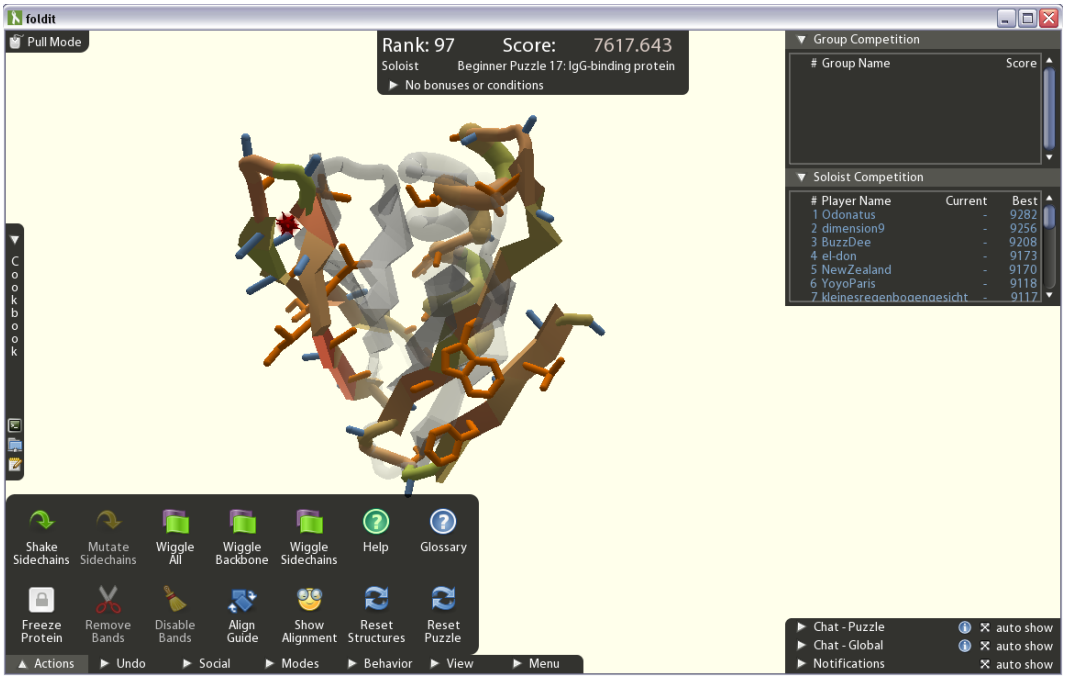

Abbildung 2.6: Beispiel einer Proteinfaltung in *Foldit*

Funktionen

Zur Optimierung der Proteinfaltung stehen dem Benutzer diverse automatische und manuelle Werkzeuge zur Veränderung der aktuellen Faltung zur Verfügung. Ein Benutzer der längere Zeit optimiert, entwickelt eigene Strategien wie er die verschiedenen Werkzeuge einsetzt. Dies ist wie oben erwähnt ein weiterer Punkt, den die Entwickler von *Foldit* untersuchen. Anhand der von den Benutzern entwickelten Optimierungsstrategien, können neue Automatismen entwickelt werden und in die automatische Proteinfaltung des Programms *Rosetta@home* eingebaut werden.
Im folgenden eine Liste aller wichtigen manuellen Funktionen die der Benutzer ausführen kann:

Nudge Jede Komponente des Proteins kann ausgewählt und verschoben werden. Die Verschiebung findet abhängig von den Wechselwirkungen der anderen Komponenten statt.

Tweak Strukturen die Helices oder Platten ähnliche sind können mit dieser Funktion gedreht werden.

Rubber Bands Zwischen zwei Komponenten kann ein sogenanntes *Rubber Band* gespannt werden. Anschließend werden diese Komponenten nach Starten der automatischen Optimierung (siehe *Wiggle*) zusammen gezogen. Es können beliebig viele *Rubber Bands* gesetzt werden.

Lock Jede Komponente kann fixiert werden, so dass sie ihre Position nicht mehr ändert.

Undo Die letzten Änderungen können rückgängig gemacht werden, außerdem können aktuelle Faltungen jederzeit gespeichert und geladen werden. Von jeder Optimierung wird ein Verlauf gespeichert.

Alingnment Bei Aktivierung dieser Funktion erscheint die Primärstruktur des Proteins und kann vom Benutzer mit den Primärstukturen anderer Proteine, dessen Faltung bekannt ist, verglichen werden. Finden sich viele Ähnlichkeiten zu einem bereits gefalteten Protein, kann der Benutzer durch Setzen von Gaps eine Alignierung erstellen. Mit Hilfe dieser Alignierung kann anschließend probiert werden, die Faltung des alignierten Proteins auf dieses zu übertragen. Die Übertragung der Struktur findet automatisch statt und funktioniert nicht immer.

Kommen wir nun zur Liste der Funktionen, die der Benutzer zwar starten und stoppen kann, die ansonsten aber automatisch optimieren.

Wiggle Diese Funktion, die die Grundstruktur der Proteinfaltung optimiert, ist wohl die wichtigste zur automatischen Optimierung. In Verbindung mit den oben beschriebenen *Rubber Bands* oder der *Lock*-Funktion ist es möglich diese Optimierung zu lenken. Fixierte Komponenten werden nicht verändert. Weiterhin kann diese Funktion auch nur auf dem Grundgerüst, nur auf den Seitenketten oder nur auf einzelnen Komponenten aufgerufen werden.

Shake Sidechains Um neben der Grundstruktur auch die Seitenketten des Proteins automatisch optimieren zu können, gibt es die Funktion *Shake Sidechains*. Diese Funktion kann ebenfalls lokal auf einer Komponente aufgerufen werden.

Rebuild Diese Funktion sucht nach komplett neuen Faltungen für die Grundstruktur des Proteins. Im Gegensatz zu *Wiggle* werden nicht nur kleinere lokale Veränderungen ausgeführt.

Diese Liste ist nicht vollständig, aber sie beschreibt kurz die wichtigsten Funktionen zur Optimierung der Proteinfaltung von *Foldit*. Neben den Möglichkeiten zur Optimierung der Faltung eines gegebenen Proteins, gibt es einen Designmodus, in dem ein Protein auch verändert werden kann. Auf diesen Modus soll hier allerdings nicht weiter eingegangen werden.

Kapitel 3

HuGO

In diesem Kapitel wird die Grundstruktur von *HuGO*, sowie die einzelnen Pakete aus denen *HuGO* zusammengesetzt ist, beschrieben. Das Paket *generalPlugin* stellt die Datenstruktur der Plugins dar. Es enthält eine Reihe von abstrakten Klassen deren Unterklassen ein neues Plugin realisieren. Im Paket *algo* sind alle Klassen, die für die Optimierung benötigt werden, enthalten. Die Ein- und Ausgabe befindet sich im Paket *io* und die graphische Oberfläche im Paket *ui*. Außerdem existiert noch das Paket *utils*, welches einige Hilfsklassen enthält. Ein vereinfachtes Klassendiagramm zur Struktur von *HuGO* ist in Abbildung 3.1 dargestellt.

3.1 Pluginstruktur

Jedes Plugin muss von den, im Paket *generalPlugin* enthaltenen abstrakten Klassen, erben und die vorgegebenen Methoden überschreiben. Dadurch ist es möglich ohne weitere Änderungen in den anderen Paketen, neue komplett funktionsfähige Plugins zu realisieren. Die Hauptklasse jedes Plugins ist die Klasse `gPlugin`, die alle für das Plugin benötigten Klassen, die im folgenden erläutert werden, enthält. Das Problem, das im jeweiligen Plugin zu optimieren ist, wird in der Klasse `gProblem` definiert. In der Klasse `gSolution` werden die ergänzenden Informationen, die für eine Lösung des Problems benötigt werden gespeichert. Alle Probleme bestehen aus sogenannten Knoten. Die konkrete Definition eines Knotens hängt von dem jeweiligen Plugin ab. Die Klasse `gProblem` verwaltet diese Knoten und je nach Plugin weitere Strukturen und Variablen, die benötigt werden. Knoten werden durch die Klasse `gNode` dargestellt. Ein Knoten besitzt eine Identitätsnummer, eine Position, einen Namen und eine sogenannte Mobility. Die Definition der Mobility ist im folgenden Abschnitt zu finden. Weitere pluginspezifische Variablen und Methoden werden in den Unterklassen ergänzt.

Der Optimierungsalgorithmus arbeitet mit sogenannten Moves, die eine Veränderung einer Lösung darstellen. Ein Move kann zum Beispiel die Verschiebung eines Knotens

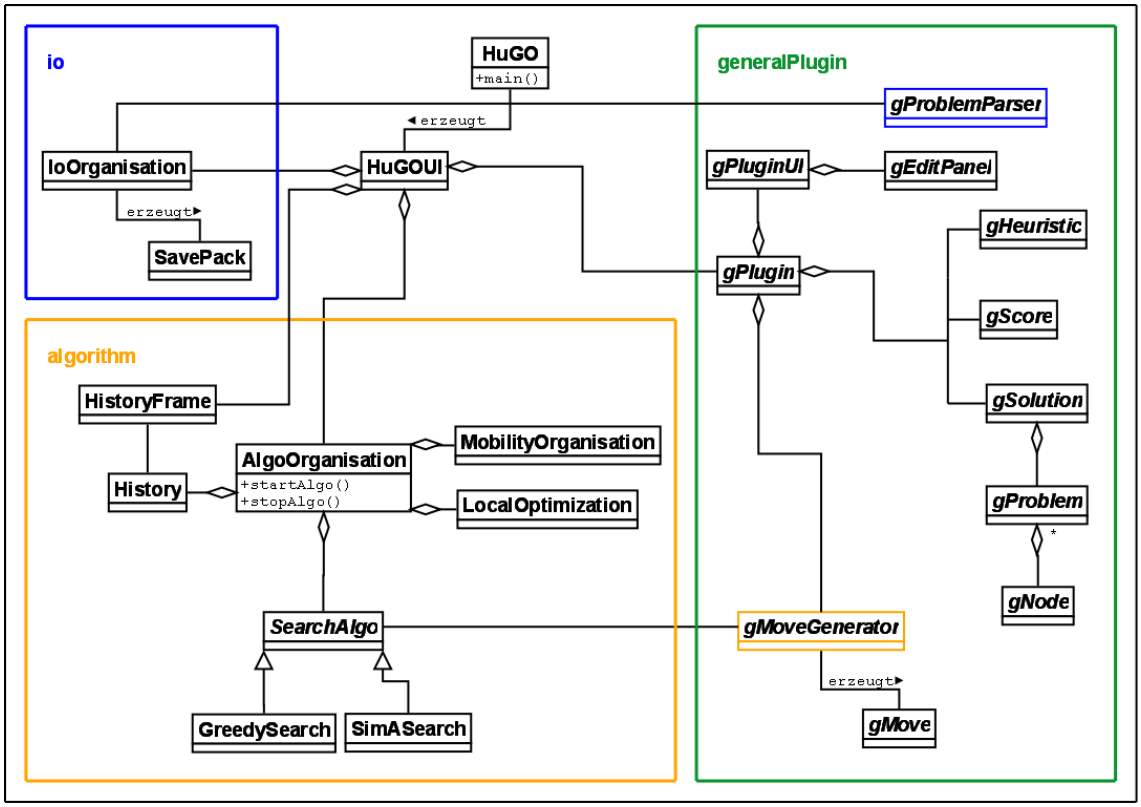

Abbildung 3.1: Vereinfachtes Klassendiagramm zur Darstellung der Grundstruktur aller wichtigen Klassen von *HuGO*

darstellen. Wie die Veränderung, die ein Move bewirkt, definiert ist, muss, in der von `gMove` erbenden Unterklasse in der zu überschreibenden Methode `doMove`, implementiert werden. Die Klasse `gMove` enthält außerdem die abstrakte Methode `createReverseMove`, die überschrieben werden muss. Sie gibt einen zum aktuellen Move komplementären Move zurück, der die Änderung des Moves rückgängig macht. Die Moves werden in der Klasse `gMoveGenerator` erzeugt. Um dies zu realisieren, muss die abstrakte Methode `generateNextMove` überschrieben werden, die einen zufälligen Move erzeugt und zurückgibt.

Die Klasse `gScore` realisiert durch abstrakte Methoden den Vergleich zweier Lösungen. Dadurch ist es dem allgemeinen Optimierungsalgorithmus erst möglich zwei Lösungen zu vergleichen. In der Klasse `gHeuristic` kann eine Heuristik für das jeweilige Plugin implementiert werden. So kann der Benutzer von HuGO sich eine Startlösung erzeugen, mit der er weiter optimieren kann. In der Klasse `gProblemParser` wird das Erzeugen eines Problems aus einem Eingabestring implementiert. Da *HuGO* keinerlei Funktionen zur Erstellung von Probleminstanzen bietet, ist die Möglichkeit des Einlesens einer Probleminstanz auf diese Art unverzichtbar. Das pluginspezifische graphische Interface wird in der Klasse `gPluginUI` erzeugt, während in der dazugehörigen Klasse `gEditPanel` die Zeichenfläche gefüllt wird. In der schon erwähnten Klasse `gProblem` befindet sich die Struktur

des eigentlichen Problems. Dazu gehört in erster Linie eine Liste von Knoten, die in jedem Plugin enthalten ist. Um den Zugriff auf diese zu ermöglichen, stehen eine Reihe von Methoden zur Verfügung, die einzelne Knoten oder die Liste aller Knoten zurückgeben. Sind für ein Plugin weitere Klassen notwendig, sollten diese der entsprechenden Unterklasse von `gProblem` hinzugefügt werden.

Ich habe mich, wie oben beschrieben, dafür entschieden die allgemeine Struktur eines Plugins, durch das Konzept der Vererbung vorzugeben. Eine alternative Möglichkeit wäre es, diese Struktur durch Interfaces darzustellen. Ich habe mich in diesem Fall für das Konzept der Vererbung entschieden, weil dem Entwickler des Plugins dadurch zwar etwas weniger Freiheiten bleiben, dafür aber eine ausgeprägtere Struktur an der man sich orientieren kann, vorhanden ist. Außerdem war es so möglich eine Reihe von allgemeinen Methoden in den abstrakten Klassen zu implementieren und so das Realisieren neuer Plugins zu vereinfachen.

3.2 Automatische Optimierung

Die Optimierungsfunktionen von *HuGO* sind so gestaltet, dass jedes Optimierungsproblem, für das ein Plugin implementiert wurde, ohne weitere Anpassungen optimiert werden kann. Es gibt außerdem die Möglichkeit, zusätzlich zu den zwei vorhandenen Optimierungsvarianten, Greedy und Simulated Annealing, eine neue Variante zu implementieren. In diesem Abschnitt wird zuerst die Realisierung des Verlaufs in *HuGO* beschreiben. Anschließend wird erläutert, wie das Konzept der Mobilities umgesetzt wurde, um dann anschließend den Optimierungsalgorithmus und die zwei implementierten Varianten zu erklären.

3.2.1 Verlauf

Im Verlauf können bisher berechnete Lösungen gespeichert werden. Während des Optimierungsablaufs werden auch einige Lösungen automatisch gespeichert. Die Anzahl dieser automatisch gespeicherten Lösungen ist allerdings begrenzt, damit es zu keinem zu hohen Speicherbedarf kommt. Die Anzahl der Lösungen die automatisch gespeichert werden können, kann vom Benutzer geändert werden. Ist die maximale Anzahl der Lösungen die gespeichert werden erreicht, wird die älteste Lösung gelöscht, bevor die aktuelle Lösung gespeichert wird. Der Verlauf ist in der Klasse `History` implementiert. Die Liste der gespeicherten Lösungen wird als `DefaultListModel` realisiert, um die Lösungen in der `JList` des `HistoryFrames` darstellen zu können. Neben der Liste der manuell und automatisch gespeicherten Lösungen, wird die beste bisher gefundene Lösung, sowie die Startlösung, die beim ersten Einlesen der Probleminstanz erzeugt wurde, gespeichert. Die `History` stellt diverse Methoden zum Abfragen und Setzen dieser gespeicherten Lösungen zur Verfügung.

In der Methode `addSolution` wird beispielsweise eine neue übergebene Lösung hinzugefügt. Um nicht unnötig Speicher zu verschwenden, wird zuerst einmal überprüft, ob die übergebene Lösung nicht schon gespeichert ist. Ist dies der Fall, wird die alte Lösung entfernt und die Neue mit neuer Zeitangabe gespeichert. Außerdem wird überprüft, ob die übergebene Lösung besser ist als die bisher Beste, um diese dann zu aktualisieren. Vor dem Speichern einer automatisch erstellten Lösung wird, wie oben schon angedeutet, überprüft ob die maximale Anzahl der Speicherplätze schon überschritten ist. Wenn dies der Fall ist, wird die älteste automatisch erstellte Lösung entfernt und die Neue gespeichert. Um die Aktualität der besten Lösung zu gewährleisten, gibt es die Methode `updateBest`, die nach jeder Änderung einer Lösung, ob manuell oder automatisch, aufgerufen werden sollte und die überprüft, ob es eine neue beste Lösung gibt. Um das Speichern und Öffnen eines Projektes zu vereinfachen, sind die beiden Methoden `getHistoryArray` und `fillHistoryFromArray` implementiert worden, die den Verlauf in ein Array kopieren bzw. den Verlauf mit den Daten aus dem Array füllen. Die Startlösung und die beste Lösung werden separat in der Klasse `SavePack` gespeichert. Genaueres dazu ist in Kapitel 3.4 zu finden.

3.2.2 Mobilities

Die sogenannten Mobilities bieten dem Benutzer die Möglichkeit Einfluss auf die Richtung der Optimierung zu nehmen. Durch Ändern der Lösung ist es nur möglich Einfluss auf die Ausgangssituation der Optimierung zu nehmen, während es mit Hilfe der Mobilities zusätzlich möglich ist, die Optimierung zu fokussieren. Der Benutzer kann jedem Knoten eine von drei Mobilities zuweisen. Standardmäßig besitzen alle Knoten eine hohe Mobility. Dies bedeutet, dass sie vom Optimierungsalgorithmus beliebig verändert werden können. Wird nun einem Knoten eine mittlere oder niedrige Mobility zugewiesen, verändert sich seine Mobilität. Knoten mit niedriger Mobility werden vom Optimierungsalgorithmus nicht verändert und Knoten mit mittlerer Mobility werden nur noch eingeschränkt verändert. Die Definition der Einschränkung ist pluginspezifisch und wird in den Moves berechnet. So wird für jeden Move, abhängig von, den in diesem Move zu ändernden Knoten, ein Mobilitywert berechnet. Liegt der Mobilitywert eines Moves bei 2, was einer mittleren Mobility entspricht, liegt die Wahrscheinlichkeit bei 50%, dass dieser Move ausgeführt werden darf. Die Berechnung der Mobility eines Moves wird im jeweiligen pluginspezifischen Move in der Methode `generateMobility` implementiert. Die Wahrscheinlichkeitsentscheidung, ob ein Move abhängig von seiner Mobility ausgeführt werden darf, wird in der Klasse `MobilityOrganisation` in der Methode `checkMobility` gefällt. Ein Move der einen Mobilitywert größer oder gleich 1 hat, wird niemals ausgeführt. Liegt die Mobility zwischen 1 und 3 wird eine rationale Zufallszahl mit Hilfe der Klasse `MyRandom` erzeugt und mit dem

Mobilitywert verglichen. Ist die erzeugte Zufallszahl größer oder gleich der Mobility des zu überprüfenden Moves wird der Move nicht ausgeführt.

Ich habe mich dafür entschieden, die Anzahl der Mobilities auf drei zu beschränken, die durch die drei Farben grün, gelb und rot dargestellt werden, wie es auch, in dem in Kapitel 2.1 vorgestellten Programm *HuGS*, realisiert ist. So ist eine intuitive, übersichtliche graphische Darstellung der Mobilities für den Benutzer möglich. Bei einer höheren Anzahl an Mobilities muss man sich die Frage stellen, ob der Benutzer während des Optimierungsvorgangs wirklich zwischen feineren Abstufungen der Mobilities unterscheiden und diese nutzen würde. Denn ein sehr wichtiger Punkt, während der benutzerunterstützen Optimierung, ist die Übersichtlichkeit, die meiner Meinung nach unter einer größeren Anzahl an Mobilities mehr leiden würde, als diese Änderung nutzen würde.

Mobility-Automatik

Um dem Benutzer den Umgang mit Mobilities zu vereinfachen, wird die sogenannte Mobility-Automatik zur Verfügung gestellt. Diese Funktion kann während der benutzergesteuerten Optimierung jederzeit aktiviert oder deaktiviert werden. Ist die Funktion aktiviert, wird nach jedem ausgeführten Move die Mobility jedes Knotens mit der vom Benutzer angegebenen Senkungswahrscheinlichkeit reduziert. Diese Wahrscheinlichkeitsentscheidung ist in der Klasse `MobilityOrganisation` in der Methode `updateMobilityAutomatic` implementiert. Mit Hilfe der Klasse `MyRandom` wird eine rationale Zufallszahl zwischen 0 und 1 erzeugt. Ist diese Zufallszahl kleiner als die Senkungswahrscheinlichkeit wird die Mobility des betrachteten Knotens um eins erhöht. Die Mobility von Knoten mit hoher Mobility wird dabei nicht weiter erhöht. Ebenso wird die Mobility von fixierten Knoten nicht geändert. Die Methode `updateMobilityAutomatic` wird im Optimierungsalgorithmus nach jeder Erzeugung eines neuen Moves für alle Knoten aufgerufen. Dies sollte bei Einstellung der Senkungswahrscheinlichkeit beachtet werden. Der Wert, der standardmäßig auf 0,08 Promille eingestellt ist, sollte also entsprechend klein gewählt werden.

3.2.3 Grundalgorithmus

Die Klasse `AlgoOrganisation` stellt den Rahmen des Optimierungsalgorithmus und implementiert das Interface `Runnable`. Die Klasse `TheadOrga` erstellt einen Thread mit einer Instanz von `AlgoOrganisation` und startet und beendet den Optimierungsprozess. Die Methode `runAlgo` in der Klasse `AlgoOrganisation` besteht aus eine Schleife, die läuft bis der Thread beendet wird. Der Optimierungsprozess innerhalb dieser Schleife läuft wie folgt ab. Durch die oben erwähnte Klasse `MoveGenerator` wird als erstes ein Move erzeugt. Dieser Move wird durch Aufruf der Methode `checkMobility` auf seine Ausführbarkeit bzgl. seiner Mobility hin überprüft. Dieses Vorgehen wurde im letzten Abschnitt bereits erläutert. Nun wird die Methode `doMove` der Klasse `SearchAlgo` aufgerufen. Die Klasse

`SearchAlgo` ist eine abstrakte Klasse, die die abstrakte Methode `doMove` enthält. Diese Methode bekommt einen Move und eine Lösung übergeben und entscheidet, ob der Move auf der Lösung ausgeführt werden soll. Wie diese Entscheidung getroffen wird, wird im ausgewählten Suchalgorithmus definiert, dies wird im folgenden Abschnitt genauer beschrieben. Anschließend wird, wie im letzten Abschnitt beschrieben, für jeden Knoten die Methode `updateMobilityAutomatic` der Klasse `MobilityOrganisation` aufgerufen, falls die Mobility-Automatik aktiviert ist. In den folgenden drei Abschnitten werden die zwei Suchvarianten, Greedy und Simulated Annealing genauer beschrieben. Außerdem wird erklärt, wie eine neue Suchvariante integriert werden kann. In Abbildung 3.2 ist ein vereinfachtes Klassendiagramm der Klassen die an der Optimierung beteiligt sind dargestellt.

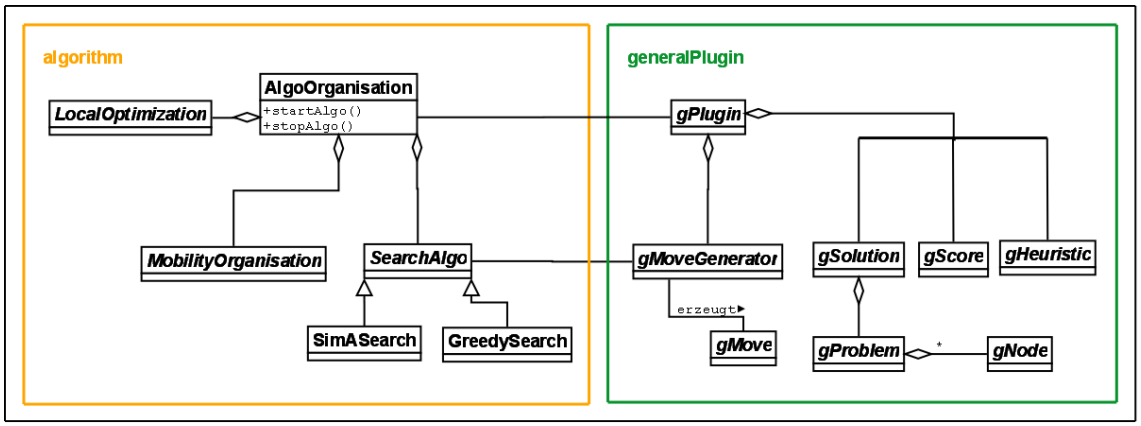

Abbildung 3.2: Vereinfachtes Klassendiagramm der Klassen die am Optimierungsalgorithmus beteiligt sind

3.2.4 Greedy

Die Klasse `GreedySearch` ist eine Unterklasse von `SearchAlgo` und implementiert einzig die Methode `doMove` die überschrieben werden muss. Hier wird überprüft, ob der übergebene Move, ausgeführt auf der übergebenen Lösung, eine Verbesserung der Güte mit sich bringt. Dazu wird der Move auf einer kopierten Version der übergebenen Lösung ausgeführt, um diese geänderte Lösung mit der Übergebenen zu vergleichen. Dieser Vergleich geschieht mit Hilfe der Methode `isBetter` der Klasse `gScore`. Diese Methode vergleicht zwei Lösungen bezüglich ihrer Güte und muss von jedem Plugin implementiert werden. Diese Optimierungsvariante dient hauptsächlich zu Testzwecken und stellt die denkbar einfachste Möglichkeit dar, eine Suchvariante, die Unterklasse von `SearchAlgo` ist, zu implementieren.

3.2.5 Simulated Annealing

Die allgemeinen Grundlagen zum Simulated Annealing Algorithmus wurden bereit in Kapitel 1.4.3 erläutert. In den folgenden zwei Abschnitten wird zuerst begründet, wieso eine Simulated Annealing Variante für HuGO eine passende Wahl ist, um dann anschließend die Realisierung zu beschreiben.

Grundidee und Motivation

Ich habe mich für eine Simulated Annealing Strategie entschieden, da dem Benutzer über die Temperatur eine weitere Möglichkeit geben ist, in die Optimierung einzugreifen und diese zu steuern. Außerdem muss jedes Optimierungsproblem, für das ein Plugin geschrieben wird, mit dieser Variante optimiert werden können. Das Algorithmen, die speziell für ein bestimmtes Problem entworfen und angepasst wurden, besser und schneller sind ist unbestritten. Doch mit HuGO soll die Möglichkeit geboten werden, ohne großen Aufwand ein Plugin für ein Optimierungsproblem zu implementieren, um es dann benutzergesteuert optimieren zu können. Für diese Art der Anwendung ist die Simulated Annealing Strategie sehr gut geeignet.

Die Simulated Annealing Variante in *HuGO* ist weiterhin um eine Tabusuche erweitert worden. Die Idee der Tabu-Suche ist es, zu verhindern, dass der Suchalgorithmus im Suchraum immer wieder die gleichen Zyklen durchläuft. Dazu wird eine sogenannte Tabuliste geführt, in der die komplementären Moves zu den gerade Durchgeführten gespeichert werden. Moves die in der Tabu-Liste enthalten sind werden nicht ausgeführt. Die Tabuliste sollte eine begrenzte Länge und die Funktionen einer FiFo-Liste besitzen. Das heißt, dass der Move der zuerst eingefügt wurde auch als erstes wieder entfernt wird, wenn die Liste ihre maximale Länge erreicht hat. [9]

Realisierung

Die Klasse `SimASearch`, in der die Simulated Annealing Variante implementiert ist, überschreibt die abstrakte Methode `doMove` der Oberklasse `SearchAlgo`. Diese Methode bekommt einen Move und eine Lösung übergeben und entscheidet, ob der Move auf der Lösung ausgeführt werden soll. Zuerst wird überprüft, ob der übergebene Move in der Tabuliste enthalten ist. Ist dies der Fall wird der Move nicht ausgeführt. Nun wird mit Hilfe der Klasse `gScore` abgefragt, ob der Move zu einer Verschlechterung der Lösung führt. Ist dies nicht der Fall, wird der Move ausgeführt. Kommen wir nun zu dem Fall, dass der Move die Lösung verschlechtert. In diesem Fall findet eine Wahrscheinlichkeitsentscheidung statt. Dazu wird zuerst einmal die Wahrscheinlichkeit w_t berechnet, mit der die verschlechternde Lösung zugelassen wird. w_t wird dabei definiert als $\exp(\frac{f_b - f_x}{T})$, mit Güte der besten bisher gefundenen Lösung f_b, Güte der Lösung nach Ausführung des Moves f_x und Temperatur T. w_t wird mit einer rationalen Zufallszahl zwischen 0 und 1

verglichen, die mit Hilfe der Klasse `MyRandom` erzeugt wird. Ist die Zufallszahl kleiner als w_t wird der Move ausgeführt, ansonsten nicht. Um so größer die Verschlechterung durch den Move ist und um so kleiner die Temperatur ist, um so geringer wird die Wahrscheinlichkeit, dass dieser Move ausgeführt wird. Vor jeder Ausführung eines Moves wird der durch die Methode `createReverseMove` erzeugte und zum Übergebenen komplementäre Move in die Tabuliste eingefügt. Erreicht die Tabuliste ihre maximale Größe, wird zuvor der erste Move der Liste, also jener der als erstes eingefügt wurde, entfernt.

Weiterhin wird vor jeder Ausführung eines Moves die Methode `updateTemperature` aufgerufen. Hier wird zuerst überprüft, ob die Variable, die die Anzahl der Iterationen für jeden Temperaturwert bestimmt, Null erreicht hat. Ist dies der Fall, findet die Temperaturaktualisierung statt und die Variable wird wieder auf den eingestellten Anfangswert gesetzt, falls nicht, wird die Variable um eins gesenkt und es findet keine Aktualisierung der Temperatur statt. Kommen wir nun zur eigentlichen Temperaturaktualisierung. Hier wird die Temperatur abhängig vom durch den Benutzer ausgewählten Modus und Wert verringert, dabei stehen zwei Modi zur Auswahl. Im ersten Modus, der sich *linear* nennt, wird die Temperatur bei jedem Update um den vom Benutzer angegeben Wert verringert. Die Temperatur sinkt also gleichmäßig von der angegebenen Starttemperatur bis zum Nullpunkt. Im zweiten Fall, namens *exponential*, wird die Temperatur um die angegebenen Promille verringert. Dadurch wird die Temperatursenkung um so langsamer, je weiter die Temperatur sinkt.

3.2.6 Entwicklung einer neue Suchvariante

Zur Implementierung eines neuen Suchalgorithmus, muss eine neue Klasse erstellt werden, die von `SearchAlgo` erbt. Die Methode `doMove` muss überschrieben werden. Sie muss zurückgeben, ob der übergebene Move auf der übergebenen Lösung ausgeführt werden soll oder nicht. Dabei kann auf eine Instanz des `gPlugins` und der `AlgoOrganisation` zurückgegriffen werden. Lösungen können mit Hilfe der Klasse `gScore` des jeweiligen Plugins verglichen werden. Damit die Suchvariante für den Benutzer auf der graphischen Oberfläche auswählbar ist, muss sie allerdings noch dort eingefügt werden.

3.2.7 Lokale Optimierung

Die Idee der lokalen Optimierung ist, dass nach manuellen Änderungen durch den Benutzer eine kurze lokale Optimierung ausgeführt wird, um die veränderte Nachbarschaft zu optimieren, indem einfache Verbesserungen ausgeführt werden. Diese Funktion ist vom Benutzer jeder Zeit an- und abschaltbar. Wenn sie aktiviert ist, muss nach jeder manuellen Änderung der Lösung durch den Benutzer die Methode `localOptimization` der Klasse `LocalOptimization` aufgerufen werden. Diese bekommt neben der zu bearbeitenden Lösung, den Knoten der gezielt verändert wurde und eine Liste der Knoten, die durch

die gezielte Veränderung mit geändert wurden, übergeben. Die Liste kann auch leer sein. Nun wird eine neue Liste namens `newChangedNodes` erstellt und mit den übergebenen veränderten Knoten gefüllt. Außerdem werden alle Knoten, die sich in der Nachbarschaft des geänderten Knotens befinden, in die Liste `newChangedNodes` eingefügt. Die Nachbarschaftsknoten liefert die Methode `getNeighborhood` der Klasse `gMoveGenerator`, denn die Definition der Nachbarschaftbeziehung ist pluginspezifisch.

Nun beginnt die eigentliche lokale Optimierung. Dabei werden in einer Schleife alle Knoten der Liste `newChangedNodes` einmal durchlaufen. Die in der Schleife neu hinzugefügten Knoten werden ebenfalls einmal betrachtet. Zunächst einmal wird durch den Aufruf der Methode `getNeighborhoodMoves` der Klasse `gMoveGenerator` eine Liste von Moves erzeugt. Die Moves, die den vom Benutzer veränderten Knoten betreffen, werden allerdings wieder entfernt. Denn dieser Knoten wurde gezielt von Benutzer verändert und diese Veränderung sollte nicht rückgängig gemacht werden. Die übrigen Moves werden darauf überprüft, ob sie die Lösung verbessern. Alle verbessernden Moves werden anschließend ausgeführt. Weiterhin werden die, durch den Move veränderten Knoten, sowie ihre Nachbarknoten in die Liste `newChangedNodes` eingefügt.

Die lokale Optimierung kann zu weitreichenden Verbesserungen führen, obwohl nur ein einzelner Knoten manuell verändert wurde. Ist die Optimierung in einem fortgeschrittenem Stadium, sind die damit zu erreichenden Verbesserungen allerdings in der Regel kleiner. Außerdem kommt es auch oft vor, dass die lokale Optimierung keine Verbesserungen findet.

3.2.8 Heuristik

Wurde eine neue Probleminstanz eingelesen und eine erste Startlösung dazu erzeugt, ist diese in der Regel zufällig generiert und sehr schlecht. Schlechte Lösungen erscheinen dem Benutzer in der Regel äußerst unübersichtlich und es ist schwer einen Ansatzpunkt für die manuelle Optimierung zu finden. Eine Möglichkeit wäre es, zu Beginn den Suchalgorithmus einige Zeit lang optimieren zu lassen, bis eine Lösung gefunden wurde, die gut genug ist dem Benutzer eine gewissen Übersichtlichkeit zu bieten. Diese Methode kann je nach Problemgröße allerdings einige Zeit in Anspruch nehmen. Um einfach und schnell eine akzeptable Startlösung zu finden, wurde eine Möglichkeit zur Realisierung einer einfachen problemspezifischen Heuristik geschaffen. Jedes Plugin muss zur Implementierung einer Heuristik eine Unterklasse von `gHeuristic` implementieren. Dabei muss nur die Methode `getSolution` überschrieben werden, die eine Lösung übergeben bekommt und die berechnete Lösung zurück gibt. Die Realisierung der Heuristik bleibt also komplett dem Entwickler des Plugins überlassen.

3.3 Graphische Oberfläche

Die graphische Oberfläche besteht aus mehreren Teilen. Zuerst betrachten wir das Hauptfenster und dessen Komponenten. Anschließend wird kurz die Darstellung des Verlaufs beschrieben und es werden einige einfache Dialoge erläutert.

3.3.1 Hauptfenster

Die Komponenten des Hauptfensters werden in der Hauptklasse der graphischen Oberfläche `HuGOUI` implementiert. Auf der linken Seite befinden sich diverse Funktionen und Optionen, die in verschiedene Menüs unterteilt sind. In Kapitel 4 werden sie genau beschrieben. Mit Ausnahme des Letzten sind die Menüs pluginübergreifend. Dieses Seitenmenü ist ein sogenannter `TaskPaneContainer` und wird in der Methode `createTaskPanePanel` implementiert. Jedes Menü ist eine `TaskPane`, die einzeln ausklappbar ist. Dadurch wird dem Benutzer ermöglicht, diesen Teil der graphischen Oberfläche an seine Bedürfnisse anzupassen. So wird ein schneller Arbeitsablauf bei der Optimierung sichergestellt. Jede `TaskPane` beinhaltet ein `JPanel` in dem wiederum die einzelnen Komponenten, wie zum Beispiel Buttons und Textfelder, enthalten sind. Für jede `TaskPane` gibt es eine Methode, die das dazugehörige `JPanel` erzeugt und das Layout erstellt. Die zugehörigen `ActionListener` befinden sich als lokale Klassen ebenfalls in der `HuGOUI`. Das JPanel der `TaskPane`, in der die pluginspezifischen Einstellungen zur Verfügung gestellt werden, wird durch die abstrakte Methode `createPluginTaskPanel` in der Klasse `gPluginUI` erzeugt.

Die Menüleiste ist für alle Plugins identisch. Sie beinhaltet nur Funktionen die pluginübergreifend sind und wird in der Methode `createMenuBar` implementiert. Die benötigten `ActionListener` befinden sich wieder als lokale Klassen in `HuGOUI`. Rechts mittig befindet sich eine große Fläche, auf der die Lösungen der Probleminstanzen des jeweiligen Plugins angezeigt werden. Diese Fläche wird in der Klasse `gEditPanel` implementiert und gehört damit zu den pluginspezifischen Klassen. `gEditPanel` ist eine Unterklasse von JPanel und wird in der Klasse `gPluginUI` erzeugt. `gEditPanel` enthält die abstrakte Methode `paintComponent` der Oberklasse `JPanel` die implementiert werden muss und in der die jeweilige Lösung gezeichnet wird.

Oberhalb des EditPanels befindet sich ein Panel, dass Informationen über die aktuelle Lösung und über die beste bisher gefundene Lösung anzeigt. Dieses Panel wird durch die abstrakte Methode `createPluginInfoPanel` in der Klasse `gPluginUI` implementiert. Die Gestaltung der Informationsanzeige ist also vom Plugin abhängig. Um die Pluginrealisierung zu vereinfachen, existiert das Paket `template`, das in Kapitel 5.1 genauer erläutert wird. Hier ist für die Informationsanzeige und einige andere Dinge schon ein Layout vorgegeben, welches übernommen oder bei Bedarf geändert werden kann. Durch diese Möglichkeit ist es einerseits möglich, schnell und einfach ein Plugin zu implementie-

ren, andererseits bleibt die Entwicklung eines Plugins flexibel. Die Klasse `HuGOUI` erbt von der Klasse `JFrame` und stellt das Hauptfenster dar. In der Methode `addMainComponents` werden alle zuvor beschriebenen Komponenten erzeugt und es wird ein Layout für alle Komponenten erstellt. Diese Methode wird im Konstruktor von `HuGOUI` aufgerufen.

3.3.2 Anzeige des Verlaufs

Über die Funktion `Verlauf - Verlauf öffnen` erscheint ein neues Fenster, dass den Verlauf anzeigt. Dieses Fenster ist in der Klasse `HistoryFrame` implementiert, die von `JFrame` erbt. Die Klasse `HistoryFrame` enthält ein Objekt der Klasse `History` in der, wie in Kapitel 3.2.1 beschrieben, der Verlauf gespeichert und verwaltet wird. In der Methode `createFrame` werden die Komponenten des Fensters erzeugt und das Layout erstellt. Das Fenster besteht zum einen aus einer `JList`, dem das von der Klasse `History` erzeugte `DefaultListModel`, zugewiesen wird. Die benötigten Listener befinden sich als lokal in der Klasse `HistoryFrame`. Das Fenster des Verlaufs kann parallel zum Hauptfenster angezeigt werden.

3.3.3 Dialoge

Nach Start von *HuGO* öffnet sich als erstes ein Dialog zur Auswahl des Plugins. Dieser Dialog ist in der Klasse `ChoosingPluginFrame` implementiert. Wenn nach Auswahl eines Plugins aus der `JComboBox` der Button *Load* betätigt wird, wird ein Objekt der Klasse `Plugin`, des ausgewählten Plugins erzeugt. Anschließend wird eine Instanz der Klasse `HuGOUI` erstellt und bekommt das zuvor erzeugte `Plugin` übergeben. Ist *HuGO* auf diesem Wege gestartet worden, ist es ohne Neustart nicht mehr möglich, das aktuelle Plugin zu wechseln. Diese Klasse muss, wie in Kapitel 5.1 genauer beschrieben, bei Implementierung eines neuen Plugins angepasst werden. Wenn der Benutzer eine Datei öffnet, die nicht korrekt codiert ist, wird ein Fehlerdialog angezeigt. Dazu wird eine Instanz der Klasse `JDialog` in `HuGOUI` erzeugt.

3.4 IO

Die Ein- und Ausgabe besteht aus drei Funktionen. Die erste ist das Einlesen von Problemen eines Plugins aus einer Textdatei in der eine Probleminstanz codiert ist. Dieser Teil wird hauptsächlich in den pluginspezifischen Klasse implementiert. Die zweite Funktion ist das Speichern eines Projektes. Dabei wird nicht nur die aktuelle Lösung gespeichert, sondern auch der gesamte Verlauf. Das Öffnen von so gespeicherten Projekten ist die dritte Funktion. Die Klasse `IoOrganisation` implementiert diese Funktionen, während ein `JFileChooser`, der in der Klasse `HuGOUI` erzeugt wird, ein `File` liefert, welches an die jeweilige Methode der `IoOrganisation` übergeben wird.

3.4.1 Neue Probleminstanz einlesen

In der Methode `loadProblem` ist das Einlesen einer Probleminstanz aus einer Textdatei implementiert. Ein `BufferedReader` erzeugt mit einem `FileReader` einen String aus dem übergebenen `File`. Dieser wird in der Methode `readSolution`, der pluginspezifischen Klasse abgeleitet und von `gProblemParser` in ein `gSolution` Objekt umgewandelt. Diese `gSolution` wird dem Plugin als aktuelle Lösung zugewiesen und die erzeugte Lösung wird im Verlauf als sogenannte Startlösung gespeichert.

3.4.2 Projekt Speichern

Das Speichern eines Projektes ist in der Methode `saveProject` implementiert. Es wird ein sogenanntes `SavePack` erstellt, dass aus der aktuellen, der besten und der Startlösung besteht. Außerdem enthält das `SavePack` ein Array in dem alle Lösungen des Verlaufs gespeichert sind. Mit einem `FileWriter` wird mit Hilfe der Klasse `XStream` dieses erzeugte `SavePack` als XML-Datei gespeichert.

3.4.3 Projekt Öffnen

Zum Öffnen einer zuvor gespeicherten Datei, wird die Methode `openProject` aufgerufen. Dort wird wieder mit Hilfe der Klasse `XStream` und einem `FileReader` ein Objekt der Klasse `SavePack` erzeugt. Dann wird die aktuelle Lösung des Plugins durch die im `SavePack` Gespeicherte ersetzt. Ebenso wird der Verlauf mit den Daten aus dem `SavePack` gefüllt.

3.5 Utils

In diesem Paket sind die Hilfsklassen `MyRandom`, `DoubleDoc`, `IntDoc` und `properties` enthalten. Die Klasse `MyRandom` bietet zwei statische Methoden um ganzzahlige und rationale Zufallszahlen zu erzeugen. Diese werden mit Hilfe der Javaklasse `Random` erzeugt. Es besteht die Möglichkeit, bei der Erzeugung des `Random` Objekts, durch Setzen eines Seeds, für die Fehlerbehebung reproduzierbares Wahrscheinlichkeitsverhalten zu erreichen. Um zu verhindern, dass ungültige Werte und Buchstaben in die Textfelder der graphischen Oberfläche eingetragen werden können, wurden die beiden Klassen `DoubleDoc` und `IntDoc` implementiert, die Unterklassen von `PlainDocument` sind. Dadurch können nur ganze bzw. rationale Zahlen in die Textfelder getippt werden. In der statischen Klasse `properties` sind alle statischen Variablen enthalten, die für diverse Einstellungen benötigt werden.

Kapitel 4

Handbuch

Dieses Kapitel ist ein Benutzerhandbuch für *HuGO*. Hier werden alle Funktionen, die ein Benutzer kennen muss, um das Programm zum Optimieren des *CrossMin*- oder *BackPack*-Plugins zu benutzen, vorgestellt. Neben den grundlegenden Funktionen, werden auch die Optimierungsvarianten erklärt. Denn zur effizienten Optimierung mit *HuGO* ist ein gewisses Grundverständnis des Optimierungsalgorithmus sehr hilfreich. Im letzten Abschnitt sind die pluginspezifischen Funktionen der beiden realisierten Plugins beschrieben.

4.1 Grundlegende Funktionen

In diesem Abschnitt werden die grundlegenden Funktionen von HuGO vorgestellt. Im ersten Unterabschnitt werden Standardfunktionen, wie das Öffnen und Speichern eines Projekt, sowie die Zoomfunktion, kurz beschrieben. Anschließend wird der Verlauf, in dem Lösungen während der aktuellen Optimierung gespeichert werden können, erläutert. Im dritten Unterabschnitt wird beschrieben, welche möglichen Anzeigemodi es gibt, mit denen der Optimierungsalgorithmus beobachtet werden kann.

4.1.1 Standardfunktionen

Nach dem Start von HuGO kann der Benutzer, wie in Abbildung 4.1 gezeigt ist, aus einer Liste das gewünschte Plugin auswählen. Nun besteht die Möglichkeit über *File - Load New Problem Instance* ein zuvor in einer Textdatei codiertes Problem zu laden. Mit der Funktion *File - Save Project* wird das aktuelle Projekt mit der aktuellen Lösung, sowie dem gesamten Verlauf und den darin enthaltenen Lösungen, gespeichert. Über die Funktion *File - Open Project* kann nun das zuvor mit HuGO gespeicherte Projekt wieder geöffnet werden. Dabei bleiben bisher ausgeführte Optimierungsschritte erhalten. Das *File* Menü ist in Abbildung 4.2 zu sehen. Ist das aktuelle Problem zu groß, um auf der Zeichenfläche komplett angezeigt zu werden und wünscht sich der Benutzer eine bessere Übersicht, so lässt sich über die Funktion *View* in der Menüleiste der Vergrößerungsfaktor einstellen.

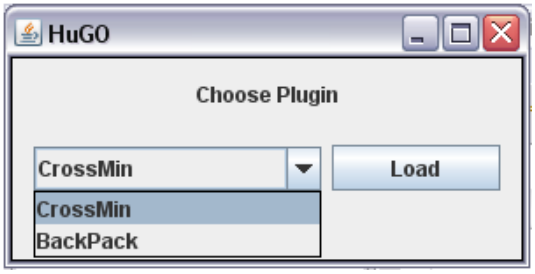

Abbildung 4.1: Der Dialog zur Auswahl eines Plugins

Dabei wird nur die Ansicht der Komponenten auf der Zeichenfläche angepasst, während die Menüs und die Informationsanzeige unverändert bleiben.

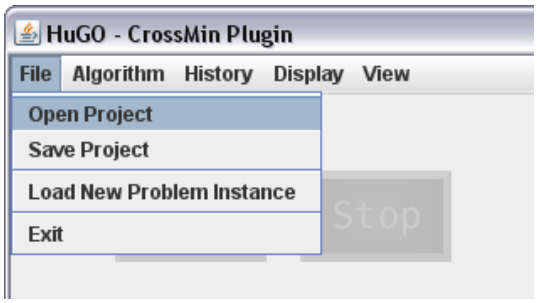

Abbildung 4.2: Das Untermenü *File*

4.1.2 Verlauf

Der Verlauf besteht aus einer Liste von Lösungen die gespeichert und jederzeit wieder geladen werden können. Es besteht immer die Möglichkeit, die aktuelle Lösung über die Funktion *History - Save Current Solution* zum Verlauf hinzuzufügen. Diese manuell gespeicherten Lösungen sind in der Verlaufsliste mit einem **M** gekennzeichnet. Die beste, bis zu diesem Zeitpunkt berechnete, Lösung wird jederzeit im Verlauf gespeichert und aktualisiert. Sie kann mit Hilfe der Funktion *History - Load Best Solution* geladen werden. Weiterhin besteht die Möglichkeit zur Startlösung zurückzukehren. Startlösung ist die Lösung, die beim ersten Laden eines Problems als Ausgangssituation erzeugt wird. Bevor die Startlösung geladen wird, wird automatisch die aktuelle Lösung im Verlauf gespeichert. Über die Funktion *History - Open History* öffnet sich ein Fenster, welches die Liste aller gespeicherten Lösungen anzeigt. Zu jeder Lösung wird das Datum und die Uhrzeit der Speicherung angezeigt, sowie pluginspezifische Eigenschaften, die die Güte der Lösung definieren. Durch Auswählen einer Lösung werden im Informationsfenster weitere Informationen angezeigt. Außerdem besteht die Möglichkeit alle oder einzelne Lösungen aus dem Verlauf zu löschen. Die Anzahl der im Verlauf gespeicherten Lösungen, die nicht

manuell gespeichert wurden, ist begrenzt. Allerdings kann diese Anzahl im Verlaufsfenster geändert werden. Die beste bisher gefundene Lösung bleibt immer gespeichert und wird oberhalb der Verlaufsliste separat angezeigt. Der Verlauf ist in Abbildung 4.3 zu sehen.

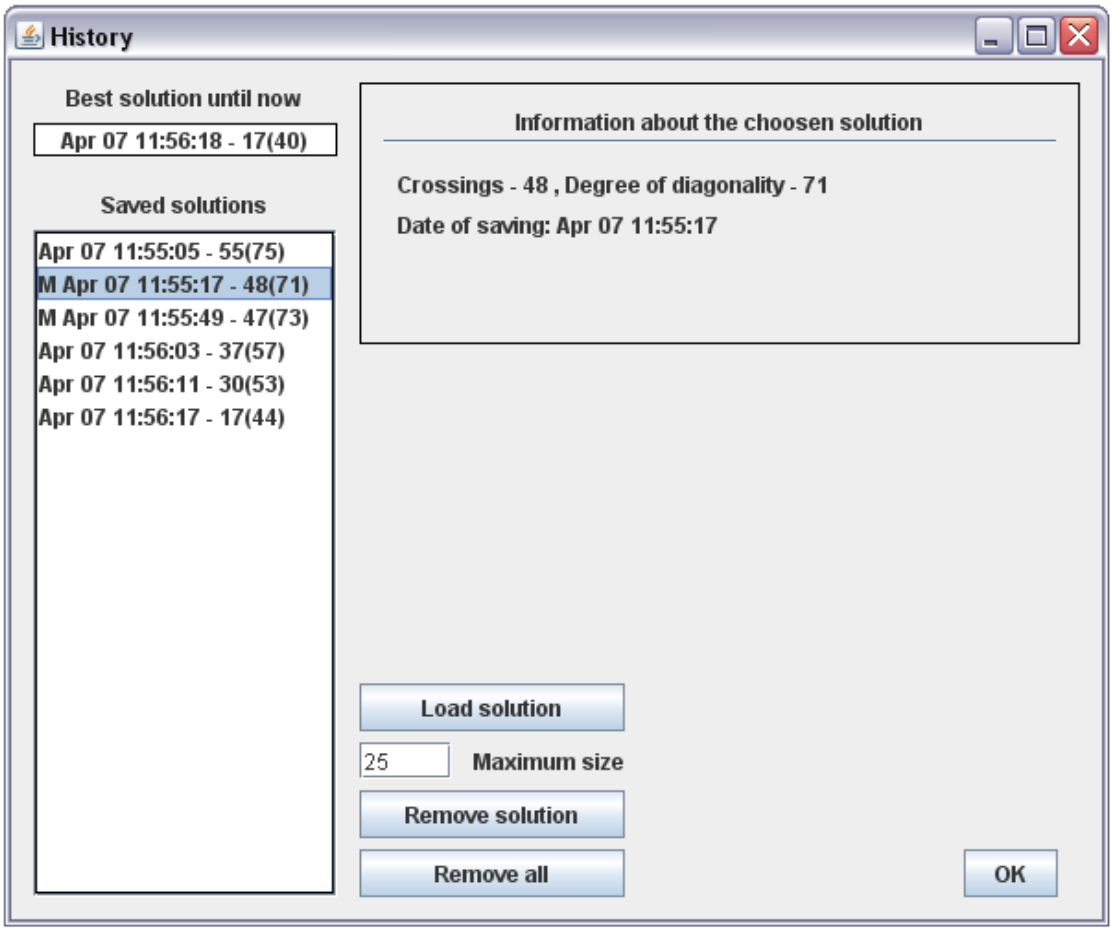

Abbildung 4.3: Das Fenster der *History*, indem der Verlauf angezeigt ist

4.1.3 Anzeigeoptionen

Unter dem Menüpunkt *Diplay* stehen eine Reihe von Einstellungsmöglichkeiten zur Verfügung. Es besteht die Möglichkeit zwischen drei verschiedenen Abbruchbedingungen für den Optimierungsalgorithmus zu wählen. Durch die Auswahl von *Stop After Each Step* wird der Optimierungsalgorithmus nach jedem ausgeführten Move gestoppt und die erzeugte Lösung wird angezeigt. Bei der zweiten Möglichkeit *Only After Improving Steps* stoppt der Algorithmus, sobald eine Lösung gefunden wurde, die besser ist als die bisher Beste. Wird die Suche durch den *Stopbutton* beendet, bevor eine bessere Lösung gefunden wurde, wird der aktuelle Stand des Optimierungsalgorithmus angezeigt. Bei Auswahl dieses Anzeigemodus wird jede vom Algorithmus berechnete Lösung auf der Zeichenfläche

angezeigt, es wird allerdings keine Pause provoziert, um die Zeit während der eine Lösung angezeigt wird, zu verlängern. Dies kann teilweise zu einer unübersichtlichen Visualisierung führen. In einigen Situationen kann ein solcher Anzeigemodus allerdings auch sinnvoll sein. Auch ohne Anhalten des Algorithmus verzögert dieser Anzeigemodus die Optimierung. Eine Aktivierung empfiehlt sich nur, wenn der Benutzer die Optimierung beobachten möchte. Der *Endless Run* Modus läuft solange bis der *Stopbutton* betätigt wird. Sobald eine verbessernde Lösung gefunden ist, wird dies über dem *Stopbutton* markiert, dabei wird auch die Güte der gefundenen Lösung angezeigt. Wird der Lauf durch den *Stopbutton* beendet, wird nicht die beste gefundene Lösung, sondern der aktuelle Stand des Algorithmus angezeigt. Wurde während des Laufs eine bessere Lösung gefunden, kann diese über *History - Load Best Solution* geladen werden. Mit der Option *Show Intermediate Steps* besteht die Möglichkeit, genau zu verfolgen wie der Algorithmus vorgeht. Ist die Option aktiviert, wird nach jedem Move die erzeugte Lösung für kurze Zeit angezeigt und die durch den Move veränderten Knoten werden hervorgehoben, der Algorithmus wird dabei für kurze Zeit pausiert. Diese Option sollte also nur aktiviert werden, wenn der Benutzer den Algorithmus verfolgen möchte, denn die Optimierung wird dadurch erheblich verzögert. Weiterhin kann in diesem Untermenü die Anzeige von Namen der Knoten an- und ausgeschaltet werden. In Abbildung 4.4 ist diese Untermenü abgebildet.

Abbildung 4.4: Das Untermenü *Display*, indem die Anzeigeoptionen geändert werden können

4.2 Funktionen zur Steuerung der Optimierung

In diesem Kapitel werden die Funktionen zur Steuerung der Optimierung vorgestellt, die der Benutzer manuell ausführen kann. Im ersten Abschnitt, wird das manuelle Ändern einer Lösung beschrieben. Im zweiten Abschnitt werden die Mobilities vorgestellt, mit deren Hilfe der Benutzer die Optimierung steuern kann. In diesem Abschnitt wird auch die *Mobility Automatic* und die *Auto red*-Funktion vorgestellt.

4.2.1 Manuelle Änderung der Lösung

Ein zentraler Punkt des Konzepts von *HuGO* ist das wiederholte Anpassen der Lösung durch den Benutzer. Dazu muss es möglich sein, die angezeigte Lösung einfach nach eigenen Wünschen zu verändern. Wie genau diese Änderungen definiert sind, ist pluginspezifisch. Im *CrossMin*-Plugin lässt sich eine Lösung beispielsweise durch Verschieben der Knoten auf neue Positionen auf der selben Schicht ändern. Im *BackPack*-Plugin funktionieren die Änderungen ähnlich. Hier können durch Drag&Drop die Gegenstände aus dem Rucksack hinaus und in ihn hinein geschoben werden. Eine genauere Beschreibung der pluginspezifischen Funktionen ist in Kapitel 4.4 zu finden.

4.2.2 Mobilities

Eine wichtige Funktion zur Steuerung der Optimierung sind Mobilities. Unter dem seitlichen Menüpunkt *Mobilities* kann jedem Knoten eine von drei Farben, die die Mobilities repräsentieren, zugewiesen werden. Standardmäßig sind alle Knoten grün, das heißt sie besitzen eine hohe Mobility. Zum Vergeben der Mobilities muss zuerst eine der drei Farben grün, gelb oder rot ausgewählt und dann auf die zu färbenden Knoten geklickt werden. Rote Knoten, stellen Knoten mit niedriger Mobility dar und werden vom Optimierungsalgorithmus nicht geändert. Gelbe Knoten, die mittlere Mobility besitzen, werden eingeschränkt geändert und grüne Knoten werden beliebig geändert. Die *Mobility Automatic* ist eine auswählbare Funktion, die den Umgang mit den Mobilities für den Benutzer ein Stück weit automatisiert. Die vergebenen Mobilities werden nach jedem Move mit einer gewissen Wahrscheinlichkeit erhöht. Knoten mit niedriger Mobility, bekommen mittlere Mobility und mittlere bekommen hohe Mobility zugewiesen. Die Wahrscheinlichkeit, mit der die Mobility erhöht wird, ist im Untermenü Mobilities in Promille angegeben und kann geändert werden. Eine weitere Funktion, die die Mobilities betrifft, ist die *Auto red*-Funktion. Ist diese aktiviert ist, wird Knoten, die manuell vom Benutzer verändert wurden, eine niedrige Mobility zugewiesen. Die Idee hinter dieser Funktion ist, dass Änderungen die der Benutzer durchführt, nicht direkt wieder vom Optimierungsalgorithmus rückgängig gemacht werden. Zusammen mit der *Mobility Automatic* wird so eine komfortable Automatik zur Benutzung der Mobilities geboten.

Neben der Möglichkeit Mobilities zu vergeben, gibt es noch die *Fix*-Funktion, mit der beliebige Knoten fixiert werden können. Fixierte Knoten werden, genau wie Knoten mit niedriger Mobility, nicht vom Optimierungsalgorithmus verändert. Ist die *Mobility Automatic* ausgeschaltet, besteht zwischen fixierten und Knoten mit niedriger Mobility kein Unterschied. Bei eingeschalteter *Mobility Automatic* werden fixierte Knoten weiterhin nie verändert. Bei Knoten mit zu Anfang niedriger Mobility, kann diese auf mittlere und hohe Werte steigen und die Knoten können wieder vom Optimierungsalgorithmus verändert werden. Das Fixieren von Knoten funktioniert ähnlich dem Zuweisen von Mobilities. Ist

die Funktion *Fix* aktiviert, wird ein anschließend angeklickter Knoten fixiert. Um diese Fixierung wieder zu entfernen, genügt ein erneutes Klicken auf den Knoten. Ist die Funktion *Fix* nicht aktiv, wird die Fixierung von Knoten nicht geändert. In Abbildung 4.5 ist diese Untermenü *Mobilities* abgebildet.

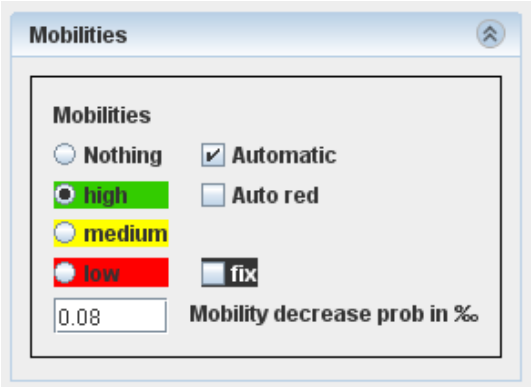

Abbildung 4.5: Das Untermenü *Mobilities*, indem die Mobilities vergeben werden können

4.3 Optimierungswerkzeuge

In diesem Unterkapitel wird beschrieben, wie die Funktionen zur Optimierung einzusetzen sind. Das wichtigste Werkzeug ist der Optimierungsalgorithmus, der durch den *Startbutton* gestartet wird. Wurde in den Anzeigeoptionen der Endloslauf ausgewählt, muss die Optimierung durch den *Stopbutton* wieder beendet werden. Doch auch bei Wahl der anderen Anzeigemöglichkeiten besteht jederzeit die Möglichkeit, die Optimierung durch den *Stopbutton* zu beenden. Es gibt zwei Optimierungsalgorithmen zwischen denen unter dem Menüpunkt *Algorithm* gewechselt werden kann. Außerdem kann eine Heuristik auf der aktuellen Lösung ausgeführt werden und die Funktion *Local Optimization* kann je nach Plugin zur Verfügung stehen. Das Seitenmenü *Algorithm* ist in Abbildung 4.6 abgebildet. Genaueres zu den Optimierungsfunktionen wird in den folgenden Abschnitten erläutert.

4.3.1 Optimierungsvarianten

Alle Optimierungsvarianten arbeiten auf sogenannten Moves. Ein Move stellt eine kleine Änderung der Lösung dar, die pluginspezifisch definiert ist. Im Algorithmus wird jeweils ein Move erzeugt, der dann von der jeweiligen Suchvariante überprüft wird, um zu entscheiden, ob der Move ausgeführt wird oder nicht.

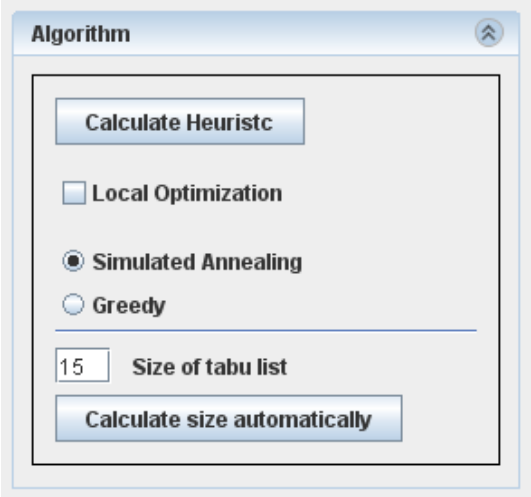

Abbildung 4.6: Das Untermenü *Algorithm*, das die Einstellungen des Optimierungsalgorithmus enthält

Greedy

Die Greedy Variante lässt keinerlei verschlechternde Veränderungen der Lösung zu. Dadurch bleibt die Optimierung schnell in lokalen Optima hängen, da es von dieser Lösung aus keinen einzelnen Move gibt, der die Lösung verbessert. Verbesserungen der Lösung sind in lokalen Optima nur möglich indem, mindestens ein verschlechternder Move und anschließend mindestens ein verbessernder Move ausgeführt wird. Führt man zuerst die Heuristik aus und lässt dann eine kurze Zeit den Greedy-Algorithmus rechnen, kommt man allerdings schnell zu einem akzeptablen Ergebnis, mit dem man dann unter Umständen gut weiter optimieren kann. Die Standardvariante sollte allerdings immer, die im Folgenden erläuterte Simulated Annealing Variante sein.

Simulated Annealing

Der Optimierungsalgorithmus mit Simulated Annealing Variante ist der wichtigste Teil der Optimierungswerkzeuge. Bei dieser Optimierungsvariante werden auch verschlechternde Moves mit einer gewissen Wahrscheinlichkeit ausgeführt. Der Algorithmus hat eine Temperatur die im Laufe der Zeit sinkt. Die Starttemperatur ist vom Benutzer im Untermenü *Temperature* einstellbar. Standardmäßig startet die Temperatur bei 20. Die Geschwindigkeit der Temperatursenkung ist abhängig von dem ausgewählten Modus und der angegebenen Anzahl an Iterationen. Die Anzahl der Iterationen gibt an, wie viele Iterationen für jeden erreichten Temperaturwert durchgeführt werden, bis die Temperatur wieder gesenkt wird. Ist die Anzahl der Iterationen 0, wird die Temperatur nach jedem erzeugten Moves gesenkt. Kommen wir nun zur eigentlichen Temperatursenkung, die nach zwei verschiedenen Modi stattfinden kann. Ist die lineare Temperatursenkung aktiviert, sinkt die

Temperatur nach jedem ausgeführten Move um den angegebenen, einstellbaren Wert. Ist dagegen die exponentielle Temperatursenkung aktiviert, wird die Temperatur nach jedem Move um die angegebenen Promille gesenkt. In diesem Fall sinkt die Temperatur nicht gleichmäßig, sondern zu Beginn schneller und in Richtung Nullpunkt immer langsamer.

Der große Vorteil der Simulated Annealing Optimierungsvariante im Gegensatz zur Greedy Variante ist, dass Simulated Annealing nicht in lokalen Optima hängen bleibt. Die Grundidee von Simulated Annealing ist, dass zu Beginn viele verschlechternde Moves zugelassen werden und mit Senkung der Temperatur die Anzahl der verschlechternden Moves reduziert wird, um zum Optimum zu kommen. Eine genaue Beschreibung der Funktionsweise der Simulated Annealing Variante ist in Kapitel 3.2.5 zu finden. Dieser Ansatz funktioniert in der Praxis natürlich nicht so perfekt wie beschrieben. Der Algorithmus weiß nicht, ob er beim Senken der Temperatur nicht noch in einem lokalen Optima ist. Der Einsatz von Simulated Annealing in der *Human Guided Optimization* bietet sich allerdings an. Die Temperatur ist während des Optimierungsvorgangs für den Benutzer sichtbar und es besteht die Möglichkeit, neben den Einstellungen zur der Temperatursenkung, die Temperatur direkt zu ändern. Angenommen die Optimierung läuft schon eine ganze Weile, die Temperatur ist schon sehr niedrig und es werden keine besseren Lösungen mehr gefunden. In diesem Fall bietet es sich zum Beispiel an, die Temperatur etwas zu erhöhen um unter Umständen ein lokales Optimum zu verlassen. Zusammen mit dem Bearbeiten der Lösungen und dem Verteilen von Mobilities, geben die Temperatureinstellungen dem Benutzer eine gute Möglichkeit zur Lenkung der Optimierung.

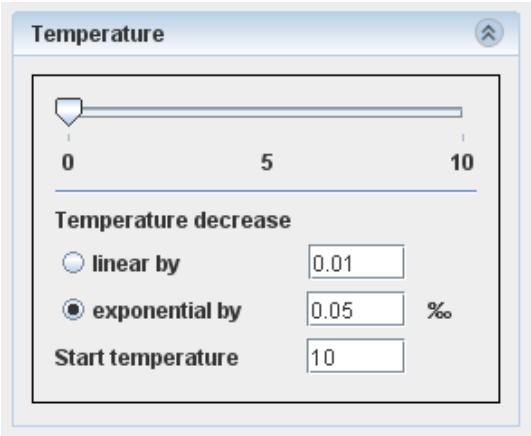

Abbildung 4.7: Das Untermenü *Temperature*, indem sich die Temperatureinstellungen der Simulated Annealing Variante befinden

4.3.2 Heuristik

Neben den oben erwähnten Werkzeugen zur Optimierung, bietet die Funktion *Heuristic* eine kleine einfache, aber sehr nützliche Zusatzfunktion. Durch Aufrufen dieser Funk-

tion wird auf der aktuellen Lösung eine einfache Heuristik aufgerufen. Diese Heuristik wird pluginspezifisch implementiert und sollte in der Regel nur eine kurze Laufzeit haben. Während des eigentlichen Optimierungsverfahrens ist es nicht immer nützlich die Heuristik berechnen zu lassen. Allerdings kann mit Hilfe der Heuristik zu Beginn der Optimierung eine relativ gute Startlösung schnell berechnet werden. Dadurch verkürzt sich die Optimierungszeit oft erheblich. Denn die ersten Lösungen sind oft sehr schlecht, da sie zufällig erzeugt werden. Sehr schlechte Lösungen sind je nach Plugin für den Benutzer oft völlig unübersichtlich. Um überhaupt mit der manuellen Optimierung anfangen zu können, muss dann erst der Optimierungsalgorithmus einige Zeit laufen. Dies kann je nach Problemgröße eine durchaus längere Zeit in Anspruch nehmen. Durch die schnelle Heuristik in Verbindung mit einem kurzen Lauf der Greedy Variante wird diese Zeit eingespart. Weiterhin kann die Berechnung der Heuristik auch in festgefahrenen Situationen einen neuen Anstoß geben.

4.3.3 lokale Optimierung

Die lokale Optimierung ist eine weitere auswählbare Funktion um die Optimierung zu unterstützen. Ist die lokale Optimierung aktiviert und wird einer oder werden mehrere Knoten durch den Benutzer verändert, sucht ein lokaler Optimierungsalgorithmus nach Verbesserungen im lokalen Umkreis dieser veränderten Knoten. Die lokalen Optimierungen, die die Lösung nur verbessern, werden sofort ausgeführt. Die dabei veränderten Knoten werden gelb hervorgehoben. Die lokale Optimierung breitet sich dabei rekursiv aus. Wenn in der direkten Nachbarschaft des manuell veränderten Knotens Verbesserungen gefunden wurden, wird wiederum in der Nachbarschaft der verbesserten Knoten nach neuen lokalen Verbesserungen gesucht. Das führt dazu, dass selbst kleine Veränderungen, durchaus weitreichende Verbesserungen nach sich ziehen können. Deshalb sollte diese Funktion nicht aktiviert werden, wenn es Ziel ist, eine bestimmte Lösung zu modellieren oder bestimmte Restriktionen einzubauen. Die genaue Definition der Nachbarschaft eines Knotens ist pluginspezifisch. Allerdings ist die lokale Optimierung nicht bei allen Plugins sinnvoll implementierbar. Wenn die Funktion im Menü ausgegraut ist, steht sie für dieses Plugin nicht zur Verfügung.

4.4 Plugin

In diesem Abschnitt werden die pluginspezifischen Funktionen und Anzeigen für die beiden implementierten Plugins *BackPack* und *CrossMin* beschrieben.

4.4.1 Das *BackPack*-Plugin

Die Gegenstände des *BackPack*-Plugins werden auf der Zeichenfläche als Rechtecke dargestellt. In jedem dieser Rechtecke befinden sich zwei Balken. Die Länge des oberen Balken, der mit dem genauen Gewicht beschriftet ist, repräsentiert das Gewicht des Gegenstandes. Der untere Balken repräsentiert den Wert des Gegenstandes und auch dieser Balken ist mit seinem genauen Wert beschriftet. Ist die Funktion *Show Node Names* aktiviert, wird oberhalb des Gewichtsbalkens der Name des Gegenstandes, der Teil der Eingabe ist, angezeigt. Im oberen Teil der Zeichenfläche ist ein Rechteck eingezeichnet, welches den Rucksack repräsentiert. Die dunkelgrau eingezeichnete Abmessung innerhalb des Rucksacks repräsentiert sein maximale Gewicht. Gegenstände können per drag&drop in den Rucksack und wieder heraus gezogen werden. In den Rucksack können allerdings nur so viele Gegenstände gepackt werden, bis das maximale Gewicht erreicht ist. Im Rucksack werden die Gewichtsbalken der einzelnen Gegenstände direkt aneinander gereiht. So ist anhand der eingezeichneten Abmessung auf einen Blick ersichtlich, wie viel Kapazität im Rucksack noch verfügbar ist. Die Balken, die die Werte der Gegenstände darstellen, werden darunter aneinander gereiht und zeigen so die Güte der aktuellen Lösung an. Das pluginspezifische Seitenmenü ist im *BackPack*-Plugin leer, da keine weiteren Komponenten nötig sind. Ein Beispiel der graphischen Darstellung einer Probleminstanz ist in Abbildung 4.8 zu sehen.

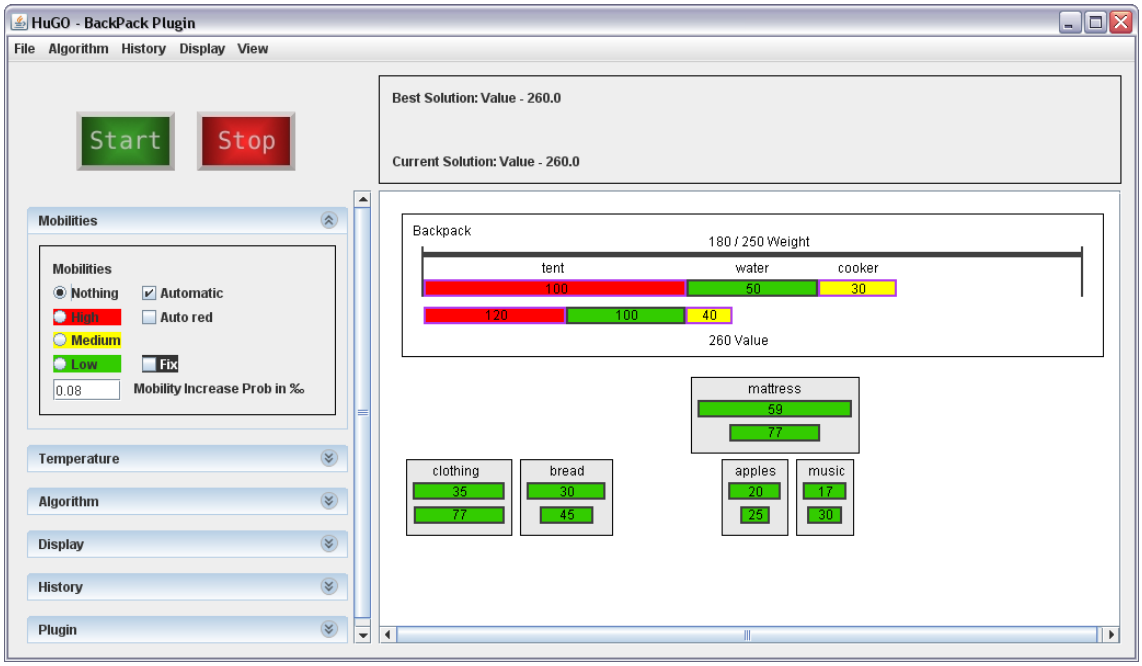

Abbildung 4.8: Eine Beispielinstanz des *BackPack*-Plugins

4.4.2 Das *CrossMin*-Plugin

In diesem Plugin wird das in Kapitel 1.4.1 definierte k-Schichten-Kreuzungsminimierungsproblem optimiert. Allerdings werden nicht nur die Kreuzungen, sondern auch diagonale Kanten minimiert. Es wird probiert, jede Kante so senkrecht wie möglich zu zeichnen. Dazu wird der Diagonalitätsgrad eingeführt, der berechnet wie diagonal ein Graph ist. Für jede Kante wird der horizontale Abstand zwischen dem oberen und dem unteren Knoten der Kante in Anzahl der Knoten, die dazwischen liegen, berechnet. Der Diagonalitätsgrad ist die Summe dieser Abstände aller Kanten. Der Optimierungsalgorithmus minimiert allerdings nur sekundär die diagonalen Kanten. Primär werden weiterhin Kreuzungen minimiert. Das heißt, hat eine Lösung eine kleinere Anzahl an Kreuzungen, so ist sie in jedem Fall besser, als eine Lösung mit mehr Kreuzungen unabhängig vom Diagonalitätsgrad.

Die Knoten des Graphen werden als Kreise dargestellt, die durch Kanten, die als Striche dargestellt werden, miteinander verbunden sein können. In der Eingabe des Graphen können einzelne Knoten als sogenannte Dummyknoten deklariert sein. Diese Knoten werden durch kleinere Kreise dargestellt. Die Bedeutung von Dummyknoten wurden in Kapitel 1.4.1 erläutert. Dummyknoten werden nur anders auf der Zeichenfläche dargestellt, in der Optimierung werden sie wie normale Knoten behandelt. Diese andere Darstellungsweise dient also nur dem Benutzer zur besseren Übersicht. Neben den Dummyknoten gibt es noch sogenannte Nullknoten, die als Abstandshalter dienen. Nullknoten sind keine realen Knoten, die in der Eingabe definiert werden und sie besitzen auch keinerlei Kanten. Diese Nullknoten füllen die Schichten soweit auf, dass alle Schichten die selbe Anzahl an Knoten enthalten. Sie können sowohl vom Benutzer als auch vom Optimierungsalgorithmus verschoben werden. Eine Veränderung des Graphen ist durch drag&drop der Knoten möglich. Dabei wird der angeklickte Knoten an die Position des Knotens verschoben, an dem der Mausdruck gelöst wird. Alle Knoten die zwischen diesen beiden Knoten liegen, werden dabei um eine Position verschoben in Richtung des veränderten Knotens. Knoten können außerdem nur innerhalb ihrer eigenen Schicht verschoben werden. In Abbildung 4.11 ist eine Beispielinstanz mit 7 Schichten abgebildet, in der auch fixierte Knoten, Nullknoten und Blöcke zu sehen sind.

Im Seitenmenü *Plugin* befinden sich im CrossMin Plugin drei Funktionen. Die ersten Beiden dienen dazu, sogenannte Blöcke zu erstellen und wieder zu entfernen. Ein Block ist eine benachbarte Menge von Knoten. Benachbarte Knoten sind in diesem Fall Knoten, die sich direkt neben- oder übereinander befinden. Durch das Drücken der STRG-Taste in Verbindung mit dem Auswählen eines Knotens, ist es möglich beliebig viele Knoten auszuwählen. Wird anschließend der *Create Block*-Button gedrückt, wird, falls diese Knoten alle benachbart sind, ein Block aus der Menge dieser Knoten erzeugt. Ein Block wird als rechtwinkliges Polygon, welches seine Knoten einschließt gezeichnet. Durch drag&drop ist ein Block genauso wie ein Knoten verschiebbar. Außerdem kann einem Block wie einem

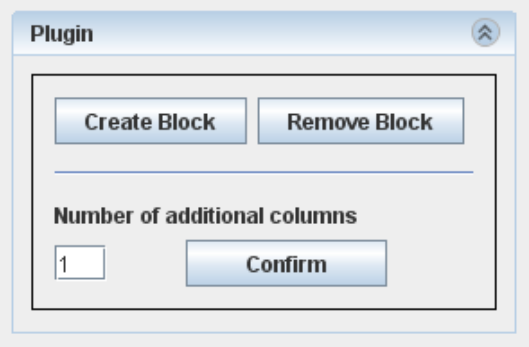

Abbildung 4.9: Das Untermenü *Plugin* des *CrossMin*-Plugins

Knoten eine Mobility zugeordnet werden und er kann fixiert werden. Durch betätigen des *Remove Block*-Buttons wird der aktuell ausgewählte Block entfernt. Knoten, die sich in einem Block befinden, verändern im Optimierungsalgorithmus ihre Position innerhalb des Blocks nicht, denn der Block kann nur als ganzes Element verschoben werden. Dadurch ist es möglich bestimmte Strukturen im Graphen zu einem Block zusammen zufassen und zu erhalten. Die Probleminstanz wird dann unter Berücksichtigung dieser Restriktion weiter optimiert.

Die dritte Funktion im pluginspezifischen Untermenü ändert die Anzahl der zusätzlichen Spalten. Zusätzliche Spalten erhöhen die Anzahl der Knoten in den Schichten um jeweils eins, indem Nullknoten eingefügt werden. Die Anzahl der zusätzlichen Spalten ist standardmäßig als 10% der Anzahl der normalen Knoten pro Schicht definiert. Erst durch Nullknoten ist eine sinnvolle Minimierung der diagonalen Kanten möglich. Auf die Kreuzungsminimierung haben diese Knoten keinerlei Einfluss. Um so größer die Anzahl der zusätzlichen Spalten ist, um so besser lassen sich die diagonale Kanten begradigen. Allerdings steigt dadurch natürlich die Breite des Graphen. Durch die Möglichkeit, die Anzahl der zusätzlichen Spalten zu setzen, kann der Benutzer flexibel und je nach Probleminstanz bestimmen, welche Faktoren wichtig sind.

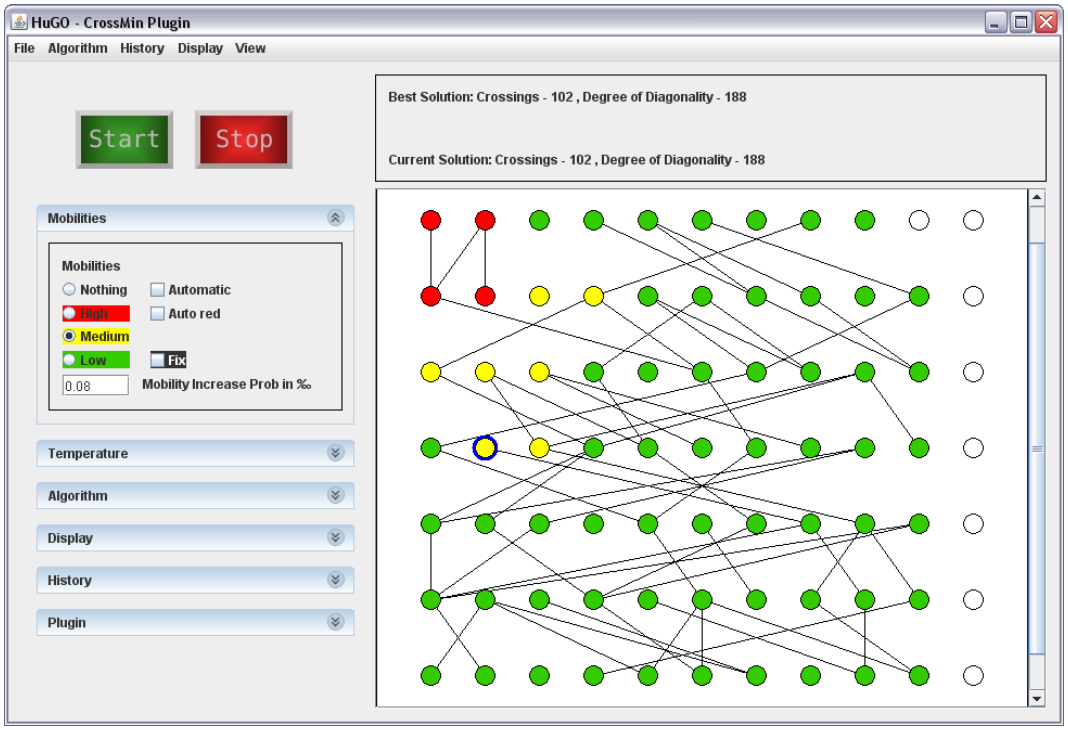

Abbildung 4.10: Eine Beispielinstanz des *CrossMin*-Plugins mit 7 Schichten und verschiedenen Mobilities

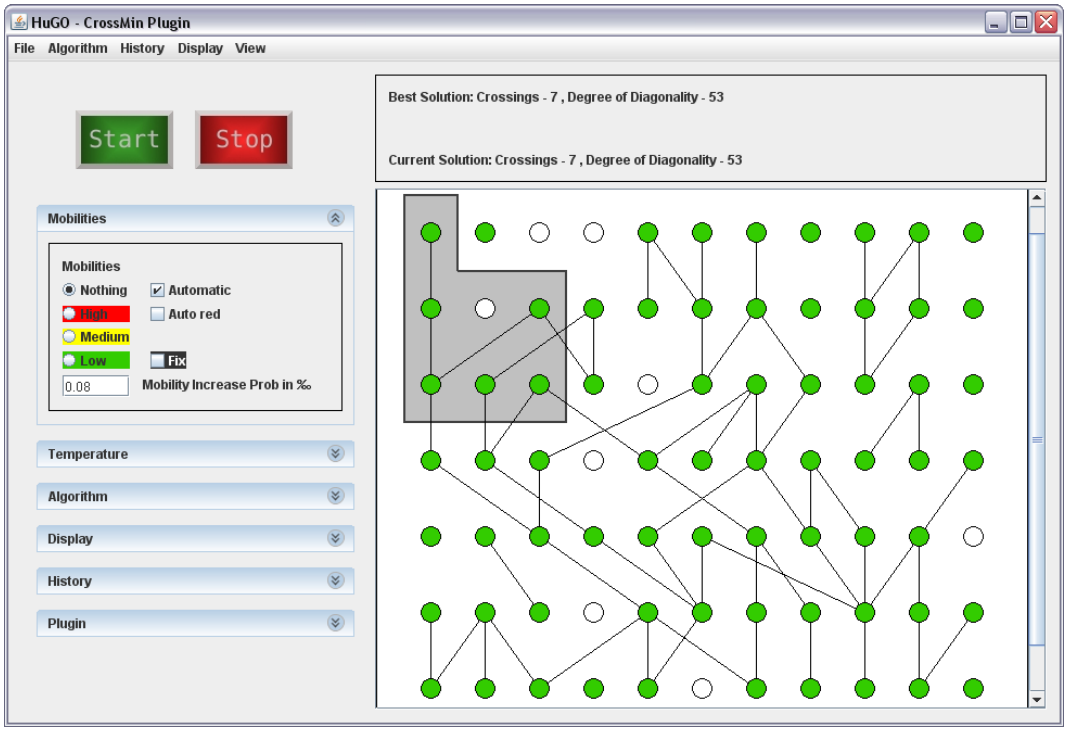

Abbildung 4.11: Eine optimierte Beispielinstanz mit einem Block

Kapitel 5

Pluginentwicklung

In diesem Kapitel wird die Entwicklung von Plugins für das Framework von *HuGO* vorgestellt. Im ersten Abschnitt wird allgemein erläutert, welche Klassen und Methoden implementiert werden müssen, um ein neues Plugin zu erstellen. Dieses wird am Beispiel des *BackPack*-Plugins deutlich gemacht. Im zweiten Abschnitt wird das *CrossMin*-Plugin vorgestellt, wobei genau erläutert wird, wie es realisiert ist.

5.1 Pluginentwicklung am Beispiel des *BackPack*-Plugins

In diesem Kapitel wird beschrieben, wie ein Plugin aufgebaut ist und welche Komponenten implementiert werden müssen, um ein neues lauffähiges Plugin zu erstellen. Dies werde ich am Beispiel des *BackPack*-Plugins erläutern. Im *BackPack*-Plugin wird das in Kapitel 1.4.2 definierte Rucksackproblem optimiert. Eigentlich ist dieses Problem nicht besonders gut geeignet, um mit Hilfe von *Human Guided Optimization* gelöst zu werden, denn es gibt, schnelle und gute Approximationsalgorithmen für dieses Problem. Weiterhin wird die visuelle Darstellung für den Benutzer mit steigender Problemgröße schnell unübersichtlich. Ich habe dieses Problem trotzdem ausgewählt, weil es sehr einfach und intuitiv ist. Es dient als Beispiel um einen Überblick über den Aufbau eines Plugins zu ermöglichen. Ein für HuGO gut geeignetes Problem, für das ebenfalls ein Plugin implementiert wurde, ist die k-Schichten-Kreuzungsminimierung, dazu in Kapitel 5.2.

Im Quellcode von *HuGO* befindet sich ein Paket namens `template`, in dem sämtliche Klassen, die für ein neues Plugin überschrieben werden müssen enthalten sind. Diese Klassen wurden dort schon so weit wie möglich implementiert und mit genauen Kommentaren versehen, welche Komponenten für ein neues Plugin ergänzt werden müssen. Weiterhin sind einige pluginspezifische Teile, wie die Methoden zum Auswählen von Knoten oder das Layout der Informationsanzeige, bereits implementiert. Mit Hilfe dieses Pakets sollte es einem Entwickler schnell möglich sein für einfache Optimierungsprobleme ein Plugin zu schreiben.

In Abbildung 5.1 ist ein Klassendiagramm abgebildet, in dem die Struktur der abstrakten Klassen erkennbar ist, die bei der Entwicklung eines neuen Plugins implementiert werden müssen.

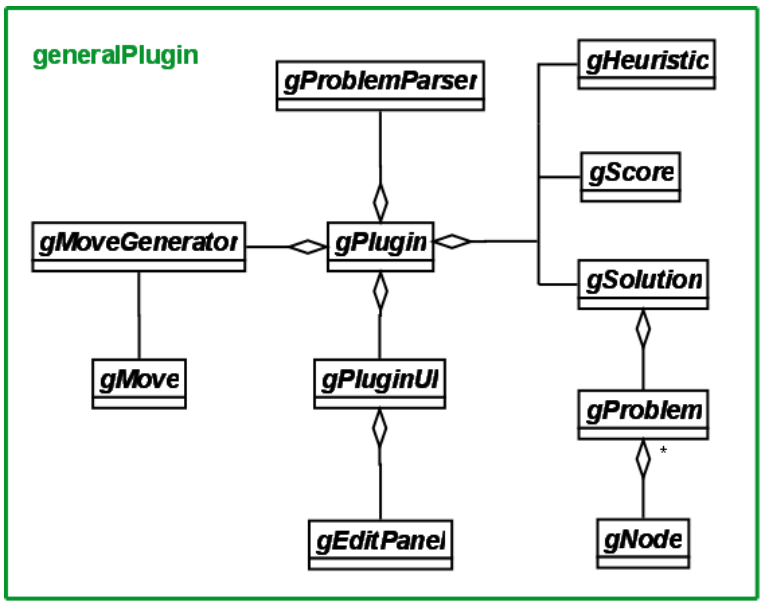

Abbildung 5.1: Ein Klassendiagramm in dem die Struktur der abstrakten Klassen, die bei der Entwicklung eines neuen Plugins implementiert werden müssen.

5.1.1 Datenstruktur

Die Datenstruktur eines Plugins besteht aus mehreren vorgegebenen Klassen, die von den abstrakten Klassen erben, die in Abbildung 5.1 dargestellt sind. Im Folgenden werden diese Klassen vorgestellt und es wird erläutert, welche Komponenten für ein neues Plugin am Beispiel des Rucksackproblems ergänzt werden müssen.

Plugin

Die Klasse `Plugin` ist die Hauptklasse der Datenstruktur und verwaltet alle anderen Klassen des Plugins. Das Plugin kennt seine aktuelle Lösung, die ein Objekt der Klasse `Solution` ist. Für die Klasse `Plugin` des *BackPack*-Plugins sind keine weiteren Änderungen nötig, als im Konstruktor Instanzen der pluginspezifischen Klassen `MoveGenerator`, `Score`, `ProblemParser`, `Heuristic` und `PluginUI` zu erzeugen und zu setzen.

Damit ein neu implementiertes Plugin im Auswahldialog *ChoosingPluginFrame* vom Benutzer ausgewählt werden kann, muss die Erzeugung eines Objekts der Klasse *Plugin* dort eingefügt werden. Genauer muss in das Array *plugins* des Name des Plugins eingefügt werden. Außerdem muss im lokalen *ActionListener* abgefragt werden, ob das ausgewählte Element der *jComboBox* den Namen des neuen Plugins liefert. Ist dies der Fall muss

ein Objekt der entsprechenden Klasse *Plugin* erzeugt und in einer vorgegebenen Variable gespeichert werden.

Node

Jedes zu optimierende Problem, muss unter anderem aus einer Menge sogenannter Knoten bestehen, deren genaue Bedeutung im Problem pluginspezifisch ist. Knoten werden implementiert, indem von der Klasse `gNode` geerbt wird. Wie in Kapitel 3.1 erläutert enthalten die Knoten einige vorgegebene Variablen, wie beispielsweise eine Mobility, ihre Position und ihren Namen, sowie dazugehörige Methoden, um diese Variablen abzufragen und zusetzen. Überschrieben werden muss nur die Methode `clone`. Alle weiteren Ergänzungen sind pluginspezifisch und ohne Vorgaben.

In unserem Beispiel, dem *BackPack*-Plugin, stellen die Knoten die Gegenstände dar. Dazu müssen sie die Eigenschaften der Gegenstände speichern. Gegenstände haben ein bestimmtes Gewicht und einen Wert, also werden diese beiden Werte als Variablen eingefügt. Außerdem wird gespeichert, ob sich ein Gegenstand im Rucksack oder außerhalb befindet. Um auf diese Eigenschaften zugreifen zu können, werden Methoden zum Abfragen hinzugefügt, während das Setzen nur über den Konstruktor möglich ist, denn ein Gegenstand ändert sein Gewicht und Nutzen nicht. Damit besitzt der Knoten alle nötigen Eigenschaften, um einen Gegenstand zu repräsentieren.

Im *BackPack*-Plugin sind keine weiteren Klassen für die Datenstruktur nötig. Man könnte eine Klasse implementieren, die den Rucksack darstellt. Allerdings habe ich dies ohne eine weitere Klasse durch die Variable in den Knoten gelöst. Diese Variable gibt an, ob sich ein Knoten im oder außerhalb des Rucksacks befindet. In anderen Plugins kann es hier nötig sein weitere Klassen zu erstellen. Dies werden wir Kapitel 5.2 im CrossMin Plugin sehen.

Problem

Die Knoten müssen nun in einer Klasse zusammengefasst werden, die das Problem darstellt. Dies geschieht in einer Unterklasse von `gProblem`. Wie schon in Kapitel 3.1 erläutert, enthält diese Klasse eine Liste aller Knoten und verschiedene Methoden, um auf diese zugreifen zu können. Es muss nur die Methode `clone` überschrieben werden.

Doch um ein Optimierungsproblem vollständig zu konstruieren reicht dies in der Regel nicht aus. Selbst in unserem *BackPack*-Plugin müssen wir noch einiges ergänzen. Der Rucksack hat ein vorgegebenes maximales Gewicht, das die Gegenstände im Rucksack nicht überschreiten dürfen. Außerdem ergibt sich aus den Gegenständen, die sich gerade im Rucksack befinden, ein aktuelles Gewicht und ein aktueller Wert, der die Güte der Lösung darstellt. Dies wird in entsprechenden Variablen gespeichert, zu denen auch Methoden zum Abfragen und Setzen implementiert werden. Weiterhin gibt es noch zwei Variablen, die für

die graphische Darstellung der Gegenstände im Rucksack wichtig sind. Sie beschreiben, die nächste freie Position, an die der neue Gegenstand im Rucksack gezeichnet werden kann.

Um die Verschiebung in den Rucksack bzw. aus ihm hinaus zu berechnen gibt es zwei Methoden. Die Methode `shiftNodeOutOfBackPack` verschiebt den übergebenen Knoten aus dem Rucksack hinaus. Dem Knoten wird seine Position für außerhalb als aktuelle Position zugewiesen und die Variable, die seine Anwesenheit im Rucksack beschreibt, wird geändert. Jeder Knoten besitzt einerseits, wie schon erwähnt, eine aktuelle Position und andererseits besitzt er eine sogenannte `outPosition`, in der seine Position außerhalb des Rucksacks fest gespeichert wird. Diese Position wird in der Klasse `JNode`, die in einem der folgenden Abschnitte erläutert wird und die, für die graphische Darstellung eines Knotens, erzeugt wird, gespeichert. Außerdem wird das aktuelle Gewicht und der aktuelle Wert des Rucksacks um das Gewicht und den Wert des Knotens gesenkt. Nun werden die im Rucksack verbliebenen Knoten neu angeordnet, um die evtl. entstandene Lücke zu schließen. Dazu werden alle Knoten im Rucksack einmal durchlaufen und mit Hilfe der oben erwähnten Variablen auf den nächsten freien Platz gesetzt.

In der Methode `shiftNodeInBackPack` wird der übergebene Knoten in den Rucksack verschoben. Dazu wird ihm eine neue Position im Rucksack zugewiesen und seine Anwesenheitsvariable wird entsprechend gesetzt. Die Position wird wieder anhand der Variablen für die nächste freie Position bestimmt. Außerdem wird das Gewicht und der Wert des Rucksacks angepasst, indem das Gewicht und der Wert des eingefügten Knotens addiert werden. Eine Überprüfung auf Gültigkeit dieser Verschiebung findet nicht in den Methoden statt und muss vor Benutzung der Methoden überprüft werden.

Zwei weitere Methoden `getNodesNotAtBackPack` und `getNodesAtBackPack` liefern jeweils eine Liste mit den Knoten außerhalb und innerhalb des Rucksacks. Diese Listen werden in der Methode erzeugt, denn in der Klasse `Problem` wird nur eine Liste aller Knoten geführt. Die Methode `checkIfPossible` überprüft, ob das übergebene Gewicht in den Rucksack passen würde ohne das maximale Gewicht zu überschreiten.

Solution

Eine `Solution` beinhaltet ihr zugehöriges `Problem`, sowie einige Variablen, wie einen Namen, den Zeitpunkt der Erstellung und die benötigte Größe auf der graphischen Oberfläche. Der Name und der Zeitpunkt der Erstellung dienen zur Identifikation im Verlauf. Außerdem existiert noch eine Variable, die manuell vom Benutzer gespeicherte Lösungen, von denen unterscheidet, die automatisch gespeichert werden. Die Klasse `Solution` erbt von `gSolution` und muss die folgenden Methoden überschreiben. In `updateUISize` wird die auf der Zeichenfläche der graphischen Oberfläche benötigte Größe berechnet und mit den Methoden `setUiWidth` und `setUiHeight` gesetzt. In unserem Beispiel *BackPack* ist

die Weite konstant, während die Höhe abhängig von der Anzahl und der Größe der Gegenstände der jeweiligen Probleminstanz ist.

Die Methode `getValue` liefert die Güte der Lösung, deren Berechnung pluginspezifisch ist. Im *BackPack*-Plugin ist die Güte der Lösung definiert, als der Wert der im Rucksack vorhandenen Gegenstände und wird in der Klasse `Problem` gespeichert. Die Methoden `getInfosForStopButton`, `getInfosForInfoLabel` und `getInfosForHistoryName` geben alle einen String zurück, der über dem *StopButton*, in der Informationsanzeige oder als Name im Verlauf angezeigt wird. Im *BackPack*-Plugin wird jeweils die Güte der Lösung, die aus dem aktuellen Wert der Gegenstände im Rucksack besteht, angezeigt. Der Name im Verlauf besteht zusätzlich aus dem Speicherzeitpunkt der Lösung, damit der Benutzer die Lösungen besser unterscheiden kann. Für den Optimierungsalgorithmus, der eine Tabuliste führt, ist es nötig in etwa zu wissen, wie viele Moves auf der aktuellen Lösung möglich sind. Eine untere Schranke dafür liefert die Methode `getLowerBoundOfNumberOfPossibleMoves`. So wird verhindert, dass die Tabuliste so groß wird, dass keine Move mehr möglich ist. Außerdem kann so automatisch eine sinnvolle Größe der Tabuliste abhängig von der Problemgröße berechnet werden. Auch in der Klasse `Solution` muss die Methode `clone` implementiert werden, um Kopien von Lösungen zum Beispiel für den Verlauf und den Optimierungsalgorithmus herstellen zu können.

5.1.2 Moves

Ein Move stellt eine Änderung der Lösung, genauer eine Änderung einiger Knoten der Probleminstanz, dar. Diese Moves werden in der Klasse `MoveGenerator` erzeugt.

Move

Die Klasse `gMove` stellt die Oberklasse für alle Moves dar. Ein Move besitzt eine Liste der Knoten, die in diesem Move geändert werden und die dazu gehörigen Methoden, die die Liste zurückgeben und Knoten einfügen. Ein Move im *BackPack*-Plugin muss auf eine bestimmte Weise Gegenstände aus dem Rucksack entfernen bzw. einfügen. Hier wären zwei verschiedene Moves, einer zum Einfügen und einer zum Entfernen, denkbar. Doch die Moves zum Entfernen würden die Lösung immer verschlechtern und ihre Wahrscheinlichkeit ausgeführt zu werden, wäre geringer. Man könnte also einen Move implementieren, der einen Gegenstand in den Rucksack einfügt und einen anderen entfernt. Allerdings tritt hier das Problem auf, dass der Gegenstand der heraus genommen wird, immer mindestens genauso schwer sein muss, wie der, der eingefügt wird. So kann es zu einem Zustand kommen, in dem bestimmte Gegenstände nicht mehr eingefügt werden können. Bei der Implementierung der Moves ist es wichtig zu beachten, dass durch eine Reihe von Moves alle möglichen Lösungen erreicht werden können. Es können beliebig viele verschiedene Moves konstruiert werden. Ich habe mich aufgrund dieser Problematik für einen sogenannten

`SwapMove` entschieden, der einen Gegenstand in den Rucksack einfügt und beliebig viele andere Gegenstände dafür aus dem Rucksack entfernt.

Der `SwapMove` besteht aus einem Knoten der eingefügt wird und einer Liste von Knoten die aus dem Rucksack entfernt werden. Diese werden im Konstruktor übergeben und dabei wird überprüft, ob sich der Knoten der eingefügt werden soll, aktuell außerhalb und die Knoten die entfernt werden sollen, aktuell innerhalb des Rucksacks befinden. Ist dies nicht der Fall, wird eine `Exception` geworfen. Außerdem werden alle Knoten in die interne Liste der Moves die verändert werden eingefügt. Für die einzufügenden und zu entfernenden Knoten gibt es außerdem Methoden, die diese zurückgeben. Um den neuen Move für das Plugin zu vervollständigen, müssen die folgenden Methoden überschrieben werden. Die Methode `getMobility` liefert die Mobility des Moves, die aus der Mobility der Knoten des Moves berechnet werden sollte. Im *BackPack*-Plugin bilden wir den Durchschnitt der Mobilities der beteiligten Knoten. Ist allerdings ein Knoten mit niedriger Mobility enthalten, wird die Mobility des Moves auch niedrig gesetzt. Dies ist wohl eine der einfachsten Varianten, die Mobility zu berechnen und lässt sich auf alle denkbaren Moves übertragen. Doch um die Flexibilität bei der Entwicklung neuer Plugins zu erhalten, ist diese Berechnung pluginspezifisch und kann sich auch, je nach Move, unterscheiden. So wäre es zum Beispiel auch möglich, das Mobility Verhalten von *HuGS*, das in Kapitel 2.1.3 beschrieben wurde, zu simulieren. Moves die nur Knoten mittlerer Mobility enthalten, würden eine niedrige Mobility zugewiesen bekommen, während Moves mit Knoten mittlerer und hoher Mobility eine hohe Mobility zugewiesen bekommen würden.

Die wichtigste Methode eines Moves ist die Methode `doMove`. Sie bekommt eine Lösung übergeben, auf der der Move ausgeführt wird. Im `SwapMove` des Rucksackproblems werden dazu die, in Kapitel 5.1.1 erläuterten, Methoden `shiftNodeOutOfBackPack` und `shiftNodeInBackPack` der Klasse `Problem` genutzt. Die Methode `isEqual` vergleicht, ob das übergebene Objekt ein identischer Move ist. Dazu wird überprüft, ob die gleichen Knoten eingefügt und entfernt werden.

Um die in Kapitel 3.2.5 beschriebene Tabuliste möglichst sinnvoll zu füllen, müssen alle Moves die Methode `createReverseMove` überschreiben. Sie gibt einen Move zurück, der in die Tabuliste eingefügt wird, wenn der aktuelle Move ausgeführt wurde. In unserem Beispiel `SwapMove` wird dabei der Move selbst zurückgegeben, da eine Umkehrung des Moves nicht möglich ist, da mit einem `SwapMove` nur ein Gegenstand in den Rucksack eingefügt werden kann.

MoveGenerator

Der `MoveGenerator` erzeugt die Moves, die der Optimierungsalgorithmus ausführen kann. Wie und welche Moves erzeugt werden ist pluginspezifisch. Zur Implementierung dieser

Klasse müssen einige abstrakte Methoden der Oberklasse `gMoveGenerator` überschreiben werden.

Die Methode `generateNextMove` bekommt eine Lösung übergeben und generiert für diese Lösung den nächsten Move, der zurückgegeben wird. Im *BackPack*-Plugin wird in dieser Methode als erstes der einzufügende Gegenstand ausgewählt. Dazu werden alle Gegenstände, die sich außerhalb des Rucksacks befinden betrachtet, um dann mit Hilfe einer Zufallszahl, die in der Klasse `MyRandom` erzeugt wird, einen davon auszuwählen. Gegenstände deren Gewicht größer ist als das maximal zulässige werden nicht ausgewählt. Anschließend werden alle Gegenstände innerhalb des Rucksacks solange durchlaufen, bis so viele ausgewählt wurden, dass das zulässige Gewicht des Rucksacks nach Einfügen des ersten Gegenstands nicht überschritten wird. Falls es dazu nicht nötig ist einen Gegenstand zu entfernen, bleibt die Liste der zu entfernenden Gegenstände leer.

Weiterhin müssen die in Kapitel 3.2.7 bereits erwähnten Methoden `getNeighborhood` und `getNeighborhoodMoves` im `MoveGenerator` überschrieben werden. Die erste Methode bekommt eine Lösung und einen Knoten übergeben und gibt die aktuelle Nachbarschaft dieses Knotens in Form einer Liste mit Knoten zurück. Die zweite Methode bekommt ebenfalls eine Lösung und einen Knoten übergeben und gibt eine Liste mit Moves zurück. In dieser Liste befinden sich alle Moves, die den übergebenen Knoten und seine direkte Nachbarschaft betreffen. Im *BackPack*-Plugin ist die Funktion der lokalen Optimierung nicht implementiert, da es keine sinnvollen Nachbarschaften gibt. Diese Funktion ist für Plugins sinnvoll, in denen die Position der Knoten abhängig von ihren Nachbarknoten wichtig ist.

5.1.3 Score

Die Klasse `Score` vergleicht zwei Lösungen auf ihre Güte. Dies geschieht pluginspezifisch, denn obwohl jede Lösung einen festgelegten Wert hat, der über die Methode `getValue` zurückgegeben wird, muss dieser Wert nicht zwangsweise als Vergleichskriterium übernommen werden. Dies bietet dem Entwickler des Plugins mehr Flexibilität. Im *BackPack*-Plugin findet, in den zwei zu überschreibenden Methoden `isBetterOrEqual` und `isReallyBetter`, ein einfacher Vergleich des Werts den `getValue` liefert statt. Im Paket `template` findet in der Klasse `Score` ebenfalls ein einfacher Vergleich des Wertes, den `getValue` liefert, statt. Falls der Entwickler des Plugins keine Änderungen vornehmen möchte, muss in dieser Klasse nichts ergänzt werden.

5.1.4 ProblemParser

Da in *HuGO* kein Editor zur Erstellung von Probleminstanzen für die einzelnen Plugins vorgesehen ist, müssen diese anders eingelesen werden. In der Klasse `ProblemParser` in der

Methode `readSolution` wird aus dem übergebenen String eine Probleminstanz erzeugt und zurück gegeben.

In unserem *BackPack*-Plugin muss dieser String wie folgt codiert sein. Der String „$N = (n_1, w_1, v_1)(n_2, w_2, v_2)...(n_k, w_k, v_k)M = maxW$“wird zu einer Probleminstanz mit k Gegenständen und einem Rucksack mit maximalen Gewicht $maxW$ decodiert. Der i-te Gegenstand trägt den Namen n_i, hat ein Gewicht von w_i und einen Wert von v_i. Kommasetzung und Leerzeichen müssen bei der Codierung beachtet werden.

Zur Erzeugung der Probleminstanz wird der übergebene String in einer Schleife solange durchlaufen, bis an Stelle des nächsten Gegenstandes der Buchstabe 'M' gefunden wird. In der Schleife wird der Substring mit Hilfe der `indexOf` Methode anhand der Kommata eingeteilt, um die Informationen über Namen, Gewicht und Nutzen auszulesen. Nach beenden der Schleife wird das maximale Gewicht ausgelesen und in der Probleminstanz gesetzt. Nun ist das Problem mit allen nötigen Informationen erstellt. Um eine übersichtlichere Darstellung auf der graphischen Oberfläche zu erreichen, werden die Gegenstände nach ihrem Gewicht absteigend sortiert. Dabei werden auch die Identifikationsnummern der Knoten in dieser Reihenfolge gesetzt. Anschließend werden den Gegenständen ihre Positionen auf der graphischen Oberfläche, wenn sie sich außerhalb des Rucksacks befinden, zugewiesen. Tritt während des Decodierens ein Fehler auf, wird eine Exception geworfen. Diese wird in der Methode, die `readSolution` aufruft, abgefangen und erzeugt einen Fehlerdialog.

5.1.5 Heuristik

In der Klasse `Heuristic` kann eine Heuristik für das jeweilige Problem implementiert werden. Diese Heuristik sollte eine überschaubare Laufzeit besitzen, denn sie dient in erster Linie dazu, dem eigentlichen Optimierungsalgorithmus eine gute Startlösung zu liefern. Die Implementierung einer Heuristik ist also für ein Plugin nicht zwingend notwendig. Für eine fehlerfreie Anwendung von *HuGO* ist es allerdings nötig, mindestens die entsprechende Klasse `Heuristic` zu erstellen und die Methode `getSolution` zu überschreiben. Dazu kann die Klasse aus dem Paket `template` übernommen werden.

Im BackPack Plugin ist eine Greedy Heuristik implementiert. Die Gegenstände werden nach ihrem Nutzen absteigend sortiert. Der Nutzen des Gegenstands i ist definiert als $b_i = \frac{v_i}{w_i}$, wenn v_i der Wert und w_i das Gewicht des Gegenstands ist. Nun werden die Gegenstände nach dieser Reihenfolge in den Rucksack gepackt, bis der nächste Gegenstand nicht mehr eingefügt werden kann. Diese Lösung wird zurück gegeben. [13]

5.1.6 Graphische Darstellung

Das Grundgerüst der graphischen Oberfläche wird, wie in Kapitel 3.3 erläutert, in der Klasse `HuGOUI` aufgebaut. Die Zeichenfläche, auf der die Probleminstanz dargestellt wird

und das `JPanel` im Seitenmenü sind allerdings pluginspezifisch. Außerdem kann der Entwickler des Plugins die Informationsanzeige über der Zeichenfläche frei gestalten.

PluginUI

In der Klasse `PluginUI` werden alle nötigen Komponenten für den pluginspezifischen Teil der graphischen Oberfläche erzeugt. Diese werden von der pluginübergreifenden Klasse `HuGOUI` abgefragt und in das Gesamtlayout einfügt. `PluginUI` ist Unterklasse von `gPluginUI` und muss drei abstrakte Methoden überschreiben, die je ein Panel erzeugen. Die Funktionen und das Layout der drei Panels werden im Folgenden erläutert.

Das wichtigste Panel ist die Zeichenfläche, auf der die Probleminstanz gezeichnet wird und auf der sie auch manuell geändert werden kann. Dafür wird ein Objekt der Klasse `EditPanel` erzeugt, dem ein `MouseListener` und ein `KeyListener`, die lokal implementiert werden, hinzu gefügt werden. Dies wird im folgenden Abschnitt genauer beschrieben. Ein weiteres Panel wird für die Informationsanzeige oberhalb der Zeichenfläche erstellt. Hier können zum Beispiel die Daten zur aktuellen Lösung angezeigt werden. Außerdem empfiehlt es sich, die Informationen der besten bisher gefundenen Lösung dort anzeigen zu lassen. Im *BackPack*-Plugin wird jeweils der Wert der aktuellen und der besten Lösung dort angezeigt. In dem, in Kapitel 3.3 beschriebenen, Seitenmenü ist ein Untermenü für pluginspezifische Komponenten vorgesehen. Dieses Untermenü zeigt das mit in der Methode *createPluginTaskPanel* erzeugten Panel an. Im *BackPack*-Plugin erzeugt diese Methode nur ein leeres Panel, da keine weiteren Komponenten nötig sind. Weiterhin müssen zwei Methoden überschrieben werden, die die Informationsanzeige aktualisieren. Eine der `updateInfoPanel` Methoden bekommt dabei einen String übergeben, in dem die Informationen über die bisher beste Lösung gespeichert sind. In der anderen Variante wird nichts übergeben, denn hier werden nur die Informationen über die aktuelle Lösung aktualisiert. Diese Methoden werden bei Änderungen an der aktuellen Lösung oder, wenn eine neue beste Lösung gefunden wurde, aufgerufen.

EditPanel

Die Klasse `EditPanel` ist eine Unterklasse von `gEditPanel`, diese wiederum erbt von `JPanel`. Somit kann die Zeichenfläche, durch überschreiben der Methode `paintComponent`, gefüllt werden. In dieser Methode werden alle Knoten durchlaufen, um ihre `paintMe` Methoden aufzurufen. Im *BackPack*-Plugin wird außerdem die Methode `paintMe` der Klasse `JBackPack` aufgerufen, um den Rucksack zu zeichnen.

Um Veränderungen durch den Benutzer zu ermöglichen, müssen MouseEvents behandelt werden. Dies geschieht in den drei Methoden `organizeMouseClick`, `organizeMousePress` und `organizeMouseRelease`. Diese Methoden sind jedoch nicht durch die Oberklasse vorgegeben. Ob und wie eine Behandlung von MouseEvents statt findet und wie die

Änderungen der Lösung durch den Benutzer implementiert werden, ist pluginspezifisch. Im Falle des *BackPack*-Plugins habe ich mich dafür entschieden, mit Hilfe dieser drei Methoden die Behandlung von MouseEvents aus der Klasse `PluginUI` in das `EditPanel` zu verlagern und dort zu implementieren. In den drei Methoden wird jeweils überprüft, ob das übergebene MouseEvent einen Knoten getroffen hat. Ist dies der Fall werden die Zustände der Knoten angepasst. Wurde ein Knoten angeklickt wird die Variable gesetzt, die ihn als markiert kennzeichnet. Durch die Markierung wird der Knoten in der graphischen Darstellung entsprechend hervorgehoben. Ein gedrückter Knoten wird ebenfalls hervorgehoben, außerdem wird er zwischengespeichert, um bei der Reaktion auf das Loslassen der Maus auf ihn zurückgreifen zu können. Genaueres zur graphischen Darstellung der Gegenstände wird im folgenden Abschnitt beschrieben. Mit Loslassen der Maustaste wird überprüft, ob es einen, zuvor als gedrückt gespeicherten Knoten, gibt. Wenn dies der Fall ist, wird abgefragt, ob die Position der Maus, eine Verschiebung des, durch den Knoten repräsentierten Gegenstandes, mit sich bringt.

Wichtig bei der Implementierung der Methoden, die für die benutzergesteuerte Änderung der Lösung verantwortlich sind, ist, dass die Liste der ausgewählten Knoten im `EditPanel` immer aktuell gehalten wird. Denn die Änderung der Mobilities und das Fixieren von Knoten wird in der Klasse `HuGOUI` automatisch durch das Auslesen dieser Liste ausgeführt. Entsprechende Methoden zum Hinzufügen und Entfernen sind in der Klasse `gEditPanel` vorgegeben.

JNode

Die Klasse `JNode` ist im *BackPack*-Plugin frei implementiert, das bedeutet, dass es gibt keinerlei Vorgaben gibt. Die Trennung der graphischen Darstellung einer Komponente von seinen funktionalen Funktionen ist aber empfehlenswert. Im Paket `template` ist dieser Ansatz ebenfalls realisiert und kann zur einfachen Konstruktion eines Plugins übernommen werden. Die Klasse `JNode` erbt von `Node` und implementiert die zur graphischen Darstellung benötigten Methoden.

Im *BackPack*-Plugin werden die Gegenstände als Rechtecke dargestellt, deren Größe abhängig vom Gewicht des Gegenstandes ist. Ein `JNode` speichert Höhe und Breite seines Gegenstands. Außerdem wird die sogenannte `outPosition` des Gegenstands gespeichert, die die Position außerhalb des Rucksacks darstellt. Diese `outPosition` wird nur bei der Erzeugung der Probleminstanz gesetzt und ist danach konstant, denn die Positionen der Gegenstände außerhalb des Rucksacks sind fest. So kann einem Gegenstand, der im Rucksack war, und der wieder raus geschoben wird, einfach seine alte Position wieder zugewiesen bekommen. Dadurch verändert sich die Anordnung der Gegenstände außerhalb des Rucksacks nicht und der Benutzer bekommt eine bessere Übersicht.

In der Methode `isMouseHit` wird ein `MouseEvent` übergeben und es wird überprüft, ob der Knoten getroffen wurde. Dies geschieht, wenn sich die Maus innerhalb des Rechtecks, das den Knoten repräsentiert, befindet. Dabei muss beachtet werden, dass sich die Positionen abhängig vom Zoomfaktor, der in `properties` gespeichert ist, ändern kann. Jede Positionsangabe muss mit dem Zoomfaktor multipliziert werden.

Die wichtigste Methode in dieser Klasse ist die Methode `paintMe`, die ein Objekt der Klasse `Graphics` übergeben bekommt. Hier wird die Komponente, die der Knoten repräsentiert, gezeichnet. Dabei sollten die verschiedenen Zustände, in denen sich ein Knoten befinden kann, beachtet werden. Wie diese Zustände hervorgehoben werden bleibt dem Entwickler des Plugins überlassen. Im BackPack-Plugin sind Gegenstände, die mit der Maus markiert wurden blau umrandet und jene die im letzten Schritt vom Algorithmus geändert wurden, sind gelb umrandet. Gegenstände die gerade angeklickt sind um verschoben zu werden, sind komplett blau gefärbt. Die Mobilities der Gegenstände sind durch einen grünen, gelben oder roten Rand dargestellt und fixierte Knoten sind durch einen kleinen grauen Kreis im Rechteck markiert. Außerdem wird, falls die Funktion aktiviert ist, der Name des Gegenstandes, der seinen Nutzen angibt, gezeichnet.

Weiterhin muss die Methode `clone` überschrieben werden, um zu gewährleisten, dass alle relevanten Daten beim Kopieren erhalten bleiben.

5.2 Das *CrossMin*-Plugin

Im *CrossMin*-Plugin wird das in Kapitel 1.4.1 beschriebene Problem der k-Schichten-Kreuzungsminimierung behandelt. Ich habe mich aus mehreren Gründen dafür entschieden dieses Problem als Plugin zu realisieren. Zuerst einmal sollte es für das Problem eine für den Benutzer übersichtliche, am besten leicht und intuitiv verständliche, graphische Darstellung geben. Außerdem sollte das Bearbeiten und Verändern der Lösung einfach ausführbar sein, denn ohne dies ist von Benutzerseite aus kein effektives Optimieren möglich. Weiterhin kann nicht erwartet werden, mit *HuGO* bessere und schnellere Ergebnisse zu berechnen, als mit einem Algorithmus der problemspezifisch entwickelt wurde. Es sollte außerdem ein Problem betrachtet werden, bei dem der Benutzer, nicht klar zu formulierende Restriktionen und Präferenzen realisieren möchte. Die graphische Darstellung von Graphen ist intuitiv verständlich und leicht darstellbar. Die k-Schichten-Kreuzungsminimierung ist ein wichtiger Bestandteil des automatischen Graphenzeichnens, und dort sind unklar formulierte Restriktionen und Präferenzen des Benutzers vorstellbar. Durch *HuGO* hat der Benutzer die Möglichkeit, die Lösungen nach seinen Wünschen zu verändern und trotzdem die Kreuzungen zu minimieren.

Bei der Entwicklung der *CrossMin*-Plugins kam die Idee auf, nicht nur die Kreuzungen zu optimieren, sondern auch schräge Kanten zu vermeiden. Dies gehört nicht mehr zu dem eigentlichen Problem der k-Schichten-Kreuzungsminimierung, sondern zum Layout des

automatischen Graphenzeichnens. Um dies zu realisieren, werden sogenannte Nullknoten erzeugt, die als Platzhalter dienen und mit denen die Schichten im Graphen aufgefüllt werden, so dass alle Schichten immer die selbe Anzahl an Knoten enthalten. Um die Möglichkeiten für das Layout des Graphen zu erweitern, wird eine Funktion zur Verfügung gestellt, die dem Graphen zusätzliche Spalten, gefüllt mit Nullknoten, hinzufügt. Dadurch wird der Graph zwar breiter, allerdings können bis zu einem gewissen Grad, dadurch die Anzahl der schrägen Kanten weiter verringert werden. Die Kreuzungsminimierung wird dadurch nicht verändert, da die schrägen Kanten nur sekundär optimiert werden und Nullknoten, die keine Kanten besitzen, die Kreuzungsanzahl nicht ändern.

Um die Möglichkeiten, die Optimierung zu beeinflussen, zu erweitern, wurde eine Funktion eingebaut, die sogenannte Blöcke erstellt. Dazu wählt der Benutzer eine beliebige Menge an Knoten, die neben- oder übereinander liegen aus, und erzeugt für diese Knotenmenge einen Block. Knoten, die sich in einem Block befinden, werden nicht mehr untereinander ausgetauscht. Der Block wird nur noch zusammen vom Optimierungsalgorithmus versetzt. Durch diese Funktion ist es dem Benutzer möglich, lokale Strukturen im Graphen zu erhalten, während der Algorithmus weiter optimiert. In den folgenden Abschnitten werde ich erläutern, wie das *CrossMin*-Plugin realisiert wurde.

5.2.1 Datenstruktur

In Abbildung 5.2 sind alle Klassen des *CrossMin*-Plugins in einem Klassendiagramm dargestellt.

Plugin

Die Klasse `Plugin` beinhaltet alle pluginspezifischen Klassen und verwaltet diese. Dazu werden ein `MoveGenerator`, eine `Score`, ein `ProblemParser`, eine `Heuristic` und eine `PluginUI` erzeugt, auf die die allgemeinen Klassen über die Methoden der Oberklasse `gPlugin` zugreifen können. Da für das *CrossMin*-Plugin die Funktion der lokalen Optimierung implementiert ist, wird diese Funktion hier aktiviert.

Node

Die Knoten stellen im *CrossMin*-Plugin die Knoten des Graphen dar. Zur Repräsentation dieser Knoten werden eine Reihe von Variablen benötigt. Zuerst einmal ist ein Knoten durch seine Position im Graphen definiert. Er befindet sich auf einer der vorhandenen Schichten auf einer festgelegten Position. Diese Positionen werden durch zwei Variablen gespeichert, die Nummer der Schicht und die Nummer der Position auf dieser Schicht darstellen. Weiterhin kann jeder Knoten mit beliebig vielen anderen Knoten durch Kanten verbunden sein. Um dies übersichtlicher zu gestalten, gibt es zwei Listen mit den inzidenten Kanten. Zum einen für Kanten die in die Schicht oberhalb des Knotens führen und zum

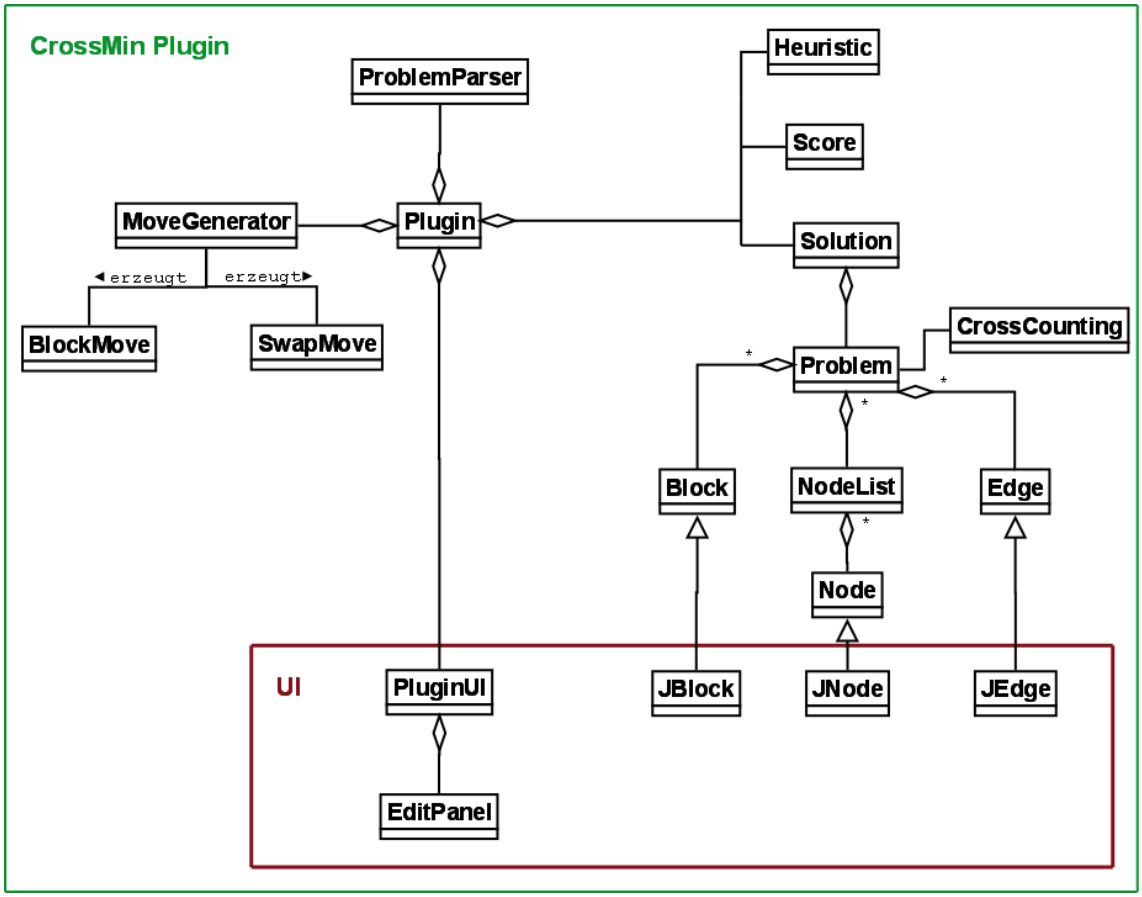

Abbildung 5.2: Das Klassendiagramm des *CrossMin*-Plugins

anderen für Kanten die nach unten führen. Außerdem werden zwei Variablen geführt, die anzeigen, ob der Knoten ein Nullknoten oder ein Dummyknoten ist. Falls sich der Knoten in einem Block befindet, kennt der Knoten diesen, falls nicht, ist diese Variable `null`. Für die Heuristik der Kreuzungsminimierung besitzt jeder Knoten einen Barycenterwert, mehr dazu in Kapitel 5.2.5. Zu den aufgeführten Variablen wurden die üblichen Methoden zum Setzen und Abfragen implementiert.

Edge

Kanten werden im *CrossMin*-Plugin durch die Klasse `Edge` repräsentiert. Da eine Kante immer genau zwei Knoten miteinander verbindet, werden diese beiden Knoten in der Klasse gespeichert und mit Methoden zum Setzen und Abfragen ergänzt. Dabei wird immer zwischen dem unteren und dem oberen Knoten unterschieden, da eine Kante immer zwischen zwei übereinander liegenden Schichten verläuft.

Block

Ein Block ist eine Menge an benachbarten Knoten und kann vom Benutzer manuell erstellt werden. Benachbarte Knoten sind in diesem Fall Knoten, die neben- oder übereinander liegen. Ein Block kann in einem Zug umrandet werden, so dass sich sämtliche Blockknoten innerhalb der Umrandung befinden. Die graphische Darstellung der Blöcke ist in Kapitel 5.2.6 beschrieben. Ein Block wird erzeugt indem der Benutzer eine benachbarte Menge von Knoten markiert und anschließend den Button *Create Block* im Seitenmenü *Plugin* betätigt. Durch markieren des Blocks und anschließendem drücken des Buttons *Remove Block* wird der Block wieder entfernt.

Die Klasse `Block` enthält also in erster Linie eine Liste mit Knoten, die sich in diesem Block befinden, sowie einige Methoden um diese zurückzugeben. Da der Benutzer beliebige Mengen an Knoten auswählen kann, wird in der Methode `isValid` überprüft, ob ein erzeugter Block gültig ist. Ein Block ist gültig, wenn alle Knoten benachbart sind. Dabei wird eine Liste durchlaufen, die zu Beginn nur einen beliebigen Knoten des Blocks enthält. Nun werden nacheinander alle Knoten, die mit diesem benachbart sind und sich im Block befinden, der Liste hinzugefügt. Beinhaltet diese Liste nach dem Ende der Schleife die gleiche Anzahl an Knoten wie der Block, ist der Block gültig.

In der Methode `swap` wird eine sogenannten *ChangeList* erzeugt, mit deren Hilfe der Block um den übergebenen Offset in der übergebenen Lösung versetzt werden kann. Die *ChangeList* besteht aus Objekten der Klasse `NodeIntTuple`, in denen jeweils ein Knoten und eine Nummer gespeichert ist. Die Nummer stellt die neue Position des Knotens dar. Ist die Versetzung des Block nicht möglich, wird eine leere Liste zurück gegeben. Dieser Fall tritt zum Beispiel auf, wenn die neuen Positionen über die Größe des Graphen hinaus gehen. Der Ablauf zur Erstellung der ChangeList sieht wie folgt aus. Es werden alle Knoten des Blocks durchlaufen und dabei wird für jeden Knoten seine neue Position berechnet und diese in die `ChangeList` eingefügt. Dabei wird auch die Positionsänderung des Knotens an der neuen Position, berechnet und eingefügt, denn dieser wird an die ehemalige Position des Blocks verschoben. Stellt sich dabei heraus, dass die Positionierung eines Knotens nicht möglich ist, wird am Ende eine leere Liste zurück gegeben, da die Versetzung des Blocks um den gegebenen Offset nicht möglich ist. Ist der Knoten mit dem die Position getauscht werden muss ebenfalls in einem Block, wird dieser Fall gesondert behandelt. Ist der Block des zu tauschenden Knotens gleich dem, der diese Methode ausführt, ist die Versetzung nicht möglich. Ist der Block jedoch ein Anderer, muss überprüft werden, ob die beiden Blöcke die gleiche Form haben. Ist dies der Fall, können sie ihre Positionen tauschen. Dazu müssen sich alle, zu tauschenden Knoten, im selben Block befinden und die Größe der beiden Blöcke muss identisch sein. Einen weiteren Sonderfall stellt die Verschiebung des Blocks um einen Offset, der kleiner ist als die Breite des Blocks, dar. In diesem Fall kann die Position der Knoten nicht getauscht werden, statt dessen müssen die Knoten

rotiert werden. Die Knoten des Blocks werden dabei um den Offset verschoben, während die Knoten die dabei geändert werden müssen, um die spezifische Breite des Blocks an dieser Stelle verschoben werden.

Zur Berechnung dieser spezifischen Breite steht die Methode `getNumberOfBlockNodes` zur Verfügung. Sie bekommt einen Knoten des Blocks übergeben und berechnet, wie breit der Block an dieser Stelle, in der Anzahl der Knoten gemessen ist. Dazu wird von dem übergebenen Knoten aus, einmal nach rechts und einmal nach links, gelaufen und die Anzahl der Knoten bis zum Ende des Blocks gezählt. Diese Zahl muss nicht mit der Anzahl der Knoten dieses Blocks auf der Schicht übereinstimmen.

Problem

In dieser Klasse wird der Graph aus den Knoten und Kanten konstruiert. Das `Problem` enthält Listen von allen Knoten und Kanten, sowie der Blöcke und stellt diverse Methoden, die diese zurückgeben, bereit. Die Knoten werden in einer Liste mit Objekten vom Typ `NodeList` gespeichert. Eine `NodeList` enthält eine Liste mit Knoten und eine Zahl, der Nummer der Schicht. Eine `NodeList` repräsentiert immer genau eine Schicht.

Zur Beurteilung der Güte wird die aktuelle Anzahl der Kreuzungen gespeichert. Da jedoch nicht nur die Anzahl der Kreuzungen minimiert wird, sondern auch schräge Kanten vermieden werden, gibt es ein weiteren Wert, der den Diagonalitätsgrad eines Graphen misst. Dieser Wert wird berechnet, indem die Diagonalitätswerte aller Kanten summiert werden. Der Diagonalitätswert einer Kante berechnet sich aus dem horizontalen Abstand, gemessen in Knoten, der zwei Knoten, die durch die Kante verbunden sind. Um die Minimierung des Diagonalitätsgrads sinnvoll zu ermöglichen, werden die Schichten beim Einlesen soweit mit sogenannten Nullknoten aufgefüllt, bis alle Schichten die gleiche Anzahl an Knoten enthalten. So ist es möglich sinnvoll schräge Kanten zu minimieren. Der Benutzer hat, wie in Kapitel 4.4.2 beschrieben, die Möglichkeit, die Anzahl der Dummyknoten zu erhöhen und die Breite des Graphen um beliebig viele Nullspalten zu ergänzen. Dies wird in der Methode `updateNumberOfNullNodes` realisiert. Dadurch, dass der Benutzer diesen Wert anpassen kann, kann er so entscheiden, wie wichtig die Minimierung der schrägen Kanten im Gegensatz zur Breite ist. Der Standardwert für die Anzahl der zusätzlichen Spalten beträgt 10% der Anzahl der Spalten im ursprünglichen Graphen.

Solution

Die Klasse `Solution` des *CrossMin*-Plugins beinhaltet im Wesentlichen das aktuelle Problem, indem durch die Anordnung der Knoten schon die Lösung enthalten ist. Neben den Methoden, die die Informationen für die Informationsanzeigen zurückgeben, muss die Methode `getValue` überschrieben werden. Hier wird die Güte der Lösung berechnet, mit der der Optimierungsalgorithmus arbeitet. Da im *CrossMin*-Plugin nicht nur die Kreuzungen

minimiert werden, reicht es nicht, nur deren Anzahl zurückzugeben. Um eine Minimierung der schrägen Kanten zu erreichen, müssen die beiden Werte, Kreuzungsanzahl und Diagonalitätsgrad zu einem Wert zusammengefasst werden. Um der größeren Wichtigkeit der Kreuzungsminimierung Ausdruck zu verleihen, habe ich mich dafür entschieden, dass die Kreuzungsanzahl die Ziffern vor dem Komma darstellen und der Diagonalitätsgrad die Ziffern hinter dem Komma. So ist gewährleistet, dass eine Lösung mit einer geringeren Anzahl an Kreuzungen in jedem Falle besser ist, als eine Lösung, die mehr Kreuzungen, aber einen kleineren Diagonalitätsgrad besitzt.

Eine weitere interessante Methode, die überschrieben werden muss, ist die Methode `getLowerBoundOfNumberOfPossibleMoves`. Diese gibt eine untere Schranke der möglichen Moves zurück und dient wie schon erwähnt dazu, dass die Tabuliste nicht so groß wird, dass keinerlei Moves mehr möglich sind. In einem normalen k-Schichten-Graph ist es einfach, die Anzahl der möglichen Knotenpaare, die getauscht werden können und damit die Anzahl der möglichen Moves, zu berechnen. In unserem Fall, kann der Graph allerdings Nullknoten enthalten, die die Anzahl der Moves einschränken, da Moves die zwei Nullknoten tauschen nicht zugelassen sind. Weiterhin schränken Blöcke die Anzahl der möglichen Moves weiter ein. Um eine untere Schranke für die Anzahl der möglichen Moves zu berechnen, werden nun sowohl Nullknoten als auch Knoten, die sich in einem Block befinden, als fest betrachtet.

CrossCounting

In der Klasse `CrossCounting` werden in zwei statischen Methoden die Kreuzungen und der Diagonalitäsgrad bestimmt. Die Laufzeit dieser beiden Methoden sollte gering gehalten werden, da sie vor jedem einzelnen Move aufgerufen werden. Für das Kreuzungszählen wird die Methode des lexikographischen Sortierens zum Zählen von Kreuzungen zwischen zwei Schichten genutzt. Bei dieser Methode werden die Knoten gesplittet, so dass jeder Knoten von genau einer Kante mit einem anderen Knoten verbunden wird. Die Anzahl der Kreuzungen wird dadurch nicht verändert. Die Knoten der oberen Schicht werden ihrer Reihenfolge nach nummeriert und die Knoten in der Schicht darunter erhalten die Nummer des, durch die Kante verbundenen, oberen Knoten. Nun wird die Zahlenfolge der unteren Knoten sortiert. Die Anzahl der dazu nötigen Inversionen, entspricht der Anzahl der Kreuzungen im Ursprungsgraphen. Da die Sortierung mit *InsertionSort* realisiert wird, erreicht das Kreuzungszählen eine theoretische Laufzeit von $O(|E| + |C|)$. Diese Laufzeit kann durch Sortieren mit Hilfe eines *Accumulator Trees* noch auf $O(|E|log|V_small|)$ reduziert werden. Da die Instanzen, die wir mit HuGO berechnen, nicht allzu groß werden, lohnt sich der Aufbau dieser Datenstruktur nicht. Die Laufzeit würde in der Praxis wahrscheinlich über der Laufzeit mit *InsertionSort* liegen. Die Methode `crossCount` zählt so die Kreuzungen jeder Schicht und gibt die Summe zurück. [2]

Zur Bestimmung des Diagonalitäsgrads werden in der Methode `countDiagonalEdges` alle Kanten durchlaufen. Für jede Kante wird der horizontale Abstand der Positionen, der Knoten, die durch die Kante verbunden werden, aufsummiert. So lässt sich der Diagonalitäsgrad in Laufzeit $O(|E|)$ bestimmen.

5.2.2 Moves

In diesem Abschnitt werden die zwei für das *CrossMin*-Plugin entwickelten Moves, `SwapMove` und `BlockMove` vorgestellt. Anschließend wird die Generierung der Moves erläutert.

SwapMove

Ein `SwapMove` tauscht die Positionen zweier Knoten, die auf der selben Schicht liegen, aus. Die Mobility des Moves ist das arithmetische Mittel der Mobilities der zwei Knoten. Ausnahme bilden Knoten mit niedriger Mobility. Sobald ein Knoten niedrige Mobility hat, wird auch dem Move ein niedriger Mobilitywert zugewiesen, unabhängig davon, welche Mobility der zweite Knoten hat. In der Methode `createReverseMove` wird ein Move erzeugt, der die Änderung des aktuellen Moves rückgängig macht. Dieser wird in die Tabuliste eingefügt. Die Umkehrung des `SwapMoves` ist einfach durch Vertauschung der beiden Knoten zu erreichen.

BlockMove

In Kapitel 5.2.1 wurden die sogenannten Blöcke vorgestellt. Sie fixieren die Struktur der Knoten innerhalb des Block, können aber im Ganzen verschoben werden. Diese Verschiebungen sind, mit den eben beschriebenen `SwapMoves`, nicht zu realisieren. Deshalb wurde die Klasse `BlockMove` entwickelt, die für Verschiebungen von Blöcken zuständig ist. Ein `BlockMove` verschiebt den übergebenen Block um den übergebenen Offset. Die eigentliche Verschiebung, die in der Methode `doMove` stattfindet, ist in der Methode `shift` in der Klasse `Block` implementiert und wurde in Kapitel 5.2.1 erläutert. Die Mobility eines `BlockMoves` wird, wie schon beim `SwapMove`, durch das arithmetische Mittel über die Mobilities der Knoten berechnet. Ein Knoten mit niedriger Mobility führt wieder dazu, dass der Move eine niedrige Mobility von 1 bekommt. Die Umkehrung des Moves in der Methode `createReverseMove` geschieht durch Umkehrung des Vorzeichens des Offsets.

MoveGenerator

Die Hauptaufgabe der Klasse `MoveGenerator` ist das Generieren von Moves für den Optimierungsalgorithmus. Für diese Funktion muss die Methode `generateNextMove` der Oberklasse überschrieben werden. Ein `SwapMove` wird erzeugt indem zufällig zwei Knoten, die auf einer Schicht liegen, ausgewählt werden. Befindet sich einer dieser beiden Knoten in

einem Block, wird die Methode `generateBlockMove` aufgerufen, die falls möglich einen `BlockMove` erzeugt. Falls es nicht möglich ist, einen `BlockMove` zu erzeugen oder falls die Methode `checkIfMoveIsUseful` den erzeugten `SwapMove` als nicht nützlich klassifiziert, werden zufällig zwei neue Knoten ausgewählt. Dies geschieht solange, bis ein Move erzeugt ist, der zurückgegeben werden kann.

Betrachten wir nun die Methode `generateBlockMove`, die den oben erzeugten ungültigen `SwapMove` übergeben bekommt. Zuerst einmal werden zwei Fälle unterschieden. Im ersten Fall befinden sich beide Knoten, die getauscht werden sollen, in einem Block. Ist dies der Fall, wird mit Hilfe der Methode `checkIfIsomorphic` überprüft, ob die beiden Blocks die gleich Form haben und getauscht werden können. Befinden sich beide Knoten im gleichen Block wird kein `BlockMove` erzeugt. Im zweiten Fall befindet sich nur einer der beiden Knoten in einem Block. Nun wird mit der Methode `shift` der Klasse `Block` überprüft, ob der gesamte Block um den Abstand der beiden Knoten im übergebenen `SwapMove` verschoben werden kann. Die oben erwähnte Methode `checkIfIsomorphic` bekommt zwei Blöcke übergeben und ermittelt, ob diese beiden Blöcke die gleich Form haben und auf den selben Schichten liegen, so dass ein Tausch der Positionen möglich ist. Um den Abstand der zwei Blöcke zu bestimmen, wird erst in jedem der zwei Blöcke der Knoten in der unteren rechten Ecke gesucht. Nun werden alle Knoten des ersten Blocks durchlaufen. Für jeden Knoten wird, seine Position verschoben um den berechneten Abstand, überprüft. Befindet sich an der berechneten Position kein Knoten der im zweiten Block liegt, sind die zwei Blöcke nicht isomorph. Findet sich an jeder berechneten Position ein Knoten der im zweiten Block liegt und ist weiterhin die Größe, also die Anzahl der Knoten im Block, gleich, sind die beiden Blöcke isomorph. Bei der zufälligen Auswahl der Knoten zur Erzeugung des `SwapMoves` können auch Nullknoten oder Knoten, die keinerlei Kanten besitzen, ausgewählt werden. Sind beide Knoten des Moves kantenlos oder ein Nullknoten, führt die Vertauschung der Positionen zu keiner sinnvollen neuen Lösung. Deshalb werden Moves dieser Art durch die Methode `checkIfMoveIsUseful` aussortiert.

Um die in Kapitel 3.2.7 beschriebene Funktion der lokalen Optimierung ausführen zu können, müssen zwei weitere Methoden implementiert werden, die eine Nachbarschaftsdefinition voraussetzen. Im Graphen lassen sich Nachbarschaften intuitiv definieren. Knoten, die sich auf einer Schicht nebeneinander befinden, sind benachbart. Außerdem sind zwei Knoten benachbart, die durch eine Kante miteinander verbunden sind. Die Methode `getNeighborhood` liefert also eine Liste aller Knoten, die mit dem übergebenen Knoten benachbart sind. In der Methode `getNeighborhoodMoves` werden zu einem gegebenen Knoten alle Moves, die die Nachbarschaft betreffen, zurück gegeben. In unserem Graphen werden dabei die Moves erzeugt, die den übergebenen Knoten mit seinen Nachbarn auf der selben Schicht tauschen. Blöcke werden bei der lokalen Optimierung nicht verschoben.

5.2.3 Score

In der Klasse `Score` müssen zwei Methoden, die die Güte von Lösungen vergleichen, überschrieben werden. Wenn die Methode `getValue` der Klasse `Solution` die Güte genau repräsentiert, kann dieser Wert hier verglichen werden. Die Methoden müssen für den Fall implementiert werden, dass es in einem Plugin nicht erwünscht ist, den Wert der Güte genau zu repräsentieren. Im *CrossMin*-Plugin wird der Wert, den die Methode `getValue` liefert, verglichen, denn dieser repräsentiert die Güte einer Lösung vollständig, wie oben schon beschrieben wurde.

5.2.4 ProblemParser

Da *HuGO* kein Editor für die Probleminstanzen enthält, müssen diese eingelesen werden. Dazu wird, wie in Kapitel 3.4 erläutert, ein String aus einer Textdatei eingelesen. Aus diesem wird in der Klasse `ProblemParser` in der Methode `readSolution` eine Probleminstanz und die erste Lösung, die angezeigt werden kann, erzeugt. Im *CrossMin*-Plugin ist der String, in dem alle Informationen für eine Probleminstanz befinden, wie folgt codiert:
Der String
$V1 = (id_{1,1}, n_{1,1}; ...; id_{1,k_1}, n_{1,k_1};)$
...
$Vl = (id_{l,1}, n_{l,1}; ...; id_{l,k_l}, n_{l,k_l};)$
$E = ([id, id], [id, id],)$
Die Decodierung wird im Folgenden erläutert. Es wird ein Graph mit l Schichten mit je k_l Knoten erzeugt. Der Knoten auf Schicht i an Position j hat die Identifikationsnummer $id_{i,j}$ und den Namen $n_{i,j}$. Die Kante e ist definiert durch die Identifikationsnummern ihrer beiden Knoten, die sich auf übereinander liegenden Schichten befinden müssen. Ein Dummyknoten kann erzeugt werden, indem einem Knoten der Name 'dummy' gegeben wird. Kommasetzung und Leerzeichen müssen bei der Codierung beachtet werden.

In der Methode `readSolution` wird ein solcher String nun zu einer Probleminstanz decodiert. Dazu werden zuerst die einzelnen Knoten der Reihe nach erzeugt und einem neuen Objekt der Klasse `Problem` hinzugefügt. Anschließend werden die Kanten erzeugt und dabei auf die schon vorhandenen Knoten im Problem zurückgegriffen. Nun werden die einzelnen Zeilen mit Nullknoten aufgefüllt, bis alle Schichten die selbe Länge besitzen. Abhängig von einer Variable in der Klasse `crossMin.properties` werden noch weitere Spalten mit Nullknoten hinzugefügt. Standardmäßig entspricht die Anzahl der zusätzlichen Nullspalten 10% der Anzahl Knoten in der größten Schicht. Da die Positionen der Knoten in den Schichten nach ihrer Reihenfolge festgelegt werden, ist es nicht weiter nötig eine Startlösung zu berechnen. Das so erzeugte Objekt der Klasse `Problem` kann einer erzeugten `Solution` übergeben werden.

5.2.5 Heuristik

Als Heuristik für das *CrossMin*-Plugin habe ich mich für die Barycenter Heuristik entschieden. Diese liefert schnell eine Lösung, die für den Benutzer wesentlich übersichtlicher ist, als eine zufällige Anordnung. Die Knoten werden dabei, eine Schicht nach der anderen, nach ihrem Barycenterwert, der für jeden Knoten berechnet wird, sortiert. Der Barycenterwert eines Knoten entspricht dabei der durchschnittliche Position der Nachbarn auf der oberen Schicht. [14]

5.2.6 Graphische Darstellung

In diesem Abschnitt werden alle Klassen, die zur graphischen Darstellung im *CrossMin*-Plugin implementiert werden müssen, vorgestellt.

PluginUI

In der Klasse `PluginUI` werden die Komponenten der graphischen Oberfläche, die pluginspezifisch sind, erzeugt und zurückgegeben, um in den pluginübergreifenden Teil der Oberfläche eingebaut zu werden. Eine dieser Komponenten ist das Panel, welches die Informationsanzeige oberhalb der Zeichenfläche darstellt. Dieses wird in der Methode `createPluginInfoPanel` erzeugt und besteht im *CrossMin*-Plugin, wie schon im *BackPack*-Plugin, im Wesentlichen aus zwei Labeln. Das obere Label zeigt die Kreuzungsanzahl und den Diagonalitätsgrad, der besten bisher gefundenen Lösung an, während das untere Label diese Informationen über die aktuelle Lösung anzeigt. Um die Informationsanzeige aktuell zu halten, muss die Methode `updateInfoPanel` überschrieben werden, deren Aufruf zu einem erneuten Abfragen der Informationen führen sollte. Die Methode `updateInfoPanel` gibt es in zwei Varianten, denn die Informationen über die beste Lösung sind ausschließlich im Verlauf gespeichert, auf den die Klasse `PluginUI` keinen Zugriff hat. In einer Variante wird dieser Methode ein String mit allen Informationen über die beste Lösung übergeben.

Kommen wir nun zur nächsten pluginspezifischen Komponente, dem Panel für das Seitenmenü. Es stellt Buttons zum Erstellen und Entfernen von Blöcken zur Verfügung und bietet durch ein Textfeld mit einem Bestätigungsbutton, die Funktion zum Ändern der Anzahl der zusätzlichen Spalten, gefüllt mit Nullknoten. In der Methode `createPlugin-TaskPanel` wird dieses Panel erzeugt und ein Layout erstellt.

Die dritte pluginspezifische Komponente ist die Zeichenfläche. Dazu wird in der Methode `createEditPanel` ein `EditPanel` erzeugt, dem ein `MouseListener` hinzugefügt wird, der die entsprechenden Methoden des `EditPanels` aufruft. Außerdem werden, sowohl dem `EditPanel` als auch dem

`PluginTaskPanel`, ein `KeyListener` hinzugefügt. Dieser realisiert, dass Auswählen mehrerer Knoten durch Drücken der STRG-Taste. Außerdem implementiert er die Eingabe

für das Textfeld. Weiterhin wird in der `PluginUI` ein `ActionListener` als lokale Klasse implementiert, um die Funktion der Buttons des Panels zu realisieren.

EditPanel

Die Zeichenfläche des *CrossMin*-Plugins wird in der Klasse `EditPanel` in der Methode `paintComponent` gezeichnet. Dazu werden die `paintMe`-Methoden aller Knoten, Kanten und Blöcke aufgerufen. Um die, in Kapitel 4.4.2 beschriebenen, Möglichkeiten, der Verschiebung von Knoten und Blöcken durch den Benutzer, zu realisieren, werden hier drei Methoden zur Reaktion auf `MouseEvents` implementiert. Diese werden, wie im letzten Abschnitt beschrieben, in den entsprechenden Methoden im `MouseListener` des EditPanels ausgeführt. Im Falle eines Mausklicks werden alle Knoten in der Methode `organizeMouseClick` durchlaufen und überprüft, ob einer getroffen wurde. Ist ein Knoten getroffen, wird er abhängig davon, ob die STRG-Taste gedrückt ist oder nicht, entweder in die Liste der ausgewählten Knoten hinzu gefügt oder aber die Liste wird geleert und er wird als einziger eingefügt. Wurde kein Knoten getroffen, wird die Liste der ausgewählten Knoten geleert. Das Setzen von Mobilities und das Fixieren der ausgewählten Knoten findet automatisch, durch das Hinzufügen in die Liste der ausgewählten Knoten, in der Klasse `HuGOUI` statt. Anschließend werden alle Blöcke betrachtet und ebenfalls darauf getestet, ob sie getroffen wurden. Ein getroffener Block wird in den entsprechenden Zustand versetzt und lokal gespeichert, um diesen Zustand beim Abwählen wieder deaktivieren zu können. Weiterhin wird die Mobility und die Fixierung des Blocks geändert, wenn die entsprechende Funktion vom Benutzer ausgewählt wurde.

Wurde die Maus gedrückt, findet in der Methode `organizeMousePress` ein ähnlicher Ablauf statt. Es werden alle Knoten und alle Blöcke betrachtet und überprüft, ob einer getroffen wurde. Ist dies der Fall, wird die Komponente lokal gespeichert, um auf sie zugreifen zu können, falls sie beim Lösen des Mausdrucks verschoben wird. Falls ein Block getroffen ist, wird zusätzlich der Knoten gespeichert, der der Mausposition im Block am nächsten liegt. Dieser Knoten wird von der Methode `isBlockHit` des Blocks zurückgegeben und dient dazu, beim Lösen des Mausdrucks, die Verschiebung des Blocks einfach und genau bestimmen zu können.

Kommen wir nun zu der Methode `organizeMouseRelease` die ausgeführt wird, wenn der Mausdruck gelöst wird. Hier findet die eigentliche manuelle Verschiebung von Knoten und Blöcken statt. Zuerst einmal wird also überprüft, ob ein Knoten oder ein Block beim Mausdruck gespeichert wurde, falls nicht, findet keinerlei Reaktion statt. Wurde ein Knoten oder Block zwischengespeichert, werden alle Knoten durchlaufen und überprüft, ob einer von ihnen vom `MouseEvent`, während des Lösens des Mausdrucks, getroffen wurde. Ist dies der Fall, findet, falls möglich, eine Verschiebung der zwischengespeicherten Komponente an diese Position statt. Ist die zwischengespeicherte Komponente ein Block, wird

die Methode `swap` des Blocks mit dem berechneten Abstand der Verschiebung aufgerufen. Ist eine Verschiebung an die neue Position nicht möglich, liefert diese Methode eine leere Liste zurück, in der keinerlei Positionsveränderungen gespeichert sind. Nun betrachten wir den Fall, dass die zwischengespeicherte Komponente ein Knoten ist. Befindet sich an der neuen Position des Knotens, ein Block, wird die Methode `swap` dieses Blocks mit der berechneten Verschiebung aufgerufen. Kommen wir jetzt zu dem Fall, dass an beiden Positionen kein Block vorhanden ist. Dann wird der `pressedNode` an seine neue Position geschoben und anschließend werden alle Knoten an der alten und der neuen Position um eins in die Richtung der alten Position geschoben. Befinden sich zwischen der alten und der neuen Position des Knotens Blöcke, werden diese übersprungen und nicht verschoben.

JNode

Die Klasse `JNode` implementiert im Wesentlichen die zwei Methoden `paintMe` und `isMouseHit`. Ein Knoten wird als Kreis dargestellt, dessen Radius in der Klasse `crossMin.properties` eingestellt werden kann. Abhängig von den derzeitigen Zuständen, in denen sich ein Knoten befinden kann, wird er farblich gefüllt oder umrandet. Ein Knoten wird grundsätzlich in einer der Mobility entsprechenden Farbe gefärbt. Wenn ein Knoten gerade vom Benutzer mit der Maus angeklickt ist, wird er blau gefärbt. Knoten die vom Benutzer ausgewählt wurden, werden blau umrandet. Diese Farben sind in Variablen in der Klasse `crossMin.properties` gespeichert und können so leicht geändert werden. Fixierte Knoten werden durch einen kleinen grauen Punkt erkenntlich gemacht. Die sogenannten Nullknoten werden weiß gefärbt, damit sie möglichst wenig auffallen, denn sie stellen freie Plätze dar. Um sie verschieben zu können, müssen sie allerdings eingezeichnet werden. Dummyknoten sind kleiner als normale Knoten. Abhängig von der entsprechenden Einstellung werden die Namen der Knoten gezeichnet.

Die Methode `isMouseHit` testet, ob das übergebene `MouseEvent` den Knoten getroffen hat. Jede Position beim Zeichnen und auch bei den Abfragen wird mit dem Vergrößerungsfaktor multipliziert, der in `properties` gespeichert ist. Dadurch wird die in Kapitel 4.1.1 beschriebene Vergrößerungsfunktion realisiert.

JEdge

In der Klasse `JEdge` wird in der Methode `paintMe` eine Linie, die die Kante repräsentiert, zwischen den beiden Knoten gezeichnet. Auch hier werden sämtliche Positionsangaben mit dem Zoomfaktor multipliziert.

JBlock

Die Klasse `JBlock` implementiert die Methoden `paintMe` und `isBlockHit`. Ein Block wird auf der graphischen Oberfläche dargestellt, indem die Menge der Knoten, die in einem

Block liegen, umrandet und grau hinterlegt wird. Ein ausgewählter Block wird, genauso wie ein ausgewählter Knoten, durch einen blauen Rand markiert. Ein aktuell angeklickter Block wird komplett blau gefärbt. In der Methode `isBlockHit` wird überprüft, ob der Block von dem übergebenen `MouseEvent` getroffen wurde, außerdem wird der Knoten der dem `MouseEvent` am nächsten liegt zurückgegeben.

Kapitel 6

Fazit

In dieser Diplomarbeit wurde das Thema der *Human Guided Optimization* betrachtet. Es wurden zwei Programme, *Foldit* und *HuGS*, die dieses Konzept realisieren, genauer untersucht. Daraufhin wurde ein neues Programm mit dem Namen *HuGO* entwickelt. Es stellt ein Framework dar, das diverse Funktionen zur benutzergesteuerten Optimierung bietet. Durch seinen modularen Aufbau können Plugins für beliebige Optimierungsprobleme integriert werden. Der Aufbau und auch viele der Funktionen sind dem Programm *HuGS* nachempfunden, zusätzlich wurden einige neue Ideen und Erweiterungen entwickelt und integriert.

Als Suchalgorithmus wurde ein Simulated Annealing Algorithmus mit integriertem Tabuansatz realisiert. Durch diesen Algorithmus bieten sich für den Benutzer neue Möglichkeiten der Steuerung durch die Temperatur des Simulated Annealing Algorithmus an. Zur Unterstützung der Optimierung wurde die Funktion der lokalen Optimierung integriert, die nach manuellen Änderungen durch den Benutzer, einen kurzen Algorithmus auf den veränderten Knoten auf der Suche nach lokalen Verbesserungen ausführt. Außerdem wurde die Idee entwickelt eine schnelle Heuristik für jedes Plugin zu implementieren, um geeignete Startlösungen zu berechnen. Für die zwei entwickelten Plugins wurden bereits Heuristiken implementiert.

Das erste Plugin optimiert das Rucksackproblem und ist als kleines Beispiel implementiert worden. Das zweite Plugin namens *CrossMin* optimiert die k-Schichten-Kreuzungsminimierung. Im *CrossMin*-Plugin wurden einige weiterführende Ideen realisiert. Es minimiert nicht nur die Anzahl der Kreuzungen, sondern vermeidet auch schräge Kanten. Da bei der visuellen Darstellung des k-Schichten-Graphen, zwangsweise ein Layout erstellt wird und der Benutzer sich mit diesem beschäftigt, bietet es sich an, eben dieses Layout zu optimieren. Weiterhin können im Graphen Blöcke erstellt werden, die die Strukturen der beinhaltenden Knoten erhalten. Dies erweitert das *CrossMin*-Plugin um eine neue interessante Möglichkeit die Optimierung zu steuern.

Abschließend lässt sich sagen, dass ein voll funktionsfähiges Framework, basierend auf der Idee der *Human Guided Optimization*, entwickelt wurde. Durch den modularen Aufbau ist es nicht nur möglich weitere Plugins einfach zu integrieren, sondern auch neue Suchalgorithmen oder andere neue Funktionen. Es wurde eine Grundstruktur entwickelt auf der weiter aufgebaut werden kann.

Anhang A

Inhaltsverzeichnis der CD

- Diplomarbeit
- HuGO
  - Beispielinstanzen
  - HuGO.jar
  - Java Dokumentation
  - Quellcode
    - algo
    - backPack
    - crossMin
    - generalPlugin
    - images
    - io
    - main
    - org
    - template
    - ui
    - util

Abbildungsverzeichnis

Literaturverzeichnis

[1] Anderson, David, Emily Anderson, Neal Lesh, Joe Marks, Brian Mirtich, David Ratajczak und Kathy Ryall: *Human-guided Simple Search.* Proc. of AAAI, Seiten 41–47, 2000.

[2] Barth, Wilhelm, Petra Mutzel und Michael Jünger: *Simple and Efficient Bilayer Cross Counting.* JGAA, 8(2):179–194, 2004.

[3] Beier, Rene und Berthold Vöcking: *15. Algorithmus der Woche: Das Rucksackproblem.* MPI Saarbrücken, 2006.

[4] Chien, Steve, Gregg Rabideau, Jason Willis und Tobias Mann: *Automating Planning and Schedulinogf Shuttle Payload Operations.* J. Artificial Intelligence, 1999.

[5] Eglese, R. W.: *Simulated annealing: A tool for operational research.* European Journal of Operational Research, 46(3):271–281, June 1990.

[6] Günther, Wolfgang, Robby Schönfeld, Bernd Becker und Paul Molitor: *k-Layer Straightline Crossing Minimization by Speeding Up Sifting.* LNCS 1984, Seiten 253–258, 2001.

[7] Grötschel, Martin, Michael Jünger und Gerhard Reinelt: *A Cutting Plane Algorithm for the Linear Ordering Problem.* Operations Research, 32(6):1195–1220, 1984.

[8] Hajek, Bruce: *Cooling Schedules for Optimal Annealing.* Mathematics of Operations Research, 13(2):311–329, May 1988.

[9] Klau, Gunar W., Neal Lesh, Joe Marks und Michael Mitzenmacher: *Human-Guided Tabu Search.* Proc. of AAAI, Seiten 209–216, June 2002.

[10] Klau, Gunar W., Neal Lesh, Joe Marks, Michael Mitzenmacher und Guy T. Schafer: *The HuGS Platform: A Toolkit for Interactive Optimization.* Technischer Bericht TR2002-08, MITSUBISHI ELECTRIC RESEARCH LABORATORIES, 2002.

[11] Klau, Gunnar W., Neal Lesh, Joe Marks und Michael Mitzenmacher: *Human-guided search.* Journal of Heuristics, 2009.

[12] Laboratories, Mitsubishi Electric Research: *A Complete and Effective Move Set for Simplified Protein Folding*, 2007. http://www.merl.com/projects/proteinfolding/.

[13] Martello, Silvano und Paolo Toth: *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley and Sons, 1990.

[14] Mutzel, Petra: *Zeichnen gerichteter Graphen*, 2005. http://ls11-www.cs.uni-dortmund.de/people/gutweng/AE-07/schichten.pdf.

[15] Mutzel, Petra und Michael Jünger: *2-Layer Straightline Crossing Minimization: Performance of Exact and Heuristic Algorithms.* JGAA, 1(1):1–25, 1997.

[16] Ryall, Kathy: *GLIDE.* LNCS, Seiten 479–480, 2002.

[17] University of Washington: *Rosetta@home.* http://boinc.bakerlab.org/rosetta/.

[18] University of Washington, Departments of Computer Science Engineering and Biochemistry: *Foldit.* http://fold.it/portal/.

[19] Wegener, Ingo: *Wie Simulated Annealing den Metropolis Algorithmus schlägt*, 2006. http://www.thi.informatik.uni-frankfurt.de/ weinard/Byothers/vortragwegener.pdf.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 12. April 2010

Sophia Kardung