TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

Diplomarbeit

Interaktive Reihenfolgeplanung für die Automobilindustrie

Ausgeführt am

Institut für Computergraphik und Algorithmen
der Technischen Universität Wien

Unter der Anleitung von

Univ.-Prof. Petra Mutzel
und
Dr. Gunnar W. Klau

durch

Bin Hu
Obere Donaustraße 43/3/54e
1020 Wien

März 2004 _____

Danksagung
An dieser Stelle gilt mein besonderer Dank Dr. Gunnar W. Klau für die hervorragende Betreuung, Unterstützung und Motivation während der Anfertigung dieser Diplomarbeit und für die Einführung in das interessante Thema. Weiters möchte ich allen Mitgliedern der Abteilung für Algorithmen und Datenstrukturen des Instituts für Computergraphik und Algorithmen an der Technischen Universität Wien für die stete Hilfsbereitschaft und die äußerst angenehme Arbeitsatmosphäre danken, darunter insbesondere Abteilungsleiterin Prof. Dr. Petra Mutzel und Dr. René Weiskircher für ihre Hilfsbereitschaft und für die Diskussionsbereitschaft, wenn knifflige Probleme auftauchten.
Besonderer Dank auch an Dr. Van-Dat Cung, der unsere Teilnahme an der „ROADEF Challenge 2005“ ermöglichte und für Fragen bezüglich der Aufgabenstellung immer offen stand. Dank gilt auch Robert Nickel für die Hilfestellung zu diesem Thema. Ebenfalls bedanke ich mich bei Markus Chimani für die technische und inhaltliche Unterstützung bei der Verfassung dieser Arbeit. Ich danke besonders Dr. Elisabeth Langer für das Korrekturlesen der Arbeit.
Nicht zuletzt gilt mein herzlicher Dank meiner Familie für die Unterstützung meines Studiums.

Inhaltsverzeichnis

1 Einleitung

Das vorliegende Programm wurde in erster Linie für die „ROADEF Challenge 2005“ gefertigt. „ROADEF Challenge 2005“ ist ein Wettbewerb, der von der „Französischen Gesellschaft für Operation Research und Entscheidugsanalyse“ in Zusammenarbeit mit Industriepartnern organisiert wird. Aufgabenstellung war das folgende Reihenfolgeplanungsproblem aus der Automobilindustrie.
Die Planung der Autoherstellerfirma Renault soll mit Hilfe eines Optimierungsalgorithmus verbessert werden. Hierbei geht es darum, die Reihenfolge einer großen Anzahl an Autos, die täglich über das Fließband laufen, zu optimieren, sodass die Zusammenarbeit in den einzelnen Phasen so reibungslos wie möglich funktioniert.
Es gibt drei Phasen: Karosseriefertigung, Lackierstation und Fließband (siehe Abbildung 1).

Abbildung 1: Aufbau (Quelle: [7])

- Karosseriefertigung: Die Karosserie für das Fahrzeug wird hier gepresst, montiert und verschweißt. Diese Phase wird jedoch nicht in der Aufgabe berücksichtigt.
- Lackierstation: Jedes Fahrzeug besitzt eine Farbe. Wenn zwei aufeinanderfolgende Fahrzeuge nicht dieselbe Farbe aufweisen, dann muss der Farbsprüher zwischendurch gereinigt werden, was zusätzliche Kosten verursacht. Es ist jedoch zu beachten, dass in jedem Fall nach n Fahrzeugen der Farbsprüher gereinigt werden muss, egal ob das nächste Fahrzeug in der Folge dieselbe Farbe besitzt oder nicht. Zusätzlich ist noch verlangt, dass nach dieser Reinigung ein Fahrzeug einer an-

deren Farbe anlaufen muss. Es ist verboten, eine Lösung mit mehr als n aufeinanderfolgenden Fahrzeugen der selben Farbe zu generieren.

- Fließband: Jedes Fahrzeug verfügt über eine Anzahl an zusätzlichen, optionalen Komponenten (z.B. Klimaanlage, Dachfenster, Heckspoiler, etc.)

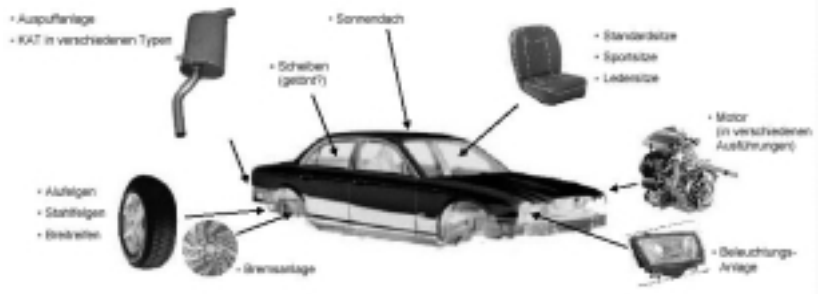


Abbildung 2: Fließband Komponenten (Quelle: [7])

Diese werden in der Fließband-Phase eingefügt. Da das Montieren jeder Komponente eine gewisse Zeit in Anspruch nimmt, sollten innerhalb eines gewissen Zeitfensters nicht zu viele Fahrzeuge mit der selben Komponente anlaufen. Spezifiziert wird immer ein Bruch der Form $\frac{Z}{N}$, was soviel bedeutet wie „nicht mehr als Z Fahrzeuge innerhalb von N aufeinanderfolgenden Fahrzeugen sollten die Komponente benötigen“. Diese Constraints sind jedoch weiche Nebenbedingungen. Ihre Verletzungen sind erlaubt, sollten aber minimiert werden. Pro Verletzung wird je nach Spezifikation ein Strafwert zum Ergebnis dazu addiert. Dieser Strafwert hängt auch davon ab, ob die verletzte Nebenbedingung eine hohe oder niedrige Priorität besitzt. Die Priorität sagt aus, wie wichtig die Einhaltung einer Nebenbedingung ist.

Als Lösungsansatz dient das interaktive, heuristische Optimierungssystem HuGS (Human Guided Search). Hauptziel der Arbeit war die Einbettung eines Lösungsansatzes für das Problem in das HuGS-Framework. HuGS ist ein Framework mit heuristischen Suchverfahren, die auf eine Vielzahl an Problemen angewandt werden können. Das besondere an HuGS ist die menschliche Interaktivität währen der Optimierungsphase.
Herkömmliche Optimierungsalgorithmen und Optimierungsprogramme sind

im Bereich der Interaktivität sehr bescheiden und erlauben höchstens einige wenige Parameterumstellungen vor dem Start. Dabei wird jedoch vernachlässigt, dass der Mensch auch seinen Teil zur Lösungssuche beitragen kann. Bei einem komplexen Problem, das visuell gut repräsentiert ist, kann der Mensch durch Intuition und Erfahrung oft Verbesserungen erkennen, die der Algorithmus möglicherweise erst nach sehr langer schematischer Suche finden kann. Genau hier greift HuGS ein und bietet dem Benutzer die Möglichkeit, den Optimierungsprozess visuell mitzuverfolgen und gegebenenfalls interaktiv mitzuwirken.
Das HuGS Projekt wurde von Mitsubishi Electric Research Laboratories (MERL) in Cambridge, MA, gegründet. Die Hauptmitwirkenden dabei sind Neal Lesh von MERL, Michael Mitzenmacher von der Harvard Universität in Cambridge und Gunnar W. Klau von der Technischen Universität Wien. Im HuGS System kann der Benutzer manuell die Lösung modifizieren, zu einer früheren Lösung zurückgehen und verschiedene Suchalgorithmen aufrufen und anpassen. Noch wichtiger ist die Möglichkeit, durch die Änderung der sogenannten Mobilities an den Suchobjekten den Suchraum einzuschränken und zu verändern.

Aufbau der Arbeit
Im Kapitel „Aufgabenstellung“ wird näher auf die Zählung der Farbwechsel und Constraintsverletzungen eingegangen. Die Reihung der Optimierungsprioritäten ergibt die Zielfunktion. Außerdem wird beschrieben, wie eine Probleminstanz aufgebaut ist.

Das Kapitel „Verwandte Forschungsgebiete“ stellt einige Papers vor, die ähnliche Probleme behaldeln.

Das Kapitel „Human Guided Search (HuGS)“ stellt das Optimierungssystem vor, in dem das vorliegende Problem integriert und damit gelöst wird. Dabei werden Terminologien des Systems und seine Funktionsweise im Detail beschrieben. Zuletzt zeigen einige Beispiele anderer Applikationen, die ebenfalls das HuGS System benutzen, wie sie ihr Problem integriert haben.

Im Kapitel „Die Paintshop Applikation“ geht es um das Hauptthema der Arbeit. Die Funktionsweise der Applikation wird geschildert, indem die einzelnen Schnittstellen zu HuGS betrachtet werden. Ebenfalls wichtig ist der Ansatz, wie der Suchraum zugunsten der Performance verkleinert wird. Am Ende wird ein Script Modus beschrieben, der die menschliche Interaktion von HuGS simulieren kann.

Danach folgen im Kapitel „Resultate“ die mit diesem Programm erreichten Werte.

Das Kapitel „Exakter Lösungsansatz“ führt Überlegungen zu reduzierten Beispielen vor, die exakt zu lösen sind. Dann erfolgt Schritt für Schritt die Formulierung des Problems zu einem ganzzahligen linearen Programm (ILP), das mit Hilfe von Programmen, die auf LPs spezialisiert sind, exakt gelöst werden kann.

Das Kapitel „Komplexität“ analysiert, wie schwer das Problem wirklich ist, und beweist, dass es NP schwer ist, indem eine Reduktion auf ein bekanntes Problem ausgeführt wird.

Das Kapitel „Zusammenfassung und Ausblick“ fasst die Ergebnisse nochmals zusammen und schildert die offenen Möglichkeiten, wie das Programm verbessert werden könnte.

Das Kapitel „Anhang“ beinhaltet eine Beschreibung, wie die Programmoberfläche zu bedienen ist.

2 Aufgabenstellung

Im folgenden Kapitel wird die Aufgabenstellung genauer erläutert. Es enthält auch die Spezifikationen über die Zählung der Farbwechsel im Paintshop-Teil und Constraintverletzungen im Fließband-Teil. Nachdem auf die Bewertung einer Lösung eingegangen wird, wird die Zielfunktion genau erläutert. Am Ende wird noch beschrieben, in welcher Form die Spezifikation angegeben ist, d.h. wie die Probleminstanzen in den Dateien abgespeichert sind.

2.1 Planungsprozess

Die Bestellungen der Kunden treffen täglich in Echtzeit in den Autofabriken ein.
Die tägliche Planung in den Fabriken besteht aus zwei Schritten:

1. Jedes Auto wird einem Produktionstag zugeordnet. Dieser hängt von den Produktionskapazitäten und dem Lieferdatum ab, das dem Kunden vom Verkäufer zugesprochen wurde.

2. Pro Tag muss ein Tagesplan entworfen werden, der die Reihenfolge der Autos am Förderband bestimmt. Natürlich sollen dabei so viele Anforderungen wie möglich an die Karosseriefertigung, die Lackierstation und das Fließband erfüllt werden.

Der Schwerpunkt liegt bei der Lackierstation und dem Fließband, da die Karosseriefertigung im Grunde keine Anforderungen stellt. Die Zuteilung eines Autos zu einem Produktionstag im ersten Schritt kann nicht rückgängig gemacht werden, da zwei verschiedene Optimierungsprogramme eingesetzt werden, um diese zwei Punkte getrennt zu behandeln. Im Moment übernimmt ein Lineares Programm die Einteilung im ersten Teil und ein Simulated Annealing Algorithmus die Planung im zweiten Teil.

2.2 Anforderungen and die Lackierstation

Das Hauptziel in der Lackierstationsphase besteht darin, möglichst wenig Lösungsmittel zu verbrauchen. Das Lösungsmittel wird verwendet, um den Lacksprüher bei einem Farbwechsel zu reinigen, wenn zwei aufeinanderfolgende Autos verschiedene Farben haben. Es ist naheliegend, Autos der selben Farbe zu gruppieren, um die Anzahl der Farbwechsel in der Sequenz zu minimieren. Anders formuliert, man versucht, die Anzahl der Lacksprüher-Reinigungen zu minimieren.
Der Lacksprüher hat jedoch eine Obergrenze an Einsätzen. Danach muss

der Sprühkopf gereinigt werden, auch wenn das nächste Auto theoretisch die selbe Farbe hätte. Es ist sogar nicht erlaubt, nach dieser notwendigen Reinigung wieder ein Auto derselben Farbe anzureihen. Diese Einschränkung ist eine Nebenbedingung, die eingehalten werden muss. Die Obergrenze der gleichfarbigen Autos wird in weiterer Folge „Maximale Farbblockgrenze“ genannt. Sie muss strikt eingehalten werden.

2.3 Anforderungen an das Fließband

Um die Last der Einbauarbeit am Fließband gut aufzuteilen, sollen Autos, die eine spezielle Einbaukomponente benötigen, möglichst gleichmäßig unter der Gesamtanzahl der Autos aufgeteilt werden.
Diese Anforderungen sind durch „Verhältnisconstraints“ der Form $\frac{Z}{N}$ angegeben, was bedeutet, dass innerhalb eines Fensters von N aufeinanderfolgenden Autos nur höchstens Z Autos eine Komponente benötigen sollen. Wenn z.B. $\frac{Z}{N} = \frac{3}{5}$ ist, dann sollen innerhalb von beliebigen fünf aufeinanderfolgende Autos in der Sequenz nur höchstens drei Autos die Komponente benötigen. Die Werte für Z und N variieren, je nachdem wie aufwändig die Einbauarbeit für eine bestimmte Komponente ist.
Der Spezialfall $Z = 1$ bedeutet, dass zwischen je zwei Autos, die eine gewisse Komponente benötigen, mindestens $N - 1$ liegen sollen, die diese Komponente nicht brauchen.
Es gibt zwei Klassen von Verhältnisconstraints, nämlich Constraints mit hoher und niedriger Priorität. Die Verletzung eines Constraints mit hoher Priorität führt wegen der Eigenschaften der Komponente bzw. des Autos zu einer sehr hohen Auslastung am Fließband. Wenn ein Constraint mit niedriger Priorität verletzt wird, entsteht im Gegensatz dazu möglicherweise nur eine kleine Unbequemlichkeit in der Produktion. Hohe Priorität sollte Vorrang gegenüber niedriger Priorität haben.
Verhältnisconstraints sind weiche Nebenbedingungen. Die Einhaltung kann nicht vor der Tagesplanung garantiert werden. Das Problem kann auch Constraints fordern, die unerfüllbar sind. Deshalb besteht das Ziel der Optimierung nur darin, die Anzahl der Constraintverletzungen möglichst gering zu halten.

2.4 Der Produktionstag $D - 1$

Wenn der Produktionstag D geplant wird, müssen die letzen Autos vom Vortag $D - 1$ auch berücksichtigt werden. Alle Autos aus $D - 1$ sind schon gefertigt, deshalb kann ihre Position klarerweise nicht verändert werden.

Diese sind aber dennoch wichtig für die Berechnung der ersten Autos am Tag D.
Wenn das letzte Auto von $D-1$ die gleiche Farbe wie das erste Auto von D hat, und die maximale Farbblockgrenze-Nebenbedingung nicht verletzt ist, so ergibt sich ein Farbwechsel weniger.
Dasselbe gilt auch für die Verhältnisconstraints. Die Berechnung der Verletzungen fangen zwar bei D an, aber womöglich wird das Ratio am Anfang nicht erfüllt, da die letzten Autos aus $D-1$ mit den ersten Autos aus D zusammen das Verhältnisconstraint verletzen.

2.5 Prioritäten bei der Optimierung

Es gibt drei Kriterien, nach denen optimiert wird:

1. Farbwechselminimierung in der Lackierstation
2. Erfüllung von Verhältnisconstraints mit hoher Priorität
3. Erfüllung von Verhältnisconstraints mit niedriger Priorität

Die Reihung legt den Zielfunktionswert fest, wobei das höchstgewichtete Kriterium mit dem Faktor 10000, das zweite Kriterium mit dem Faktor 100 und das letztgereihte Kriterium mit dem Faktor 1 gewichtet wird.
D.h. wenn die Erfüllung von Verhältnisconstraints mit hoher Priorität um eine Stufe wichtiger ist als die Minimierung der Farbwechsel, so muss die Anzahl der Farbwechsel um mindestens 100 verringert werden, damit sich eine zusätzliche Verletzung eines Constraints mit hoher Priorität bezahlt macht. Deshalb sollten die Prioritäten ein wichtiger Aspekt in der Optimierung darstellen.

2.6 Zählung der Constraintverletzungen

Die beste Lösung bei einem Constraint der Form $\frac{Z}{N}$ ist ein Produktionsplan mit höchstens Z Autos in einer beliebigen Sequenz von N aufeinanderfolgenden Autos, die eine gewisse Komponente brauchen. Die beste Lösung bei einem Constraint der Form $\frac{1}{N}$ ist ein Plan, wo sich zwischen je zwei Autos, die diese Komponente benötigen, mindestens $N-1$ Autos befinden, die die Komponente nicht benötigen.
Wenn die Constraints nicht eingehalten werden können, so muss die Anzahl der Verletzungen zumindest minimal gehalten werden, damit der Arbeitsprozess erleichtert wird.
Die Zählung selbst basiert auf dem Anlegen eines Betrachtungsfensters der

Länge von N Autos, wenn das zu untersuchende Verhältnisconstraint $\frac{Z}{N}$ lautet. Innerhalb dieses Fensters wird gezählt, wie viele Autos die Komponente benötigen. Wenn diese Anzahl größer als Z ist, dann sind Verletzungen vorhanden. Das Betrachtungsfenster wandert inkrementell durch die Sequenz.

Beispiel
Das Verhältnisconstraint beträgt $\frac{1}{5}$. Deshalb ist die Größe des Betrachtungsfensters fünf Autos. Betrachten wir folgende Sequenzen:

`___X___X__`: eine Verletzung, nämlich bei `X___X`
`___X__X___`: zwei Verletzungen, nämlich bei `_X__X` und bei `X__X_`

Wir können die folgende Formel bei einem Verhältnisconstraint $\frac{Z}{N}$ aufstellen:

$$\text{Vio} = \begin{cases} k - Z & \text{falls } k > Z \\ 0 & \text{sonst} \end{cases}$$

Vio ... Anzahl der Verletzungen
k ... Anzahl der Autos, die die Komponente benötigen

Der Tag $D - 1$
Wie oben erwähnt, müssen die letzten Autos von Tag $D - 1$ mit in die Berechnung der Verletzungen einbezogen werden. Das Betrachtungsfenster des Constraints $\frac{Z}{N}$ beim ersten Auto von Tag D muss also die letzten $N - 1$ Autos von Tag $D - 1$ beinhalten, um eine plausible Lösung zu erhalten.

Beispiel
Das Verhältnisconstraint beträgt $\frac{1}{5}$, die Autos von Tag $D - 1$ sind mit `A` gekennzeichnet, die von Tag D mit `B`.
Die Untersuchung der Sequenz `AA_A_AB__B` muss also mit `_A_AB` beginnen, die vier Autos vom Vortag enthält.

Der Tag $D + 1$
Grundsätzlich wird der Tag $D + 1$ nicht in Betracht gezogen, aber bei einer garantierten Verletzung muss das Betrachtungsfenster diese noch aufnehmen können.

Beispiel
Das Verhältnisconstraint beträgt $\frac{1}{5}$, und die Autos von Tag D enden auf

`_X____XX`.
`_X____XX` ergibt nicht nur eine Verletzung am Ende in `___XX`, wie man vielleicht meinen könnte, sondern die Verletzungen in dieser Sequenz betragen 4:

`___XX`: eine Verletzung
`__XX`: eine Verletzung
`_XX`: eine Verletzung
`XX`: eine Verletzung

Der Grund für diese Zählung liegt darin, dass am Tag $D + 1$ die ersten Constraintverletzungen nicht zu vermeiden wären.

2.7 Leicht und schwer zu erfüllende Verhältnisconstraints

Renault gibt Szenarien an, wo Constraints mit hoher Priorität in zwei Kategorien eingeteilt sind:

- Leicht zu erfüllende Verhältnisconstraints: Hierbei handelt es sich um einen Constraint über eine Komponente, wo Renaults industrielle Applikationen schon eine Sequenz ohne Verletzung aufbauen kann. Somit ist die Lösung ohne Verletzung möglich.
- Schwer zu erfüllende Verhältnisconstraints: Renault ist nicht im Stande, bei dieser Komponente eine Sequenz ohne Verletzungen aufzubauen. Es kann eine verletzungsfreie Lösung existieren, muss aber nicht.

Es gibt keine solche Einteilung für die Constraints mit niedriger Priorität.

2.8 Zielfunktion

Ausgehend von den Zielsetzungen, die bereits erwähnt sind, fassen wir unser Optimierungsziel in einer Funktion zusammen:

Minimiere $C_{CC} \cdot N_{CC} + C_{HPC} \cdot N_{HPC} + C_{LPC} \cdot N_{LPC}$

C_{CC} ... Kosten für eine Sprühkopfreinigung bei Farbwechsel
N_{CC} ... Anzahl der Farbwechsel
C_{HPC} ... Kosten für eine Verletzung eines Constraints mit hoher Priorität
N_{HPC} ... Anzahl der Verletzungen eines Constraints mit niedriger Priorität
C_{LPC} ... Kosten für eine Verletzung eines Constraints mit hoher Priorität
N_{LPC} ... Anzahl der Verletzungen eines Constraints mit niedriger Priorität

C_{CC}, C_{HPC} und C_{LPC} unterscheiden sich immer um einen Faktor von 100 und sind problemabhängig nach ihren Prioritäten geordnet.

2.9 Probleminstanzen

Um ein Problem zu laden, muss es in einem Format eingelesen werden, das die Spezifikation von „ROADEF Challenge 2005“ erfüllt. Jede Instanz besteht aus vier Dateien:

- optimization_objectives.txt
  Hier wird die Priorität der Bewertungsfunktionen angegeben. Grundsätzlich besteht die Minimierungszielfunktion aus drei Einzelbewertungen: Farbwechsel, Verletzung von Constraints mit hoher und niedriger Priorität.

  Header: `rank;objective name;`
  Danach wird die Wichtigkeit der Einzelbewertungen angegeben, wobei sich jede Prioritätsstufe um einen Faktor von 100 unterscheidet.

  Beispiel:
  ```
  1;high_priority_level_and_easy_to_satisfy_ratio_constraints;
  2;paint_color_batches;
  3;low_priority_level_ratio_constraints;
  ```

  Hier würde die Zielfunktion lauten:
  Minimiere $100 \cdot N_{CC} + 10000 \cdot N_{HPC} + 1 \cdot N_{LPC}$

- paint_batch_limit.txt
  Hier wird die Anzahl der aufeinanderfolgende Fahrzeuge angegeben, die die gleiche Farbe besitzen dürfen. Nach dieser Anzahl wird der Sprühkopf gereinigt und eine neue Farbe muss eingelegt werden.

  Header: `limitation;`
  Danach kommt einfach die Zahl, gefolgt von einem Strichpunkt.

  Beispiel:
  ```
  20;
  ```

- ratios.txt
  Hier wird angegeben, wie die Constraints im Fließband Teil aussehen.

  Header: `Ratio;Prio;Ident;`
  Danach kommen der Reihe nach alle Constraints in der Form von einem Bruch, eine Angabe, ob sich dabei um ein Constraint mit hoher oder niedriger Priorität handelt und die Bezeichnung des Constraints.

  Beispiel:
  `5/6;1;HPRC1;`
  `1/3;0;LPRC4;`

  Der erste Constraint besagt, dass von sechs aufeinanderfolgende Fahrzeuge nur höchstens fünf die Komponente `HPRC1` benötigen sollen, damit dieser Constraint nicht verletzt wird. Es ist außerdem ein Constraint mit hoher Priorität.
  Der zweite Constraint besagt, dass von drei aufeinanderfolgende Fahrzeuge nur höchstens einer die Komponente `LPRC4` benötigen sollen, damit dieser Constraint nicht verletzt wird. Es ist außerdem ein Constraint mit niedriger Priorität.

- vehicles.txt
  Hier werden alle Fahrzeuge und ihre Eigenschaften aufgezählt.

  Header: `Date;SeqRank;Ident;Paint Color;HPRC1;HPRC2;\ldots{}`
  Die einzelnen Werte besagen folgendes:

  - Date: Ein Datum, das angibt, ob das Fahrzeug vom Vortag stammt, oder zum aktuellen Tag gehört. Wenn es zum Vortag gehört, dann werden lediglich nur die Constraintverletzungen berücksichtigt und die Position bleibt fix.
  - SeqRank: Die Initialreihenfolge des Fahrzeugs.
  - Ident: Die Identifikation des Fahrzeugs besteht aus einer Zahl.
  - Paint Color: Die Farbe des Fahrzeugs, angegeben durch eine Zahl.
  - HPRC1; ... : Ein Bitcode, der angibt, ob das Fahrzeug eine gewisse Komponente benötigt oder nicht.

  Beispiel:
  `2003 38 3;509;022033820002;2;1;0;0;0;0;0;0;0;0`

```
2003 38 3;510;022033830048;2;1;0;1;1;0;0;0;0;1
2003 38 4;1;022033840162;3;1;0;0;0;0;0;1;0;0
2003 38 4;2;022033840386;4;1;1;0;1;0;0;1;0;0
2003 38 4;3;022033830379;4;1;0;0;0;0;0;0;0;0
2003 38 4;4;022033830171;4;1;0;0;0;0;0;0;1;0
```

Um eine Probleminstanz in die vorliegende Applikation zu laden, müssen diese vier Dateien in einem Verzeichnis vorhanden sein. Danach kann eine davon ausgewählt werden, um alle vier hineinzuladen.

3 Verwandte Forschungsgebiete

Es gibt eine Reihe von verwandten und ähnlichen Projekten, die ebenfalls auf das „Car Sequencing“ Problem spezialisiert sind. Einige davon, je nach dem wie verwandt sie mit dem aktuellen Problem sind, werden im Folgenden mit mehr oder weniger Details vorgestellt.
Ebenfalls bemerkenswert ist die Tatsache, dass keines der erwähnten Papers auf die Minimierung der Farbwechsel eingeht, sondern alle lediglich das Fließband betrachten.

3.1 Reihenfolgeplanung von Bitvektoren mit Distanzbedingungen

In „Sequencing Bitvectors with Distance Constraints [8]“ von R. Nickel und W. Hochstättler wird das Problem als Mixed Integer Problem formuliert. Es sind β-dimensionale Bitvektoren in einer endlichen Multimenge $F = \{b_1, \ldots, b_1, b_2, \ldots, b_2, b_\alpha, \ldots, b_\alpha\}$ gegeben, und gesucht wird eine Reihung der Vektoren, sodass für jede Zeile j gilt: Zwischen je zwei Bits mit dem Wert „1“ sollten mindestens $\overline{m}_j$ und höchstens $\underline{m}_j$ Bits mit dem Wert „0“ liegen. Diese Formulierung beschreibt klarer Weise nur Fließband Constraints der Form $\frac{1}{N}$. Constraints der Form $\frac{Z}{N}$ mit $Z \geq 2$ können nicht mit einem simplen $\overline{m}_j$ ausgedrückt werden. Nichts desto trotz ist die Problemstellung sehr ähnlich, und die Komplexitätsanalyse bezieht sich auch auf einen Beweis in diesem Paper. Deshalb gehen wir etwas genauer auf dieses Paper ein.

Distance-Constraint-Bitvector-Sequencing (DCBS)
Gegeben: Multimenge $F = \{b_1, \ldots, b_1, b_2, \ldots, b_2, b_\alpha, \ldots, b_\alpha\}$ von β-dimensionalen 0-1 Vektoren und Schranken $\underline{m}_j, \overline{m}_j \in N$ mit $\underline{m}_j < \overline{m}_j (j = 1 \ldots \beta)$.
Gesucht: Eine Sequenz $b_{i_1}, b_{i_2}, \ldots, b_{i_n}$ der Vektoren in F, sodass für jede Komponente j zwischen je zwei „1“ mindestens $\underline{m}_j$ und höchstens $\overline{m}_j$ „0“ existieren.

Mixed Integer Program
Sei n die Länge der Sequenz, die mit k indiziert wird, und α die Anzahl der verschiedenen Bitvektoren, die mit i indiziert werden. Wir führen Entscheidungsvariablen x_{ik} ein, die den Wert „1“ besitzen, dann und nur dann, wenn der Bitvektor b_i auf Position k sequenziert wird. Wenn F α verschiedene Bitvektoren

$b_i (i = 1 \ldots \alpha)$ enthält, dann beschreibt N_i die Anzahl der gleichen Bitvektoren b_i. Somit lässt sich das Problem als folgendes Lineares Programm (LP) formulieren:

1. $\sum_{i=1}^{\alpha} x_{ik} = 1 \qquad \forall k = 1 \ldots n$
2. $\sum_{k=1}^{n} x_{ik} = N_i \qquad \forall i = 1 \ldots \alpha$
3. $X_{j,0} = 0 \qquad \forall j = 1 \ldots \beta$
4. $X_{j,k} = X_{j,k-1} + \sum_{i=1}^{P} x_{ik} b_{ij} \qquad \forall j = 1 \ldots \beta, \forall k = 1 \ldots n$
5. $X_{j,k} - X_{j,k-\underline{m}_j} \leq 1 \qquad \forall j = 1 \ldots \beta, \forall k = \underline{m}_j + 1 \ldots n$
6. $X_{j,k} - X_{j,k-\overline{m}_j} \geq 2 \qquad \forall j = 1 \ldots \beta, \forall k = \overline{m}_j + 1 \ldots n$
7. $x_{ik} \in \{0, 1\}$

Gleichungen (1) und (2) garantieren, dass auf jeder Position k sich genau ein Vektor befindet, und jeder Vektor b_i genau N_i Mal in der Sequenz auftritt. (3) und (4) zählen einfach die Anzahl der Auftritte der Komponente j bis zur Position k. (5) und (6) sind die Distanzconstraints des Problems. In Summe ergibt das ein Constrainterfüllungsproblem mit in etwa $n+\alpha+3\beta n$ Constraints und $\alpha n + \beta n$ Variablen.

Der Beweis, dass das Problem NP schwer ist, geht über die Reduktion auf das „Three Partition Problem“ und ist umfangreich in [8] beschrieben. Das Problem bleibt immer noch NP schwer, wenn folgende Vereinfachung getroffen wird:

(DCBS) bleibt NP schwer wenn wir die folgende Simplifizierung durchführen:

1. Die Constraints aller Komponenten sind identisch und auf $\underline{m}_j = 1$ und $\overline{m}_j = 3$ vereinfacht.

2. Die Obergrenzen $\overline{m}_j$ werden alle ignoriert und die Untergrenzen sind auf $\underline{m}_j = 1$ fixiert.

Der genaue Beweis wird unten im Kapitel 8, „Komplexität“, angegeben.

Polynomielle Betrachtung
Es gibt noch eine Betrachtungsweise, die das Problem in der Theorie polynomiell macht, wenn folgende Einschränkung getroffen wird: Wenn $\beta \leq \overline{\beta}$ für ein fixes $\overline{\beta}$ gilt, dann wird α, die Anzahl der möglichen verschiedenen Bitvektoren auch eingeschränkt, da $\alpha \in O(2^\beta)$ gilt. Die endgültige Komplexität liegt im Endeffekt in $O(n^\alpha)$, was für die Praxis leider nicht viel Bedeutung hat. Falls wir diese Komplexität in Kauf nehmen können, bietet [8] ein Dynamisches Programm für die exakte Lösung an.

Parameterized Heuristic Procedure
Um eine praxisnahe Lösung zu bieten, beschreibt das Paper auch einen „Parameterized Heuristic Procedure“ Algorithmus. Im Gegensatz zu Greedy, der bei einem solchen Problem wahrscheinlich zuerst alle leicht zu sequenzierenden Vektoren anordnet (um somit die Constraintverletzungen minimal zu halten), wodurch aber schwierige Teile übrig bleiben, versucht diese Lösung eine Einordnungsreihenfolge zu finden, die einem möglichst optimalen Kompromiss zwischen leicht und schwer entspricht. Das wird in einem kleinen Block von Pseudocode am besten demonstriert:

$S = \emptyset$
solange nicht alle Vektoren sequenziert sind
  setze $p_{min} = \min\{p_i \mid S + b_i$ verletzt p_i Constraints, $i = 1 \ldots \alpha\}$
  setze $b^* =$ Element aus $\{b_i \mid S + b_i$ verletzt p_{min} Constraints, $i = 1 \ldots \alpha\}$
    welches den möglichst optimalen Fluss von 1er in den
    Komponenten garantiert.
  gib b^* in die Sequenz S
end solange

3.2 Reihenfolgeplanung mit Bestrafungsmatrix

Die Problemstellung in „Traditional Heuristic versus Hopfield Neural Network Approaches to a Car Sequencing Problem [5]“ ist ebenfalls sehr ähnlich

wie die aktuelle. Sie behandelt ebenfalls offline Scheduling, wo keine Einschränkung in der Umordnung der Bitvektoren existiert. Eine Minimumdistanz m_i gibt an, wie viel Leerraum zwischen Vektoren vorhanden sein soll, die beide die Komponente i benötigen. Wie schon in [8] erwähnt, kann diese Formulierung ebenfalls nur Constraints der Form $\frac{1}{N}$ beschreiben. Dennoch beschreibt dieses Paper einen interessanten Zusatz, der besagt, dass die Verletzung eines Constraints nicht immer gleich zu bewerten sein soll. Eine Verletzung der Komponente i wegen einer nicht eingehaltenen Distanz $d_i < m_i$ bestraft den Lösungswert abhängig davon, wie stark d_i von m_i abweicht. Wenn die Abweichung z.B. nur bei eins liegt, so ist es weniger störend wie wenn $d_i = 1$ wäre, falls $m_i > 2$ gilt. Für diesen Zweck wird eine Penalty Matrix P eingeführt. Der Eintrag P_{ij} beschreibt den Bestrafungswert, wenn in der j-ten Komponente der Zwischenraum i beträgt. Ist $i \geq m_j$, so gilt klarer Weise $P_{ij} = 0$.

Nehmen wir an, es sind N Autos und M Modelle in der Sequenz.
Wir definieren:

$$x_{kj} = \begin{cases} 1 & \text{falls das } k\text{-te Fahrzeug in der Sequenz von Typ j ist} \\ 0 & \text{sonst} \end{cases}$$

$\forall k = 1 \ldots N, \forall j = 1 \ldots M$

Als ein Beispiel wählen wir $N = 10$ und $M = 4$ (z.B. Sedan(S), Utility(U), Luxury(L) und Wagon(W)), und die Distanzconstraints zwischen Autos mit gleichen Typen betragen 2, 5, 8 und 3 für diese vier Modelle. Eine mögliche Lösung wäre:

$$X = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Diese Lösung entspricht der Sequenz S, S, U, U, U, L, L, W, W, W. Natürlich ist diese Reihung keine gute Lösung. Deshalb muss eine Bestrafung für eine

schlechte Sequenz wie oben groß genug sein, damit die Permutation diese eliminiert.

Dafür führen wir die Bestrafungsmatrix P ein. Angenommen auf der k-ten Position in der Sequenz ist ein Auto vom Typ Luxury ($j = 3$), dann ist die Bestrafung für die $(k-1)$-te Position in der Sequenz, wenn das Fahrzeug zum Typ Luxury gehört, der Eintrag $P_{1,3}$, die Bestrafung für die $(k-2)$-te Position ist $P_{2,3}$, usw. Als Beispiel nehmen wir für die Matrix P folgende Werte an:

$$P = \begin{pmatrix} 4 & 9 & 22 & 7 \\ 2 & 7 & 10 & 5 \\ 0 & 5 & 18 & 3 \\ 0 & 3 & 16 & 0 \\ 0 & 1 & 14 & 0 \\ 0 & 0 & 12 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 8 & 1 \end{pmatrix}$$

In diesem Fall ist es wichtig, die Distanz bei Typ Luxury einzuhalten, da sie die höchsten Strafkosten verursachen.

Mit Hilfe von X und P kommt man auf die folgende CSP (Constraint Satisfaction Problem) Formulierung:

$$\text{Minimiere } \sum_{k=1}^{N} \sum_{j=1}^{M} X_{kj} \sum_{i=1}^{k-1} P_{(k-i)j} X_{ij}$$

1. $\sum_{k=1}^{N} X_{kj} = D_j N_i \qquad \forall j = 1 \ldots M$
2. $\sum_{j=1}^{M} X_{kj} = 1 \qquad \forall k = 1 \ldots N$
3. $X_{kj} \in \{0, 1\}$

Die Komplexität dieser Formulierung ist ebenfalls interessant. Die Autoren von [5] behaupten, dass sie nur als ein nichtlineares ganzzahliges Programm mit einer indefiniten quadratischen Form formuliert werden kann. Wegen der Komplexität arbeitet [5] nur mit heuristischen Algorithmen. Standardheuristiken wie Steepest Descent oder Simulated Annealing, aber auch eine Lösung mittels einem Hopfield Neuronalnetzwerk werden dort eingesetzt.

3.3 Reihenfolgeplanung in Echtzeit

Eine Vielzahl an Papers wie z.B. „The Problem of JIT Dynamic Sequencing. A Model and a Parametric Procedure [2]“ beschäftigt sich mit Echtzeitplanung (Just-In-Time Scheduling), d.h. dynamische Planung von Fahrzeugen, die zwischen der Lackierstation und dem Fließband durch einen Puffer mit n Förderbändern geschickt werden, um somit eine beschränkte Umordnung zu ermöglichen. Nach dem Paintshop kann das Fahrzeug wahlweise auf ein Pufferförderband F_i mit $i \leq n$ gegeben werden, falls F_i noch nicht voll ist. Vor dem Fließband wird ebenfalls entschieden, von welchem Pufferförderband das nächste Fahrzeug eingereiht wird.

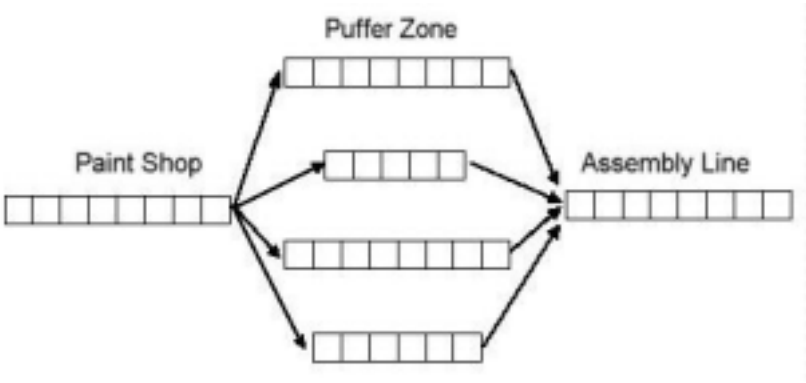


Abbildung 3: Just-In-Time Scheduling Aufbau

„Facts and Questions about the Maximum Deviation Just-In-Time Scheduling Problem [6]“ betrachtet ebenfalls ein Echtzeitproblem, aber die Aufgabe ist etwas anders: In diesem Paper wird angenommen, dass es n verschiedene Komponenten gibt, die alle Produktionskosten von einer Zeiteinheit besitzen. Wenn der Bedarf an jeder Komponente d_i, $i = 1 \ldots n$ beträgt, dann existiert eine ideale Produktionsrate r_i der Komponenten mit $r_i = \frac{d_i}{\sum_{j=1}^{n} d_j}$. Diese ideale Rate wird in der Realität jedoch kaum erreicht, deshalb ist das Ziel in diesem Paper, die totale und maximale Abweichung zur Idealrate zu minimieren.

Alle diese Ideen sind sehr gut und praxisorientiert, da durch die Umordnung im Zwischenpuffer die Constraints der Lackierstation und des Fließbands ge-

trennt gezielter optimiert werden können. Da dies jedoch nicht die eigentliche Aufgabenstellung der Arbeit darstellt, wird auf diese Lösungsansätze nicht weiter eingegangen.

4 Human Guided Search (HuGS)

Hier gehen wir näher auf das HuGS System ein. Neben der allgemeinen Einführung über die Ideen, die dahinter stehen, sollen die Funktionsweisen und die Vorgehensweisen einer Suche mittels HuGS erwähnt sein. Zum Schluss gibt es einen Überblick über die Klassenstruktur, die beim Schreiben einer neuen Applikation sehr wichtig ist.

4.1 Einführung

Da die ganze Paintshop Applikation in HuGS eingebaut ist, ist es wichtig, die grundlegenden Vorgehensweise von HuGS zu verstehen. Die Motivation dahinter ist, wie bereits erwähnt, die Integration von menschlichen Ideen während des automatischen Optimierungsprozesses. Außerdem können auch mehrere Anwender gleichzeitig über die Problematik diskutieren, und bei der Optimierung mitwirken. Folgende Abbildung zeigt, wie das an einem Optimierungstisch ablaufen könnte.

Abbildung 4: Optimierungstisch (Quelle: [4])

Hier an dieser Stelle noch ein Zitat aus „The HuGS Platform: A Toolkit for Interactive Optimization [4]“, um die Motivation zu erhöhen:

„Forschung im Bereich von Optimierungsproblemen wie Routing, Layout, Scheduling, etc. ist auf die Entwicklung von automatischen Algorithmen

konzentriert, die einen exponentiell großen Lösungsbereich möglichst effizient durchsuchen. Gewöhnlich besteht die Rolle des Benutzers in so einem System darin, das Problem zu spezifizieren, Schwerpunkte auf Kriterien für die Ergebnisse festzulegen und dann den Suchprozess zu initialisieren.

Das Human-Guided-Search-Framework, motiviert die Benutzer, im Optimierungsprozess mitzuwirken. Es bieten einfache visuelle Metaphern, die den Benutzern eine ausführliche Untersuchung des Suchraums erlauben. Wir bieten außerdem Middleware, die eine rasche Integrierung von Problemen in das Framework erlauben.
Das Ziel ist auf Aspekte gerichtet, die oft im Bereich der Optimierung vernachlässigt werden, aber dennoch für Menschen essentiell wichtig sind, um eine brauchbare Lösung zu einem Problem suchen. Die Benutzer müssen die generierten Lösungen nachvollziehen und ihnen vertrauen können, um sie effizient einsetzen zu können. Weiteres ist es oft unmöglich, alle Suchparameter vor dem Suchprozess mit sinnvollen Werten zu belegen. Stellen Sie sich vor, jemand muss einen monatlichen Arbeitsplan aufstellen. Er muss die Lösung nachvollziehen können, um es den Angestellten gut zu übermitteln. Noch wichtiger ist es zu verstehen, wie Modifikationen gemacht werden können, wenn mögliche Änderungen auftreten. Außerdem gibt es oft besondere Aspekte und Spezialwünsche, die schlecht dem Computer mitteilbar sind. Wenn der Benutzer bei der Generierung des Scheduling Plans aktiv mitwirkt, so kann er den Vorgang zu einer plausible Lösung hinsteuern, die alles realitätsbezogene Wissen berücksichtigt.
Zusätzlich kann die menschliche Interaktion aus dem folgenden Grund das Ergebnis verbessern: In den Bereichen wie visuelle Wahrnehmung, Lernen durch Erfahrung, strategische Überlegungen, etc. übertrifft der Mensch immer noch den Computer. Wenn die Visualisierung anschaulich ist, kann der Mensch den Computer in den aussichtsvollen Regionen des Lösungsraums verstärkt suchen lassen.“ (Übersetzung vom Autor)

4.2 Terminologien

Hier werden die wichtigsten Begriffe angeführt, die im Zusammenhang mit HuGS immer wieder auftauchen. Es ist notwendig, eine eindeutige Bedeutung festzulegen, um im Weiteren die Funktionsweise von HuGS zu erläutern. Die Erklärung der Terminologien wurden aus „Human-Guided Search: Survey and Recent Results [1]“ entnommen.

Wir benutzen folgende Abstraktionen für eine Beschreibung der HuGS Ap-

plikationen: *Probleme*, *Lösungen*, *Moves* und *Nodes*.

Ein *Problem* ist eine Instanz von einem Typ eines Problems, das optimiert werden soll. Ein Protein-Problem beinhaltet z.B. eine Sequenz von Aminosäuren.

Das Ziel der Optimierung besteht darin, eine *Lösung* zu dem gegebenen Problem zu finden. Eine Lösung bei einem Delivery-Problem besteht z.B. aus einer Sequenz von Kundschaften. Wir nehmen an, dass in jeder Applikation eine Möglichkeit existiert, zwei beliebige Lösungen miteinander zu vergleichen und als Resultat eine Aussage entsteht, die besagt, dass eine Lösung besser als die andere ist, oder dass beide gleich gut sind.
Es ist durchaus möglich, temporär eine ungültige Lösung zu erzeugen, wo bestimmte Hardconstraints verletzt sind. Dies hat den Vorteil, dass das globale Optimum somit leichter erreichbar wird. Am Ende einer solchen Vorgehensweise muss natürlich ein Reparaturalgorithmus die Verletzungen wieder beseitigen.

Für jedes Problem müssen *Moves* entworfen werden, die auf die Lösungen eine Transformation anbringen. Wenn ein Move auf eine Lösung angewandt wird, entsteht eine neue Lösung. Für eine Delivery-Applikation kann z.B. ein Move darin bestehen, einen Kunden aus der Route zu geben oder einen neuen mit aufzunehmen.

Als letztes nehmen wir noch an, dass jedes Problem über eine endliche Anzahl an *Nodes* verfügt. Die Nodes sind die elementaren Bestandteile eines Problems. Bei einer Delivery Applikation wären das die Kunden, bei einer Protein-Applikation die Aminosäuren, usw. Jeder Move ist definiert als eine Änderung eines oder mehrerer Nodes in ihrem Zustand oder ihrer Reihenfolge.

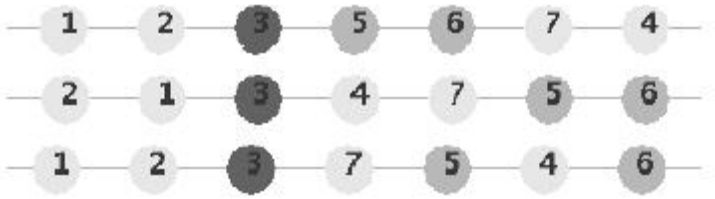
Die Entscheidung, was als ein Node definiert wird und welche Moves benutzt werden, ist keine leichte designerische Angelegenheit.

4.3 Mobilities

Weiters gibt es noch einen sehr wichtigen Mechanismus, die sogenannten *Mobilities*. Sie erlauben den Benutzern alle Nodes getrennt zu konfigurieren, damit sie von den Suchalgorithmen unterschiedlich behandelt werden. Jedem Node ist eine Mobility zugeordnet: high, medium oder low.

Beispiel
Angenommen ein Problem enthält sieben Knoten und eine Lösung besteht aus einer Reihenfolge der sieben Knoten. Ein Move besteht aus einer Vertauschung von zwei Knoten. Eine Lösung ist gegeben durch:

Die Mobilities von den Knoten 1, 2, 4 und 7 sind high (hell), Knoten 5 und 6 sind medium und Knoten 3 ist low (dunkel).
Angenommen der Suchalgorithmus kann dann nur zwei adjazente Knoten vertauschen, wenn mindestens einer davon Mobility high hat und keiner davon Mobility low. Dies führt zu einem eingeschränkten Suchraum und die entstehenden Lösungen könnten dann so aussehen:



Beachtenswert ist, dass der Knoten 3, der auf Mobility low gesetzt ist, das Problem in zwei kleinere Subprobleme teilt. Ebenso können zwar die Knoten mit Mobility medium ihre Positionen ändern, aber ihre relative Reihenfolge zueinander bleiben konstant.
Mobilities können den Suchraum drastisch verkleinern. In diesem Beispiel gibt es zwölf verschiedene Anordnungen, die die Mobility Constraints erfüllen, aber ohne Mobilities gäbe es $7! = 5040$ Möglichkeiten!

4.4 Funktionsweise

Die Funktionsweise von HuGS ist recht einfach erklärt. Nachdem eine Probleminstanz in die Applikation geladen wurde, fangen die Suchalgorithmen auch schon an zu arbeiten, indem Moves auf die Lösungen angewandt wer-

den. Das Resultat wird laufend in der Visualisierung angezeigt. Der Kern von HuGS besteht aus seinen Suchalgorithmen.

4.4.1 Exhaustive Search

Es gibt zwei Varianten dieser Suchalgorithmen: Steepest Descent und Greedy. Beide Algorithmen führen zuerst alle Moves aus, die unter Beachtung der Mobilities ausführbar sind. Danach kombinieren sie immer zwei dieser Moves miteinander, dann 3, usw. Der Steepest Descent sucht immer weiter nach dem Move, der am meisten Gewinn bringt, wobei Greedy wiederum sofort den ersten Move ausführt, der die aktuelle Lösung verbessert und dann wieder von vorne alles durchmacht.

4.4.2 Tabu Search

Während die Ergebnisse durch die Hilfe von menschlicher Interaktion schon mit den simplen Suchalgorithmen recht zufriedenstellend erscheinen, sind diese mit dem Tabu Search Algorithmus noch weiter verbesserbar.
Tabu Search ist eine Heuristik für das Durchkämmen eines großen Suchraums. Wie andere Lokale Suchalgorithmen benutzt Tabu auch die Nachbarschaftsstruktur, die im Lösungsraum definiert wird. In jeder Iteration evaluiert Tabu alle Nachbarschaftslösungen der momentanen Lösung und geht dann zu der über, die am meisten Verbesserung verspricht. Nachbarschaften sind durch die Problemstellung und somit durch die Moves festgelegt.
Der klassische Ansatz, um eine effiziente Suche zu betreiben, wird durch die „Ausweitungsmethode“ garantiert: Eine „Tabu“ Liste wird immer mitgeführt, die die Ausführung bestimmter Moves temporär verbietet, um das Backtracking zu verhindern. Aktuelle Tabu Algorithmen beinhalten oft auch „Verschärfungsmethoden“, die vielversprechende Suchregionen verstärkt oder sogar vollständig durchsuchen. (Das ist aber in der momentanen Version von HuGS nicht enthalten.)
HuGS präsentiert jedoch einen Guidable Tabu Search (GTabu) Algorithmus. Dieser analysiert immer die momentane Lösung und die Konfiguration der Mobilities. In jeder Iteration evaluiert GTabu alle unter Beachtung der Mobilities erlaubten Moves, um den besten davon ausfindig zu machen. Er führt diesen Move aus, was die momentane Lösung durchaus temporär auch verschlechtern kann und aktualisiert die Mobilities. Dadurch lässt sich eine kreisförmige Suche verhindern und der Algorithmus wird motiviert, neue Regionen zu erforschen. GTabu aktualisiert die Mobilities auf zwei Arten:

1. Es gibt die sogenannte „Memory“ Funktion, die das sofortige Backtracking verhindert, indem die Mobilities der gerade geänderten Nodes auf medium gesetzt werden. Wenn z.B. im vorigen Beispiel zwei Nodes ihre Positionen austauschen, dann erhalten beide Nodes die Mobility medium. Somit können diese beiden Nodes nicht mehr unmittelbar danach ihre Positionen zurück tauschen.
   Die Mobilities werden nach einer Anzahl von Iterationen, die vom Benutzer spezifiziert wird, wieder auf den ursprünglichen Wert zurückgesetzt. Dieser Parameter ist die *memSize*. Die meisten Tabu Algorithmen haben einen solchen Mechanismus.

2. Es gibt auch die sogenannte „Diversify“ Funktion, welche den Algorithmus motivieren soll, die Moves zu wählen, die in den vergangenen Iterationen nicht oft angewandt wurden. Der Algorithmus führt eine Liste von allen Nodes, die nach ihrer Anzahl an Änderungen absteigend sortiert sind. Der Diversity Wert eines Nodes ist die Position in dieser Liste, die durch die Gesamtanzahl der Nodes dividiert wird. Der Diversity Wert eines Moves ist der Durchschnitt aller Diversity Werte der Nodes, die dieser Move anrührt. Der Diversity Wert einer Suche ist der Durchschnitt aller Diversity Werte der Moves, die in der Zeit ausgeführt wurden, seitdem das letzte globale Optimum gefunden wurde. Der Benutzer kann einen minimalen Diversity Wert *minDiv* zwischen 0 und 1 für die Suche angeben. Wenn der Diversity Wert der Suche unter diesen Grenzwert liegt, dann werden die Mobilities aller Nodes, die einen Diversity Wert größer als *minDiv* besitzen, für eine Iteration auf medium gesetzt. Das zwingt dem Algorithmus, die Moves auf Nodes auszuführen, die weniger oft verändert sind.

4.5 Klassenstruktur

HuGS ist ein großes Java Framework, das recht abstrakt implementiert ist, um viel Ableitungsfreiraum zu besitzen. Um neue Problemapplikationen zu integrieren, müssen lediglich die Schnittstellenkomponenten abgeleitet und implementiert werden. Folgende Klassen sind abzuleiten:

- `Hugs.java`: Die Hauptklasse, von wo aus alles gestartet wird. Sie erfüllt hauptsächlich Initialisierungsaufgaben und konstruiert die anderen Klassen. Die Hauptklassen aller integrierten Applikationen sind von ihr abgeleitet.

- `Problem.java`: fasst die Spezifikation des Problems zusammen. Wei-

teres muss sie imstande sein, Probleminstanzen einzulesen und auszugeben.

- `Solution.java`: stellt eine Lösung des Problems dar. Die Zustände aller Nodes und der Score dieser Konfiguration wird darin gespeichert. Genau so wie Problem.java soll sie imstande sein, abgespeicherte Lösungen einzulesen und generierte abzuspeichern.

- `Node.java`: stellt den essentiellen Bestandteil eines Problems dar. Bei einem Grafenproblem wäre das ein Knoten, bei einem Packungsproblem ein Objekt, etc. Die Konfiguration der Menge aller Nodes stellt eine Lösung dar. Der Zustand bzw. Parameter eines Nodes wird durch einen Move verändert.

- `Move.java`: verursacht eine Änderung in der Lösung, um eine Nachbarlösung zu generieren. Der Typ des Moves, die Änderungsparametern, etc. sind klarer Weise darin enthalten.
  Je nach Problemstellung können oder sollen mehrere Arten von Moves abgeleitet und implementiert werden, die verschiedene Verhaltensweisen besitzen, um auf möglichst viele Wege einen möglichst großen Suchraum abzuarbeiten.

- `MoveGenerator.java`: erzeugt eine Liste von Moves, die dann von den Suchalgorithmen ausgeführt werden. Sie berücksichtigt den SearchAdjuster und modifiziert dementsprechend die Movesorten, Suchtiefe, etc.

- `SearchAdjuster.java`: übernimmt die Suchparameter durch Benutzereingabe. Welche Moves sollen generiert werden? Wie tief soll die Suche laufen?

- `Score.java`: beinhaltet die Bewertungsfunktion eines Problems. Nach diesem Kriterium wird optimiert. Das ist auch der Wert, der besagt, ob die Lösung besser als eine andere ist oder nicht.

- `Visualization.java`: zeichnet eine Lösung visuell auf dem Bildschirm. Es ist auch die Schnittstelle zwischen dem Programm und dem Benutzer, da alle Benutzerinteraktionen auch über ihn eingegeben werden.

Die anderen Klassen in search, support und utils sind für einen Entwickler weniger wichtig und können als Black Boxes betrachtet werden. In search sind die Suchalgorithmen implementiert, sowie der Searchmanager, der den ganzen Suchprozess verwaltet. In utils und support sind im wesentlichen viele Hilfsklassen und Hilfsmethoden, auf denen HuGS zurückgreift.

4.6 Andere Applikationen unter HuGS

Hier werden ein paar weitere Applikationen vorgestellt, die ebenfalls das HuGS Framework benutzen, um schwierige Probleme zu lösen:

Labeling
Hier wird das 4-Positionen Labeling Modell betrachtet, wo die Labels nur links oben, rechts oben, links unten und rechts unten direkt an einem Knoten (entspricht z.B. einer Stadt) angebracht werden können. Ziel ist es, so viele Labels wie möglich auf einer Landkarte unterzubringen, sodass sich keine zwei Labels überschneiden.
Es verfügt außerdem über eine Prioritätenfunktion, die besagt, dass nicht jeder Knoten den gleichen Wert besitzt, ob er gelabelt wird oder nicht. Wenn zwei Knoten sehr dicht zueinander liegen, sodass sich zwei Labels nicht gleichzeitig ausgehen, so ist es z.B. wichtiger, die Großstadt Wien zu labeln als die anliegende kleinere Stadt Mödling. Dafür verwendet die Applikation einen Prioritätswert für jede zu labelnde Stadt.

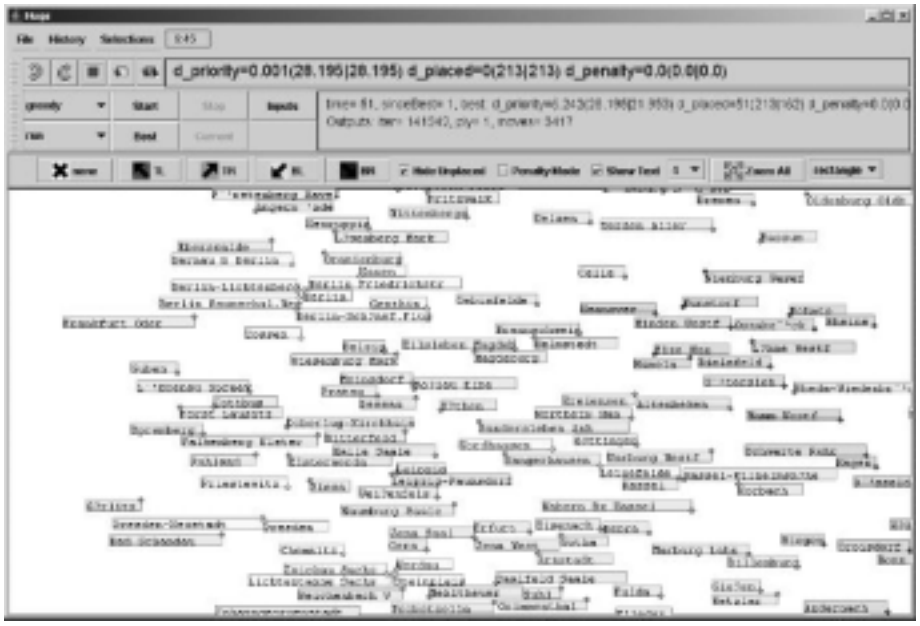


Abbildung 5: Labeling Applikation

Jobshop
In dieser Variation des intensiv studierten Jobshop Problems werden n Jobs und m Maschinen betrachtet, wo jeder Job aus m Arbeitsoperationen zusammengesetzt ist, die in einer bestimmten Reihenfolge und jeweils von einer bestimmten Maschine auszuführen sind. Die Arbeitsaufträge auf den Maschinen dürfen sich nicht überlappen, aber in welcher Reihenfolge die Maschinen die Jobs ausführen, ist variabel. Ziel ist es, die Zeit zu minimieren, die benötigt wird, bis der letzte Job fertiggestellt wurde.

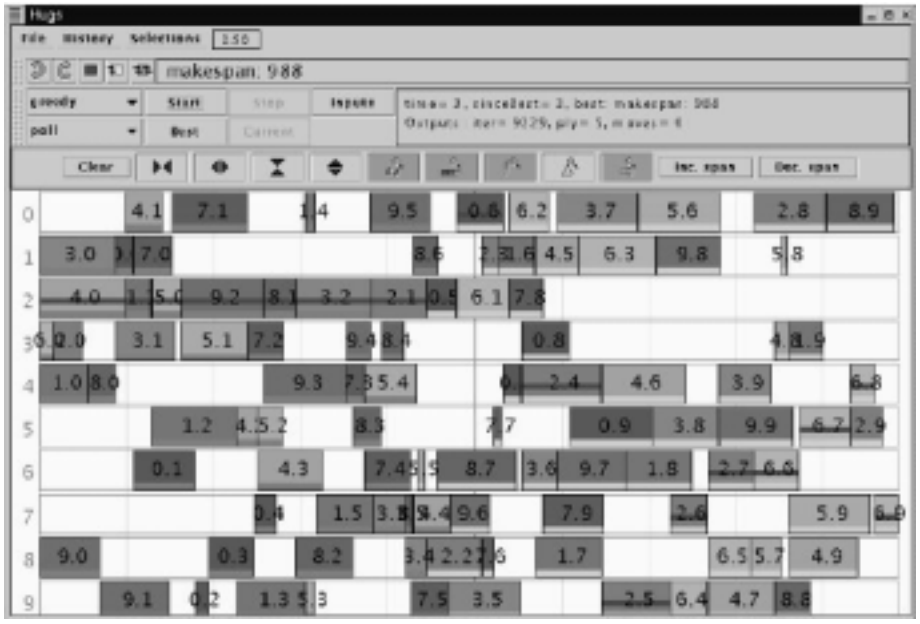


Abbildung 6: Jobshop Applikation (Quelle: [4])

Delivery
Die Delivery Applikation ist eine Abwandlung des Traveling Salesman Problems, wo es hier nicht notwendig ist, alle Städte zu besuchen. Statt dessen geht es darum, möglichst viele Städte zu besuchen, um Pakete zu liefern, wenn eine maximale Weglänge angegeben ist, die man zurücklegen darf.
Eine Probleminstanz beinhaltet eine Menge an Kundschaften, die sich an einem konstanten geographischen Ort befinden, den Start- bzw. Zielpunkt und die maximale Weglänge. Alle Kundschaften haben eine unterschiedli-

che Priorität, die visuell durch die Größe des Knotens dargestellt wird. Der Startpunkt ist das schwarze Quadrat in der Mitte.

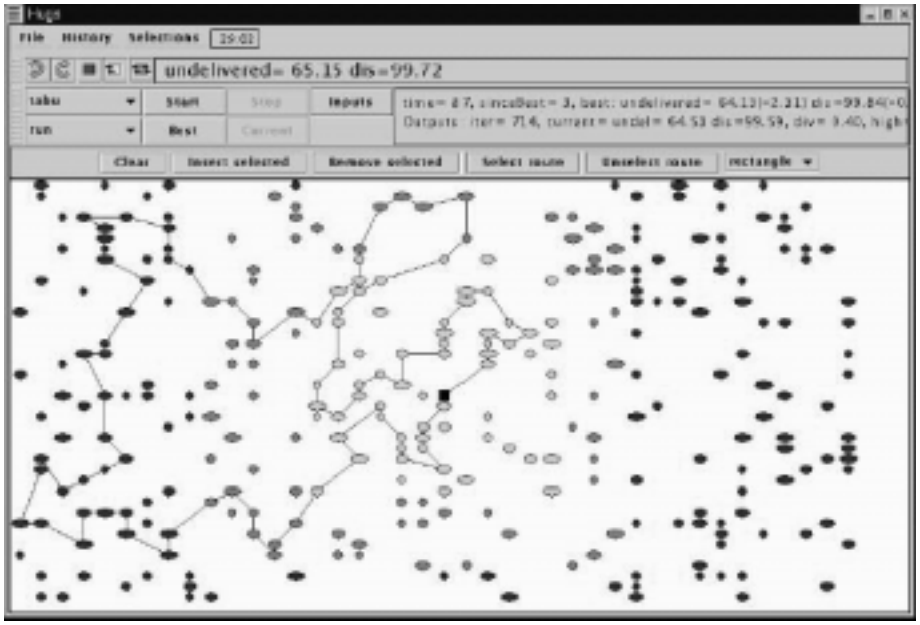


Abbildung 7: Delivery Applikation (Quelle: [4])

5 Die Paintshop Applikation

Die Paintshop Applikation stellt das Hauptthema dieser Arbeit dar. Sie benutzt das HuGS System, um das vorgestellte Reihenfolgeplanungsproblem für die Automobilindustrie zu lösen und nutzt die Vorteile der menschlichen Interaktion aus, um eine möglichst plausible Lösung zu generieren.

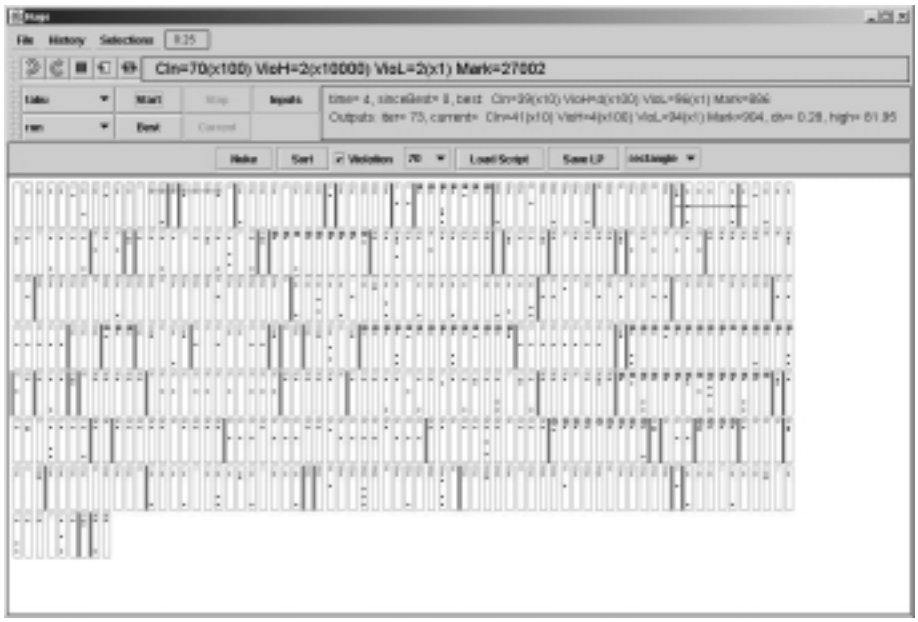


Abbildung 8: Paintshop Applikation

Im Weiteren wird beschrieben, wie in dieser Applikation die Struktur der Lösung, Nodes und Moves gewählt wurden, damit der Optimierungsprozess mit den im HuGS Core enthaltenen Suchalgorithmen möglichst gut funktioniert. Ebenfalls nicht irrelevant ist der Searchadjuster und der Movegenerator, die zusammen den Suchraum sinnvoll verkleinern. Zum Schluss wird noch der Script Modus vorgestellt, der menschliche Interaktion durch Scripts simuliert. Das ist notwendig, da für die „ROADEF Challenge 2005“ nur automatische Optimierungsprogramme erlaubt sind.

5.1 Funktionsweise

Die grundsätzliche Funktionsweise von HuGS ist immer gleich. Sie erzeugt eine Initiallösung und eine Move List mit einer Anzahl an Moves, die erlaubt sind, lässt den ausgewählten Suchalgorithmus die Liste durchiterieren und verwertet je nach Algorithmus die neue Lösung.
Als Beispiel betrachten wir die Vorgehensweise von Tabu Search anhand eines kurzen vereinfachten Pseudocodes:

```
s = Initiallösung
loop
   generiere Move List L mit allen Moves, die erlaubt sind
   best = s
   solange L nicht leer
      m = nächster Move in L und entferne diesen daraus
      neu = neue Lösung, wenn m auf s angewandt wird
      wenn neu besser als best, dann best = neu
   end solange
   s = best
   aktualisiere die Mobilities
end loop
```

Wenn der Suchalgorithmus auf Greedy eingestellt wäre, würde die innere „solange“ Schleife einfach terminiert, wenn eine neue bessere Lösung gefunden wurde. Die Zeile generiere Move List L beachtet die Mobilities der Nodes und die Parameter vom Searchadjuster, der später erläutert wird.

5.2 Wahl der Lösung

Bei dieser Aufgabe geht es im Grunde darum, eine Reihenfolge aller Autos zu finden, die in der Zielfunktion der Aufgabenstellung einen möglichst guten Wert erreicht. Eine Lösung besteht somit aus einer Permutation der n Fahrzeuge, die im Problem vorkommen.

5.3 Wahl der Nodes

Es ist sehr naheliegend, bei diesem Problem die Fahrzeuge als Node zu implementieren. In einem Node sind folgende Informationen gespeichert:

- Datum: Wie in der Aufgabenstellung schon beschrieben, enthält eine Probleminstanz immer die Fahrzeuge des Produktionstags D, aber auch einige der letzten Fahrzeuge aus dem Vortag $D-1$, damit am Anfang der Farbwechsel und die Verhältnisconstraints berechnet werden können. Je länger der längste Verhältnisconstraint ist, desto mehr Fahrzeuge aus $D-1$ müssen angegeben sein.

- Initialreihenfolge: Die Fahrzeuge einer Probleminstanz sind nicht irgendwie geordnet, sondern die Reihenfolge stellt schon die momentane Lösung bei Renault dar. Deshalb haben alle eingelesenen Fahrzeuge einen Index, der angibt, wo sie sich in der Initialreihenfolge befinden.

- Identifikation: Das ist einfach die ID des Fahrzeugs. Sie spielt in der Optimierungsphase keine Rolle, sondern wird nur bei der Ausgabe der Lösung angegeben, damit die Fahrzeuge zuordenbar sind.

- Farbe: Nur fortlaufende Zahlen sind für die Farben angegeben. Es ist auch praktisch egal, ob der wirkliche Farbwert angegeben ist, oder nur eine Zahl. Wichtig ist es nur zu wissen, ob sich zwei Fahrzeuge in ihren Farben unterscheiden oder nicht.

- Constraints: In dieser interessanten Sektion eines Nodes wird angegeben, ob ein gewisses Fahrzeug eine Komponente am Fließband benötigt oder nicht. Das ganze wird in einem großen Bitvektor gespeichert.

- Hashwert: Für die Performance wird noch ein zusätzlicher Hash Wert gespeichert, der die Fahrzeuge eindeutig kennzeichnet. Dieser Wert ist bei zwei Nodes genau dann gleich, wenn sowohl die Farbe als auch die Constraints übereinstimmen. Wenn in der Optimierungsphase ein Move zwei Nodes vertauschen möchte, wird immer zuerst überprüft, ob sie die gleichen Hash Werte besitzen. Ist das der Fall, so wird dieser Move einfach weggelassen, da es zu keiner Änderung in der Lösung führt.

5.4 Wahl der Moves

Die Wahl der Moves ist, wie schon im allgemeinen HuGS Kapitel erwähnt, ein sehr wichtiger Punkt bei der Implementierung einer Applikation für HuGS. Ob die Suchalgorithmen gut oder schlecht funktionieren hängt in erster Linie von der Wahl der Moves ab. Die Paintshop Applikation verfügt über folgende Moves:

5.4.1 Swap Move

Dies ist der einfachste Move, der auch intuitiv sofort einfallen würde. Er vertauscht einfach die Positionen von zwei Nodes miteinander, wobei die Mobilities beider Nodes auf high gesetzt sein müssen. Es ist durchaus eine Sache des Geschmacks und der Implementierung, ob dieser Move auch erlaubt sein soll, falls ein Mobility auf high und der andere auf medium ist. Beim vorliegenden Programm wurde jedoch entschieden, dass er strikt nur bei high erlaubt ist, da der Suchraum ohnehin schon extrem groß ist.
Die Stärke von Swap Move liegt darin, dass die Durchführung und die Auswertung seiner Änderungen sehr schnell geht (es können immer nur an zwei Positionen in einem kleinen lokalen Bereich die Verhältnisconstraintverletzungen und die Farbwechsel verändert werden). Wenn die Minimierung der Constraintverletzungen im Vordergrund steht, dann ist Swap Move eigentlich die effizienteste Variante unter allen Moves.
Der Nachteil bei einem Swap Move liegt auf der Hand: Wenn die Priorität der Farbänderungsminimierung sehr hoch ist, dann führt dieser Move oft zu einer Verschlechterung der Lösung, da ein Block von gleichfarbigen Nodes somit unterbrochen wird und zwei Farbwechsel dazukommen.

Beispiel
In der Ausgangslösung besitzen die Knoten 1, 2, 3, 4 und 7 dieselben Farben. Es gibt insgesamt zwei Farbwechsel in der Sequenz, nämlich zwischen 4 und 5, und zwischen 6 und 7. (Abb. 9)

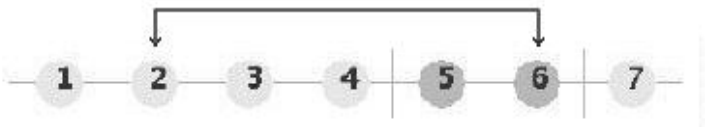


Abbildung 9: Ausgangslösung

Nun vertauschen wir die beiden Knoten 2 und 6 und die resultierende Lösung hat vier Farbwechsel, da die lange gleichfarbige Sequenz von 1 bis 4 unterbrochen wurde. (Abb. 10)

Wenn die Priorität der Farbänderungsminimierung gering ist, so ist es nicht sehr störend, wenn zusätzlich ein paar Farbänderungen dazukommen, wenn Verhältnisconstraints Verletzungen dafür eliminiert werden.

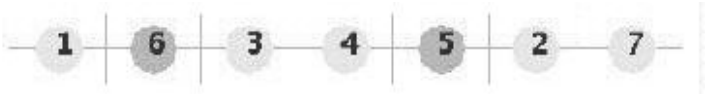


Abbildung 10: Lösung nach dem Swap Move

5.4.2 Block Move

Der Block Move wurde eingeführt, um die Schwachstelle des Swap Moves auszubessern. Ein Block Move verschiebt immer einen Block von aufeinanderfolgenden Nodes an eine Position zwischen zwei anderen Blöcken. Was ist nun ein Block?
Wenn wir etwas über die Nachteile des Swap Moves nachdenken, dann kommen wir auf die Idee, gleichfarbige Nodes zu einem Block zusammenzufassen. Er ist einfach eine beliebig lange Sequenz von aufeinanderfolgenden gleichfarbigen Nodes, sofern die maximale Farbblockgrenze nicht überschritten wird.
Ein Block selbst ist für den Block Move eine atomare Einheit, die nicht zerstört werden kann. Bei einem Block Move verschieben wir einen Block von gleichfarbigen Nodes auf eine Position, wo keine Blöcke unterbrochen werden (d.h. zwischen zwei Blöcke, auf die Anfangsposition oder auf die Endposition der Sequenz). Dadurch kann ein Block Move die Farbwechsel-Anzahl in einer Lösung nie verschlechtern. Ein Block Move ist nur ausführbar, wenn die Mobility aller Nodes eines Blocks auf high gesetzt ist.
Bemerkung: Ein Block Move verschiebt einen Block an eine Position, vertauscht aber keine zwei Blöcke miteinander. Eine Vertauschung von zwei Blöcken kann nur als zwei getrennte Block Moves realisiert werden.

Beispiel
In der Ausgangslösung existieren drei Blöcke: Knoten 1 bis 4, 5 bis 6 und 7 bilden jeweils einen Block. Es gibt wie im vorigen Beispiel zwei Farbänderungen. (Abb. 11)

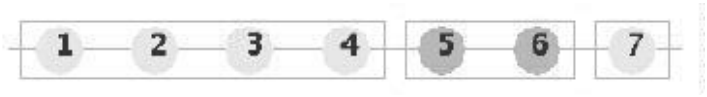


Abbildung 11: Ausgangslösung

Nun schieben wir Block 1 zwischen Block 2 und Block 3. (oder wir verschieben Block 2 auf die Anfangsposition. Es ist abhängig von der Betrachtungsweise). Als Resultat erhalten wir eine Lösung mit einem Farbwechsel und die Blöcke 1 und 3 aus Abbildung 11 werden sofort zusammengeschmolzen. (Abb. 12)

Abbildung 12: Lösung nach dem Block Move

Wenn die Priorität der Farbänderungsminimierung sehr hoch ist, erreicht dieser Typ von Moves sehr schnell einen guten Wert in der Zielfunktion, da er äußerst schnell die Anzahl der Farbänderungen auf das Minimum bringt. Der Nachteil an Block Moves besteht darin, dass nicht der gesamte Lösungsraum durchforscht wird. Sie verkleinern sogar den aktiven Suchraum immer weiter, da Blöcke zusammengelegt werden. Um auf ein globales Optimum kommen zu können, müssen zumindest Block Moves mit Swap Moves kombiniert werden.

5.4.3 Nuke Move

Der Nuke Move dient im Grunde nur dazu, aus einem lokalen Optimum wieder herauszuspringen. Dabei ordnet er die Reihenfolge aller Nodes in einem Betrachtungsraum zufällig um.
Es gibt zwei Varianten von Nuke Moves:

1. Er tauscht nur Nodes gegen andere aus, die dieselbe Farbe besitzen. Diese Variante könnte eingesetzt werden, wenn die Anzahl der Farbwechsel zufriedenstellend ist und der Benutzer nur die Verhältnisconstraints Verletzungen umordnen möchte.

2. Er ignoriert die Farben und vertauscht beliebige Nodes. D.h. alle betroffenen Nodes werden randomisiert.

Nuke Moves sind nur manuell ausführbar, oder können durch den Script Mode (auf den später eingegangen wird) ausgelöst werden. Dies haben Testläufe ergeben, wo Nuke Moves ab und zu automatisch ausgelöst wurden. Nach einem solchen Move verschlechtert sich in der Regel die Lösung drastisch,

sodass sie auch nicht nach wenigen Iterationen auf den ursprünglichen Wert kommen kann. Somit bringen diese Moves im automatisierten Betrieb leider sehr wenig, da dem Computer die Intuition fehlt, wann dieser Move sinnvoll eingesetzt werden soll.

5.4.4 Sort Move

Bei einer Instanz, wo die Farbänderungsminimierung in Vordergrund steht, könnte es ebenso sinnvoll sein, die Nodes von vornherein nach ihren Farben zu sortieren. Das wäre ein einziger Move, der die Anzahl der Farbänderungen auf ein Minimum bringen würde. Bei Block Moves würde es länger dauern. Andererseits könnte eine so schnelle Konvergenz dazu führen, dass das globale Optimum schwerer erreicht wird. Genau aus diesem Grund ist dieser Move ebenfalls nur manuell oder durch den Script Mode auslösbar. In der Regel ist es in einem automatischen Betrieb besser, den Computer Block Moves benutzen zu lassen, um die Anzahl der Farbänderungen zu senken.
Technisch gesehen wurde der Sort Move mit einem Insertion Sort implementiert, der grob gesagt nach dem Farbwert sortiert. Insertion Sort ist bei vorsortierten Datensätzen, die bei annähernd optimierten Autosequenzen auch gegeben sind, sehr schnell, deshalb wurde dieser ausgewählt.
Auf eine Sache musste noch geachtet werden, nämlich auf die maximale Farbblockgrenze. Es reicht nicht, einfach alle Nodes blind nach ihrem Farbwert zu sortieren. Angenommen die maximale Farbblockgrenze wäre zehn und die Sequenz, die mit Sort Move sortiert werden soll, enthält zwölf Nodes mit derselben Farbe, dann würden am Ende nach einem einfachen Sortiervorgang alle zwölf Nodes zusammen liegen und somit die maximale Farbblockgrenze überschritten (das würde die Lösung ungültig machen, da die maximale Farbblockgrenze auf jeden Fall eingehalten werden muss).
Es ist daher notwendig, die Farbwerte vor dem Sortieren so zu alternieren, dass danach garantiert eine gültige Lösung entsteht. Angenommen die maximale Farbblockgrenze wäre s, dann würden die ersten s Nodes einer bestimmten Farbe in der zu sortierenden Sequenz ihren Farbwert f beibehalten. Der Farbwert der nächsten s Nodes mit derselben Farbe muss dann auf $f + H$ geändert werden, wobei H eine ausreichend große Zahl ist, sodass $f + H$ einen nicht vorkommenden Farbwert darstellt. Für H kann einfach die Anzahl der verschiedenen Farben im Problem gewählt werden. Die nächsten s Nodes bekommen dann die Farbe $f + 2H$, usw.
Dadurch wird garantiert, dass zwischen den Blöcken der gleichfarbigen Nodes immer mindestens ein Node mit anderer Farbe steht. Falls zwischen den Nodes mit der Farbe $f + iH$ und $f + (i + 1)H$ keine andere Farbe vorkommt,

so müssen andere Nodes umgeschichtet werden, um die Lücke zu füllen. Dies geschieht auch durch Farbwertänderung.

5.5 Die Rolle des Searchadjusters und des Movegenerators

Der Searchadjuster hat die Funktion, den Suchraum zu beeinflussen. Dies ist in dieser Applikation besonders wichtig, da sowohl Lösungsraum als auch Suchraum pro Iteration extrem groß sind.
Der Lösungsraum besteht aus allen Permutationen der n Nodes im Problem, somit liegt seine Dimension in $O(n!)$.
Der Suchraum pro Iteration besteht aus allen möglichen Moves, die ausgeführt werden können. Wenn wir den Nuke Move und den Sort Move weglassen, die nur manuell aufrufbar sind, dann bleiben noch der Swap Move und der Block Move übrig.
Der Suchraum von Swap Move ist recht leicht zu erfassen. Wenn in der Lösung n Nodes vorkommen und jeder mit jedem vertauscht werden kann, dann liegt die Dimension in $O(n^2)$.
Der Suchraum von Block Move ist hingegen lösungsabhängig, da die Anzahl der Blöcke sehr unterschiedlich sein kann. Bei einem Problem, wo die Minimierung der Farbwechsel im Hintergrund steht, kann die Anzahl der Farbblöcke durchaus sehr hoch sein. Wenn b die Anzahl der Farbblöcke ist, dann liegt der Suchraum von Block Move in $O(b^2)$, da jeder Block auf eine Position zwischen zwei Blöcken wandern kann.
In Summe ergibt das die Dimension $O(n^2 + b^2)$, wobei $b \in O(n)$ gilt.

Als Vergleichswert betrachten wir das vorher schon erwähnte Vier-Positionen-Labeling-Problem: Dort hat jeder Node fünf Zustände (der fünfte Zustand ist der unplazierte Zustand), was einen Lösungsraum von $O(5^n)$ ergibt, wenn die Anzahl der Labels n beträgt. Der einzige Move, der dort implementiert ist, besteht darin, ein Label auf eine andere Position umzuplazieren. Der Suchraum pro Iteration ist dadurch auf $O(5n)$ beschränkt, was sogar linear ist.

Aus diesem Grund muss der Suchraum sinnvoll eingeschränkt werden, wenn der Tabu Algorithmus bei einer Instanz von über 1000 Autos noch in vernünftiger Zeit ablaufen soll, da durch die interne Struktur von HuGS der Zeitaufwand pro Move nicht klein ist.
Der Searchadjuster zur Paintshop Applikation enthält folgende zusätzliche Einträge:

- Swaps: gibt an, ob Swap Moves überhaupt ausgeführt werden sollen.

- Swap Size: gibt an, wie viele Swap Moves pro Iteration ausprobiert werden.
- Blocks: gibt an, ob Block Moves überhaupt ausgeführt werden sollen.
- Block Size: gibt an, wie viele Block Moves pro Iteration ausprobiert werden.

Der Movegenerator, der im Normalfall pro Iteration alle möglichen Moves zu einer Move List aufbaut, die dann vom Suchalgorithmus durchgegangen wird, bezieht hier die Daten aus dem Searchadjuster und baut höchstens so viele Moves auf, wie sie durch diese Parameter angegeben sind.
Wenn z.B. die Swap Size auf 1000 eingestellt ist, gibt der Movegenerator nur 1000 zufällige Swap Moves des gesamten Suchraums in die Move List.
Mit Hilfe des Searchadjuster und des Movegenerators schaffen die Suchalgorithmen in einem Zeitinterval immer eine annähernd konstante Anzahl von Suchiterationen, egal wie groß die Probleminstanzen sind. Es ist natürlich immer noch von der Anzahl und von der Länge der Verhältnisconstraints abhängig, wie schnell ein Move durchgeführt werden kann.

5.6 Script Modus

Da bei der „ROADEF Challenge 2005“ menschliche Interaktion in der Optimierungsphase nicht vorgesehen ist, muss ein Script Modus einen Teil dieser Aufgabe übernehmen. Er hat zwei Aufgaben:

- Aufzeichnung von Aktionen, die der Benutzer während einer Optimierungsphase ausführt, angelegen und dann in einer Script Datei speichern.
- Die Script Datei laden und die aufgezeichneten Aktionen ausführen.

Damit die Aufzeichnung überhaupt funktionieren kann, ist das gesamte Script Konzept in die HuGS Visualization integriert. Die Methoden, die durch menschliche Interaktion aufgerufen werden, zeichnen die aufgerufenen Aktionen auf. Dabei wird eine Logger Klasse angelegt, die die Aufzeichnungen mit einem Timestamp ausstattet. Beim Speichern der Script Kommandos schreibt die Logger Klasse all ihre Inhalte in eine Datei.
Wenn eine Script Datei geladen wird, legt die Einleseroutine pro Kommando ein Aktionsobjekt an. Jedes Aktionsobjekt verfügt über folgende Daten:

- Timestamp: enthält die Zeitangabe in Sekunden nach Programmstart, wann die Aktion ausgelöst werden soll.

- Typ: stellt eine Konstante dar, die besagt, von welchem Typ diese Aktion ist. Typen sind z.B. Markierung, Aktivierung eines Moves, Prozessterminierung, etc.
- Parameter: Je nach Aktionstyp sind einige oder gar keine Parameter erforderlich. Bei einer Markierung muss z.B. immer angegeben sein, von wo bis wo die Nodes markiert werden sollen. Bei der Prozessterminierung sind gar keine Parameter notwendig.
- Pointer auf das nächste Aktionsobjekt: Um den Speicheraufwand effizient zu halten, werden nur so viele Objekte wie nötig erzeugt und jedes Objekt wird mit dem nächsten verkettet. Beim Durchgehen der Objektkette kann immer nur das erste Objekt eine Aktion auslösen. D.h. Objekt O_i vor Objekt $O_j \Rightarrow O_i$.Timestamp $\leq O_j$.Timestamp.

In einem Thread, der jede Sekunde ein Mal ausgeführt wird, vergleicht der Script Modus die momentane Zeit mit dem Timestamp des ersten Aktionobjekts. Sind diese beide gleich, wird die Aktion aufgerufen, die im Objekt angegeben ist. Dabei werden genau die Methoden in der Visualisierung ausgeführt, die den Eintrag im Logger verursacht haben.
Es ist übrigens nicht notwendig, die Script Datei mittels Logger abzuspeichern. Der Benutzer kann diese auch direkt editieren. Der Script Modus versteht folgende Kommandos:

- `start`: Startet den Optimierungsprozess.
- `stop`: Beendet den Optimierungsprozess.
- `mark <anfang> <ende>`: Markiert alle Nodes zwischen `<anfang>` und `<ende>`.
- `clearmark`: Hebt die Markierung wieder auf.
- `setmobility <wert>`: Setzt die Mobility der markierten Nodes auf `<wert>`. Dabei ist `<wert>` eine Zahl aus 0, 1 und 2, wobei 0 low, 1 medium und 2 high bedeutet.
- `swapmove <pos1> <pos2>`: Führt den Swap Move zwischen den Nodes an de Position `<pos1>` und `<pos2>` aus.
- `blockmove <anfang> <ende> <zielpos>`: Führt den Block Move für den Block aus, der alle Nodes zwischen `<anfang>` und `<ende>` nach Position `<zielpos>` verschiebt.

- `sortmove <anfang> <ende>`: Führt den Sort Move aus, der alle Nodes zwischen `<anfang>` und `<ende>` nach ihrer Farbe sortiert.
- `nukemove`: Führt die erste Variante des Nuke Moves aus, der alle Nodes zwischen `<anfang>` und `<ende>` beliebig vertauscht, wobei die farbliche Anordnung nicht zerstört wird.
- `randomize <anfang> <ende>`: Führt die zweite Variante des Nuke Moves aus, der alle Nodes zwischen `<anfang>` und `<ende>` beliebig vertauscht, wobei die farbliche Anordnung zerstört wird.
- `set swaps <wert>`: Setzt den Parameter im Searchadjuster für Swap Moves auf `<wert>`. Ist `<wert>` gleich 0, so werden überhaupt keine Swap Moves ausgeführt. Ansonsten führt der Suchalgorithmus pro Iteration immer `<wert>` Swap Moves aus.
- `set blocks <wert>`: Setzt den Parameter im Searchadjuster für Block Moves auf `<wert>`. Ist `<wert>` gleich 0, so werden überhaupt keine Block Moves ausgeführt. Ansonsten führt der Suchalgorithmus pro Iteration immer `<wert>` Block Moves aus.
- `save <datei>`: Speichert die momentane Lösung in der Visualisierung in `<datei>` ab.
- `exit`: Terminiert die Paintshop Applikation.

Ein Kommando muss immer folgende Form besitzen:
`<Time> <Command> [Param1] [Param2]...`

6 Resultate

Bei der „ROADEF Challenge 2005“ wurde in der ersten Ausscheidungsrunde das TestSet A veröffentlicht. Die meisten Instanzen von dieser Set beinhalten ca. 400 bis 1300 ein Fahrzeuge, die wiederum über bis zu 22 Komponenten verfügen. Hier die detaillierten Daten:

Instanz	#Fahrz.	#Komp.	C_{CC}	C_{HPC}	C_{LPC}
022_3_4_ep_raf_enp	499	9	100	10000	1
022_3_4_raf_ep_enp	499	9	10000	100	1
024_38_3_ep_enp_raf	1274	13	1	10000	100
024_38_3_ep_raf_enp	1274	13	100	10000	1
025_38_5_ep_enp_raf	1329	13	1	10000	100
025_38_5_ep_raf_enp	1329	13	100	10000	1
025_38_1_ep_enp_raf	1233	22	1	10000	100
025_38_1_ep_raf_enp	1233	22	100	10000	1
039_38_4_ep_raf_ch1	981	5	100	10000	-
039_38_4_raf_ep_ch1	981	5	10000	100	-
048_39_1_ep_enp_raf	618	19	1	10000	100
048_39_1_ep_raf_enp	618	19	100	10000	1
064_38_2_ep_raf_enp_ch1	904	9	100	10000	1
064_38_2_raf_ep_enp_ch1	904	9	10000	100	1
064_38_2_ep_raf_enp_ch2	434	6	100	10000	1
064_38_2_raf_ep_enp_ch2	434	6	10000	100	1

Tabelle 1: Instanzen

C_{CC} ... Kosten für eine Sprühkopfreinigung bei Farbwechsel
C_{HPC} ... Kosten für eine Verletzung eines Constraints mit hoher Priorität
C_{LPC} ... Kosten für eine Verletzung eines Constraints mit niedriger Priorität

Die Laufzeit des Algorithmus der Challenge beträgt 600 Sekunden. Da es sich hierbei um ein heuristisches und nichtdeterministisches Programm handelt, werden fünf Durchläufe pro Instanz gemacht und dann der Mittelwert aufgezeichnet.

Die folgende Tabelle stellt den erreichten Score dem alten Score aus dem im Moment bei Renault eingesetzten Programm gegenüber: (niedrigerer Score

ist wegen der Minimierungszielfunktion besser)

Instanz	alter Score	neuer Score
022_3_4_ep_raf_enp	27002	4682
022_3_4_raf_ep_enp	114805	114523
024_38_3_ep_enp_raf	828164	679410
024_38_3_ep_raf_enp	766505	642430
025_38_5_ep_enp_raf	1073892	914294
025_38_5_ep_raf_enp	1026900	926950
025_38_1_ep_enp_raf	207790	60008
025_38_1_ep_raf_enp	31075	29957
039_38_4_ep_raf_ch1	1182900	1017480
039_38_4_raf_ep_ch1	729300	709920
048_39_1_ep_enp_raf	430234	338375
048_39_1_ep_raf_enp	369062	327878
064_38_2_ep_raf_enp_ch1	40287	16834
064_38_2_raf_ep_enp_ch1	687043	685047
064_38_2_ep_raf_enp_ch2	284650	4035
064_38_2_raf_ep_enp_ch2	319761	306758

Tabelle 2: Resultate

Alle neuen Werte sind Durschnittsergebnisse von fünf Durchläufe zu je 600 Sekunden auf einem Rechner mit folgender Spezifikation:

AMD Athlon XP 1600+ (1400 MHz)
512 MB RAM
Windows XP Professional SP1

Bei den größeren Instanzen mit vielen Fahrzeugen und Komponenten wie z.B. „025_38_1_ep_enp_raf“ lässt sich erkennen, dass das Java Programm in der Laufzeit leider etwas benachteiligt ist. 600 Sekunden sind von der Seite des Algorithmus aus unter Umständen nicht mehr ausreichend, um an die Grenzen des Machbaren kommen.
Die Situation wird durch die GUI sogar noch geringfügig verschlechtert, da sie auch eine gewisse Rechenleistung in Anspruch nimmt. Das ist jedoch nicht vermeidbar, da die Visualisierung eng mit der menschlichen Interaktion und damit auch mit dem Script Modus gekoppelt ist.

Als Vergleich dienen folgende zwei Skizzen, die das Laufzeitverhalten bei einer kleinen Instanz „022_3_4_ep_raf_enp“ und bei einer weitaus größeren Instanz „025_38_1_ep_enp_raf“ charakterisieren:

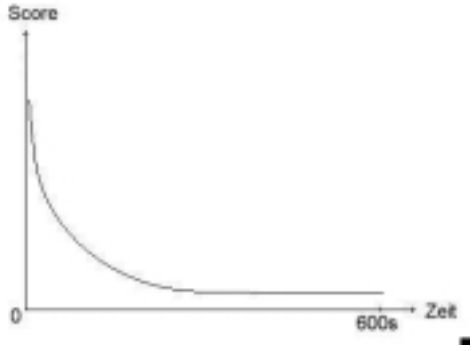


Abbildung 13: 022_3_4_ep_raf_enp

Bei „022_3_4_ep_raf_enp“ merken wir deutlich, wie rasch der Score gleich von Anfang an besser wird und schon nach Halbzeit konvergiert. Hier hätte eine Laufzeiteinschränkung auf 300 Sekunden keinen wesentlichen Einfluss auf das Ergebnis.

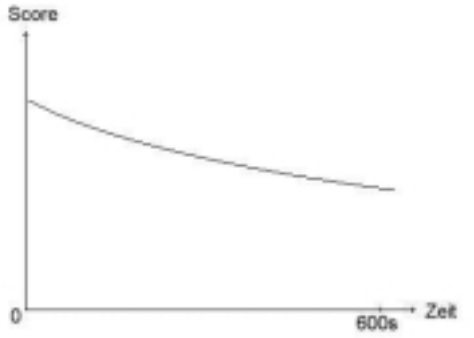


Abbildung 14: 025_38_1_ep_enp_raf

Hier bei „025_38_1_ep_enp_raf“ konvergiert das Ergebnis äußerst langsam. Selbst bei 600 Sekunden werden immer noch viele Verbesserungen gemacht.

Bei doppelter Laufzeit könnte der Score wesentlich besser ausfallen.

6.1 Die Skripten

Für jede Instanz wurde ein eigener Skript generiert, um die menschliche Interaktion zu simulieren. Dabei handelt es sich um eine Auflistung von Befehlen, die zu einem bestimmten Zeitpunkt aktiviert werden.

Beispiel

```
1 set swaps 100
1 set blocks 0
90 set swaps 0
90 set blocks 50
180 set swaps 500
180 set blocks 0
240 mark 228 1232
240 setmobility 0
241 mark 228 729
241 setmobility 2
300 mark 228 1232
300 setmobility 0
301 mark 730 1232
301 setmobility 2
360 mark 228 1232
360 setmobility 2
420 set swaps 0
420 set blocks 100
500 set swaps 1000
500 set blocks 0
590 stop
592 save 025_38_1_EP_RAF_ENP
595 exit
```

Hier schränkt das Skript beim Start die Suche auf 100 Swap Moves pro Iteration ein. Nach 90 Sekunden schaltet es die Swap Moves aus und setzt die Anzahl der Block Moves pro Iteration auf 50. Nach 180 Sekunden wird das Ganze wieder rückgängig gemacht.
Von der 240-ten Sekunde bis zur 360-ten Sekunde versucht das Skript, durch

das Umsetzen der Mobilities den Suchraum einzuschränken. Dabei werden immer die Nodes mittels `mark` markiert und dann per `setmobility` auf den gewünschten Wert gesetzt.
Ab der 420-ten Sekunde werden die Movesorten noch einige Male umgesetzt. In der 590-ten Sekunde hält die Suche an, danach wird das Ergebnis in die Datei „025_38_1_EP_RAF_ENP“ gespeichert und das Programm terminiert.

Das ist ein repräsentatives Beispiel einer Skript Datei. Durch die Abwechselung von Sort Moves und Block Moves ist es möglich, die im Kapitel 5.4 erwähnten Nachteile beider Moves auszugleichen. Die Einschränkung der Mobilities auf low bei der Hälfte der Nodes verursacht eine verstärkte Suche in der anderen Hälfe des Instanzes.
Alle anderen Skript Datein sind nach diesen Prinzipien aufgebaut. Je nach Optimierungspriorität wird die Laufzeit unterschiedlich zwischen Swap Moves und Block Moves aufgeteilt. Wenn die Gewichtung der Verhältnisconstraintsverletzungen groß ist, werden mehr Swap Moves eingesetzt. Wenn die Bestrafung der Farbwechsel groß ist, werden mehr Block Moves eingesetzt. Sort- und Nuke Moves kommen in den Skripten nicht vor, da diese Moves gezielt eingesetzt werden sollen und der Fortschritt der Suche nicht im Vorraus abgeschätzt werden kann. Bei kleineren Instanzen wird der mittlere Teil weggelassen, wo die Mobilities eingeschränkt werden, da es in diesem Fall nicht notwendig ist, den Suchraum zu verkleinern.

6.2 Vergleiche

Wir untersuchen im Folge die Unterschiede zwischen den verschiedenen Vorgangsweisen. Dabei handelt es sich um den Vergleich zwischen Optimierung mit und ohne menschlicher Interaktion und zwischen den Suchalgorithmen Greedy und Tabu.

Guided vs. Unguided
In Abbildung 15 sehen wir anhand eines Beispiels den Unterschied zwischen guided und unguided Tabu Search. Bei diesem Beispiel handelt es sich um „022_3_4_ep_raf_enp“, einer mittelgroßen Instanz. Es ist leicht zu sehen, dass die unguided Version immer wieder in einem längeren Zeitraum keine Verbesserungen verursacht. Dies liegt daran, dass er oft in einem lokalen Optimum festsitzt. Die guided Version besitzt diesen Nachteil nicht, da der Benutzer das Festsitzen sofort bemerken und darauf reagieren kann (z.B. mittels einer Änderung der Suchstrategie, der Movesorte oder der Mobilities).

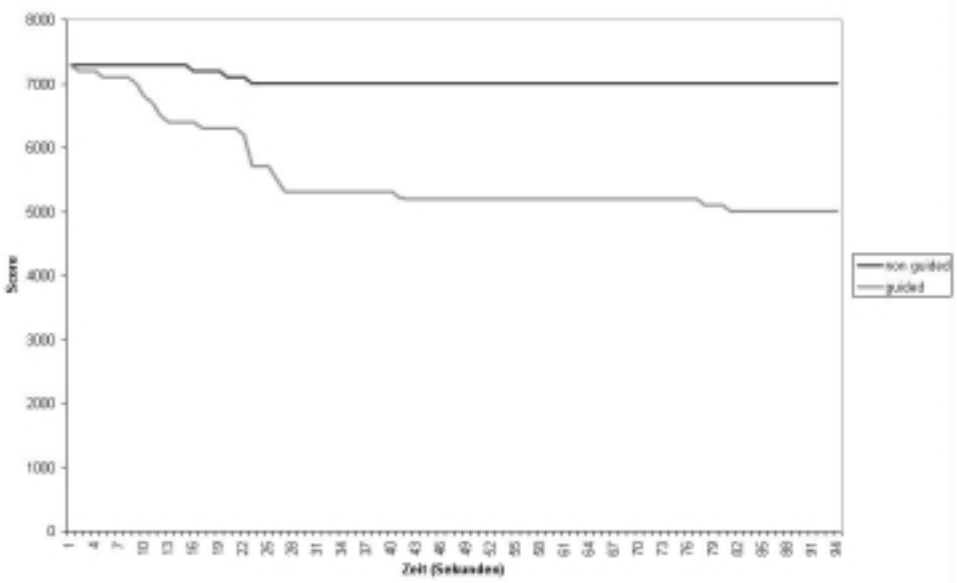


Abbildung 15: Guided vs. Unguided

Tabu vs. Greedy
Die Abbildung 16 zeigt uns anhand der selben Probleminstanz den Unter-

schied zwischen den Suchalgorithmen Tabu und Greedy. Wegen der raschen Geschwindigkeit von Greedy verbessert sich die Lösung in der Anfangsphase schneller als Tabu. Aber genau deswegen konvergiert sie zu schnell, d.h. in der späteren Phase bleibt Greedy leichter in lokan Optimen hängen. Da er kaum Verschlechterungen in Kauf nimmt, kommt er nur durch die Erhöhung des Ply-Werts wieder heraus. Das kostet auch dementsprechend Zeit. Tabu hat den Vorteil, dass er mehr Moves pro Iteration durchrechnet und davon den vielversprechendsten ausführt. Durch die aufgezeichnete Tabu-Liste lassen sich langzeitige Aufenthalte in lokalen Optimen vermeiden.

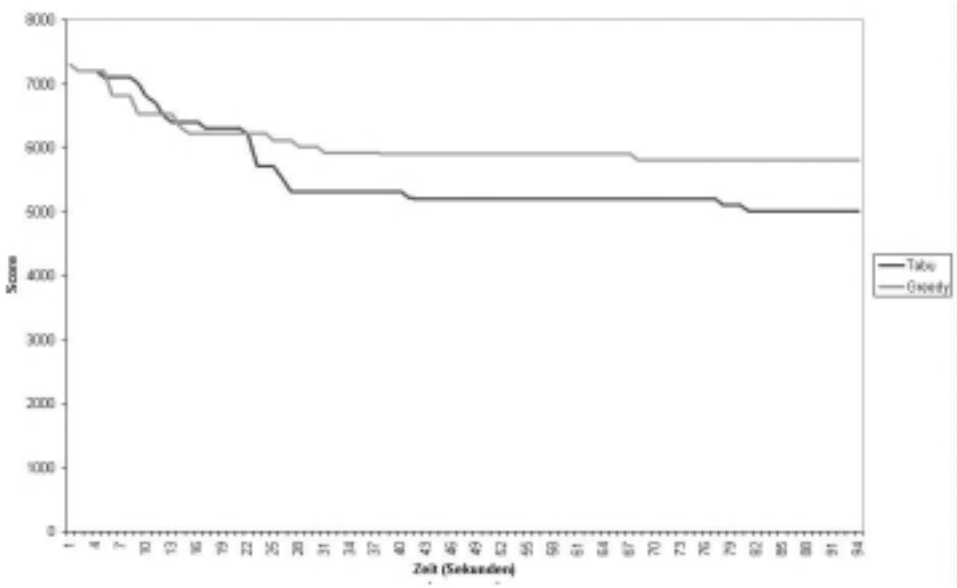


Abbildung 16: Tabu vs. Greedy

Charakteristiken von Große Instanzen
Bei großen Instanzen wie z.B. „025_38_1_ep_enp_raf“ in Abbildung 17 ist die Wahl der Suchalgorithmen weniger kritisch. Überraschenderweise schneidet hier Greedy in der Anfangsphase am schlechtesten ab. Dies liegt vor allem an der Größe der Instanz. Es existieren von Anfang an sehr viele Moves, die die Lösung verbessern können. Greedy wählt immer den ersten davon aus, egal um wieviel die Lösung dadurch verbessert wird. Tabu analysiert hingegen die ersten n Moves und führt davon den besten aus, was die rasche Verbesserung der Lösung erklärt. Hier ist der Unterschied zwischen guided und unguided Tabu nicht groß. Da der Algorithmus fast nie in lokalen Optimen festsitzt,

muss die menschliche Interaktion auch nicht eingreifen.

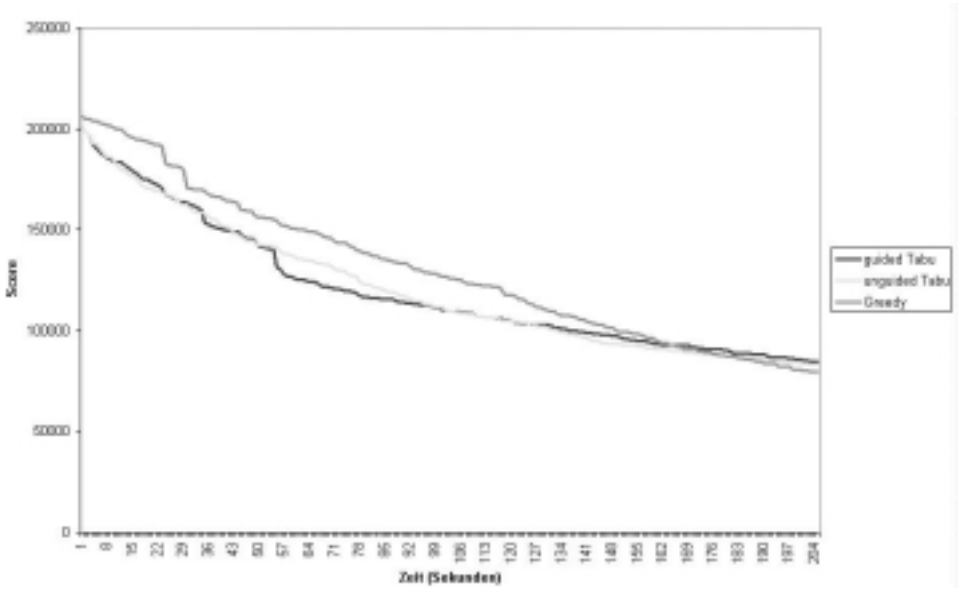


Abbildung 17: Große Instanzen

7 Exakter Lösungsansatz

In diesem Kapitel wird versucht, das Problem von der Richtung zu betrachten, wie man es optimal lösen könnte. Dazu führen wir zuerst ein paar Überlegungen an, die reduzierte Teilaufgaben betrachten. Des weiteren wird das Problem in ein ganzzahliges lineares Programm (ILP) gebracht, sodass es von Programmen, die ILPs exakt lösen können (z.B. CPLEX), akzeptiert wird.

7.1 Reduzierte Teilprobleme

Farbwechselminimierung Als erstens überlegen wir uns, was passieren würde, wenn nur die Farbwechselminimierung betrachtet wird. D.h. alle Verhältnisconstraints werden vorläufig außer Acht gelassen. Das Ergebnis ist sehr trivial.

Lemma 7.1 *Wenn es nur um die Farbwechselminimierung geht, so stellt die Sortierung der Fahrzeuge nach ihrer Farbe unter Berücksichtigung der maximalen Farbblockgrenze die optimale Lösung dar.*
Besteht die Instanz aus der Multimenge $\{a, \ldots, a, b, \ldots, b, \ldots\}$*, wo jede Buchstabe eine Farbe darstellt, so teilen wir sie in einfärbige Partitionen* $A = \{a, \ldots, a\}$*,* $B = \{b, \ldots, b\}$*, ... auf.*
Jede dieser Partitionen teilen wir wieder in Partitionen A_1*, ...,* A_α*,* B_1*, ...,* B_β*, ... mit* $A = \bigcup_{i=1}^{\alpha} A_i$*,* $B = \bigcup_{i=1}^{\beta} B_i$*, ... auf, sodass jede Partition mit Ausnahme der jeweils letzten Partition einer Farbe genau* s *Elemente besitzt, wenn* s *die maximale Farbblockgrenze darstellt.* A_α*,* B_β*, ... können unterbesetzt sein.*
Nun ordnen wir alle Partitionen so um, dass keine zwei Partitionen mit derselben Farbe nebeneinander stehen. Ist dies nicht möglich, so ist das Problem wegen der maximalen Farbblockgrenze unlösbar. Sonst stellt diese Reihenfolge eine optimale Lösung dar.

Beweis Wenn eine bessere Lösung existieren würde, so müsste es einen Block Move geben, der zwei unterbesetzte Partitionen der gleichen Farbe zusammenfügen könnte, sodass eine gleichfärbige Partition entsteht. Das ist aber laut der Konstruktion unmöglich, da es pro Farbe nur eine unterbesetzte Partition geben kann.

Minimierung der Verhältnisconstraintverletzungen
Nun überlegen wir uns, was wir machen könnten, wenn nur ein Verhältnisconstraint gäbe. Die Farbwechselminimierung lassen wir außer Acht.

Lemma 7.2 *Die Gleichverteilung der Fahrzeuge mit einem Merkmal unter den Fahrzeugen ohne dem Merkmal stellt die Optimallösung des Problems mit einem Verhältnisconstraint dar.*
Besteht die Instanz aus n Fahrzeugen und besitzen k Fahrzeuge $x_i, i \in 1 \dots k$ davon das Merkmal, so stellt die Sequenz, wo alle x_i an den Positionen $\lceil (i-1) \cdot \frac{n}{k} \rceil$ stehen und die Zwischenräume mit Fahrzeugen ohne dem Merkmal aufgefüllt sind, eine optimale Lösung dar.

Beweis Fall 1: Wenn es keine Verletzung existiert, dann ist es eine Optimallösung.
Fall 2: Wenn es Verletzungen gibt, kann durch eine Umordnung kein besseres Ergebnis erzielt werden. Betrachten wir die Teilsequenz mit drei Fahrzeugen, die das Merkmal besitzten: $x_{i-1}, o, \dots, o, x_i, o, \dots, o, x_{i+1}$. o sind Fahrzeuge, die das Merkmal nicht besitzen und somit den Zwischenraum füllen. Um im Betrachtungsfenster $x_i, o, \dots, o, x_{i+1}$ die Anzahl der Verletzungen um l zu verringern, muss x_i um l Positionen nach vorne verschoben werden. Dadurch ergeben sich jedoch l zusätzliche Verletzungen von x_i mit x_{i-1} und somit bleibt die Gesamtanzahl der Verletzungen gleich.

7.2 Ganzzahliges lineares Programm (ILP)

Ein ganzzahliges lineares Programm ist einfach eine mathematische Formulierung eines Problems, wo es eine lineare Zielfunktion zu maximieren bzw. minimieren gilt, die etliche ganzzahlige Entscheidungsvariablen umfasst und alle linearen Nebenbedingungen erfüllen muss.

Beispiel
Zielfunktion: Maximiere $4a + b + 3c$
unter den Nebenbedingungen $a + b \leq 3$ und $b + 4c \leq 7$
Entscheidungsvariablen a, b und c sind nichtnegative ganze Zahlen.

Das wäre ein einfaches ganzzahliges lineares Problem, das mit der Simplex Methode gelöst werden kann. Die Zielfunktion und Nebenbedingungen entsprechen Funktionen in einem n-dimensionalen Raum, wobei n die Anzahl der Nebenbedingungen ist. Die Nebenbedingungen schränken den Lösungsraum des ILPs ein und die Optimallösung ist stets ein Eckpunkt dieses Lösungsraums. Simplex untersucht, vereinfacht dargestellt, ob eine Verbesserung der Lösung erzielt werden kann, indem er von einem Eckpunkt zu einem anderen Eckpunkt wandert.

Nun gehen wir Schritt für Schritt auf die Entstehung der ILP Formulierung ein. Auf folgende Terminologien müssen wir uns vorerst einigen:

- n ... Anzahl der Fahrzeuge
- β ... Anzahl der verschiedenen Komponenten
- f ... Anzahl der verschiedenen Farben
- s ... Maximale Farbblockgrenze: Nach dieser Anzahl an gleichfarbigen Fahrzeugen muss die Farbe gewechselt werden

Die Matrix B mit den Einträgen b_{ij} gibt an, welche Fahrzeuge welche Komponenten benötigen. B hat die Dimension $n \times \beta$.

$$b_{ij} = \begin{cases} 1 & \text{falls Fahrzeug } i \text{ die Komponente } j \text{ benötigt} \\ 0 & \text{sonst} \end{cases}$$

Dann gibt die Matrix C mit den Einträgen c_{ij} an, welche Farbe die Fahrzeuge besitzen. C hat die Dimension $n \times f$.

$$c_{ij} = \begin{cases} 1 & \text{falls Fahrzeug } i \text{ die Farbe } j \text{ besitzt} \\ 0 & \text{sonst} \end{cases}$$

Dann können wir die binären Entscheidungsvariablen x_{ij} mit $i, j \in 1 \dots n$ einführen, die besagen, welche Fahrzeuge an welchen Stellen untergebracht sind. Die x_{ij} bilden zusammen die Matrix X mit der Dimension $n \times n$.

$$x_{ij} = \begin{cases} 1 & \text{falls Fahrzeug } i \text{ an Stelle } j \text{ steht} \\ 0 & \text{sonst} \end{cases}$$

Da logischerweise jedes Fahrzeug nur an einer Position stehen kann, müssen die Spalten- und Zeilensummen in X gleich eins sein. Somit kommen wir auch schon zu unseren ersten Nebenbedingungen:

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j = 1 \dots n$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i = 1 \dots n$$

Des weiteren führen wir eine Hilfsmatrix P ein, die die Zählung der Verhältnisconstraintverletzungen erleichtern soll. Die Elemente von P sind p_{ij} mit $i \in 1 \dots \beta, j \in 0 \dots n$. Element p_{ij} besagt einfach, wie viele Fahrzeuge bis zur Position j die Komponente i erfordern. Das ist von essentieller Bedeutung bei der Zählung von Constraintverletzungen, da die Differenz $p_{ij} - p_{ik}$

für $j > k$ angibt, wie viele Fahrzeuge zwischen den Positionen j und k die Komponente i benötigen.
Ist bei der Komponente i der Verhältnisconstraint $\frac{Z}{N}$, so würde ein Wert $P_{ij} - P_{ij-N} = v$ mit $v > Z$ eine Constraintverletzung ergeben.
Auf folgende Art und Weise können wir die Werte p_{ij} berechnen:

$$p_{i0} = 0 \qquad \forall i = 1 \ldots \beta$$

$$p_{ik} = p_{ik-1} + \sum_{j=1}^{n} b_{ji} \cdot x_{jk} \qquad \forall i = 1 \ldots \beta, \forall k = 1 \ldots n$$

Nachdem wir die Matrix P haben, ist der Rest einfach. Nun müssen wir sie nur noch ein Mal durchgehen und schauen, ob wir im jeweiligen Betrachtungsfenster eine Verletzung finden. Die Verletzungen speichern wir in der Matrix V ab mit den Elementen v_{ij} ab.
v_{ij} können wir als $\max\{0, p_{ij} - p_{ij-N_i} - Z_i\}$ definieren. Da das LP System jedoch keinen solchen Ausdruck akzeptiert, wandeln wir sie in folgende zwei Ungleichungen um:

$$v_{ij} \geq 0 \qquad \forall i = 1 \ldots \beta, \forall j = N_i \ldots n$$

$$v_{ij} \geq p_{ij} - p_{ij-N_i} - Z_i \qquad \forall i = 1 \ldots \beta, \forall j = N_i \ldots n$$

Nun kommen wir zum Zählen der Farbwechsel. Da berechnen wir uns ebenfalls vorerst eine Hilfsmatrix Q mit den Elementen q_{ij}, die einfach die tatsächliche Reihenfolge der Fahrzeuge im Bezug zur Farbe wiedergibt:

$$q_{ik} = \sum_{j=1}^{n} b_{ji} \cdot x_{jk} \qquad \forall i = 1 \ldots f, \forall k = 1 \ldots n$$

Diese sieht sehr ähnlich wie die Matrix C aus, nur gibt sie nicht an, welche Farbe das Fahrzeug k besitzt, sondern welche Farbe das Fahrzeug an der Position k besitzt.
Wir müssen jedoch die maximale Farbblockgrenze beachten und die Möglichkeiten beim Aufbau der Matrix Q einschränken, indem wir verlangen, dass nur höchstens s aufeinanderfolgende „1“ in Q existieren können:

$$\sum_{j=0}^{s} q_{ik+j} \leq s \qquad \forall i = 1 \ldots f, \forall k = 1 \ldots n - s$$

Dann können wir die Farbwechsel einfach mit der Matrix W zählen, indem wir ihre Elemente w_{ij} mit $w_{ij} = \max\{0, q_{ij} - q_{ij-1}\}$ definieren. w_{ij} ist genau

dann eins, wenn ein Sprung von null auf eins in der Matrix Q existiert. Das Ganze bringen wir noch auf die LP verträgliche Form:

$$w_{ij} \geq 0 \qquad \forall i = 1 \ldots \beta, \forall j = 2 \ldots n$$

$$w_{ij} \geq q_{ij} - q_{ij-1} \qquad \forall i = 1 \ldots \beta, \forall j = 2 \ldots n$$

Da die Summe über alle Elemente in V die Anzahl der Constraintverletzungen und die Summe über alle Elemente in W die Anzahl der Farbwechsel darstellt, können wir die Zielfunktion so definieren:

$$\text{Minimiere: } \sum_{i=1}^{\beta} \sum_{j=N_i}^{n} v_{ij} + \sum_{i=1}^{f} \sum_{j=2}^{n} w_{ij}$$

7.3 Die ILP Formulierung

Nun kombinieren wir noch alle Überlegungen und fassen die Ergebnisse zu dieser Formulierung zusammen:

Zielfunktion: $\min \sum_{i=1}^{\beta} \sum_{j=N_i}^{n} v_{ij} + \sum_{i=1}^{f} \sum_{j=2}^{n} w_{ij}$

unter den Nebenbedingungen

1. $\sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j = 1 \ldots n$
2. $\sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i = 1 \ldots n$
3. $p_{i0} = 0 \qquad \forall i = 1 \ldots \beta$
4. $p_{ik} = p_{ik-1} + \sum_{j=1}^{n} b_{ji} \cdot x_{jk} \qquad \forall i = 1 \ldots \beta, \forall k = 1 \ldots n$
5. $v_{ij} \geq 0 \qquad \forall i = 1 \ldots \beta, \forall j = N_i \ldots n$
6. $v_{ij} \geq p_{ij} - p_{ij-N_i} - Z_i \qquad \forall i = 1 \ldots \beta, \forall j = N_i \ldots n$

7. $q_{ik} = \sum_{j=1}^{n} b_{ji} \cdot x_{jk}$ $\forall i = 1 \ldots f, \forall k = 1 \ldots n$

8. $\sum_{j=0}^{s} q_{ik+j} \leq s$ $\forall i = 1 \ldots f, \forall k = 1 \ldots n - s$

9. $w_{ij} \geq 0$ $\forall i = 1 \ldots \beta, \forall j = 2 \ldots n$

10. $w_{ij} \geq q_{ij} - q_{ij-1}$ $\forall i = 1 \ldots \beta, \forall j = 2 \ldots n$

7.4 Beispiel für die ILP-Formulierung

Anhand eines einfachen Beispiels lässt sich zeigen, wie die ILP-Formulierung funktioniert. Gegeben sei vier Fahrzeuge mit den folgenden Komponenten und Farbvektoren:

Komponenten
Fahrzeug1 : $\begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}$
Fahrzeug2 : $\begin{pmatrix} 1 & 1 & 0 & 0 \end{pmatrix}$
Fahrzeug3 : $\begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}$
Fahrzeug4 : $\begin{pmatrix} 1 & 1 & 0 & 1 \end{pmatrix}$

Farben
Fahrzeug1 : $\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$
Fahrzeug2 : $\begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$
Fahrzeug3 : $\begin{pmatrix} 0 & 1 & 0 \end{pmatrix}$
Fahrzeug4 : $\begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$

Reihenfolge
2, 4, 3, 1

Aus diesen Angaben lassen sich folgende Matrizen aufstellen.

$$X = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

$$C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Die Zeilen (3), (4) und (7) berechnen dann die Hilfsmatrizen P und Q.

$$P = \begin{pmatrix} 0 & 1 & 2 & 2 & 3 \\ 0 & 1 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 2 & 2 \end{pmatrix}$$

$$Q = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Die Matrizen V und W, die die Anzahl der Constraintverletzungen und Anzahl der Farbwechsel angeben, sehen dann so aus.

$$V = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$W = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Sie besagen laut Definition drei Constraintverletzungen und zwei Farbwechsel. Laut Zielfunktion erreicht diese Anordnung einen Wert von $3 + 2 = 5$.

Natürlich muss in der Praxis die Gewichtung der Optimierungsziele berücksichtigt werden. Ebenso müssen die Verletzungen von Constraints mit hoher und niedriger Priorität auseinander gehalten werden, damit eine sinnvolle Zielfunktion entstehen kann. Diese Einzelheiten sind jedoch trivial und kommen deshalb in dieser Abhandlung über die ILP Formulierung nicht vor.

7.5 Gütegarantie des ILPs

Wir müssen vorerst zeigen, dass die ILP immer eine gültige Lösung im Sinne einer Permutation der Farhzeuge liefert. Dies ist sehr einfach, da Nebenbedingungen (1) und (2) garantieren, dass sich an jeder Position genau ein Fahrzeug befinden muss und jedes Fahrzeug genau an einer Stelle stehen muss. Somit ist die Lösung eine Permutation.
Die Nebenbedingunden (3) und (4) erzeugen die Matrix P, die die aufsummierte tatsächliche Reihenfolge der Constraints wiedergibt. Wenn wir, wie in der Aufgabenstellung beschrieben, die Verletzungen zählen, müssen wir Betrachtungsfenstern in die Matrix P legen. Die Zählung wird durch die Aufsummierung in P auf eine Differenzbildung vereinfacht. Damit wird die Anzahl der Constraintverletzungen korrekt gezählt und in der Matrix V gespeichert. Ebenso verhält es sich bei der Farbwechselzählung. Alle Farbwechsel können als Sprünge von null auf eins in der Matrix Q, die die tatsächliche Reihenfolge der Farben darstellt, aufgefasst werden. Diese Sprünge werden in (9) und (10) gezählt und in der Matrix W gespeichert. Die Nebenbedingung (8) garantiert, dass die maximale Farbblockgrenze nicht verletzt wird, indem die Blöcke der gleichen Farben in der Matrix Q überprüft werden.
Somit liefert die Lösung des ILPs auch genau eine Optimallösung des Originalproblems, da der Zielfunktionswert des ILPs durch die Herleitung exakt mit der Zielfunktionswert des Originalproblems übereinstimmt.

7.6 Lösen des ILPs

Die Paintshop Applikation verfügt über dir Möglichkeit, eine geladene Probleminstanz auf die oben erwähnte ILP Form zu bringen und sie in eine Datei zu speichern. Der Sinn besteht natürlich darin, das Problem von einem Programm wie CPLEX exakt lösen zu lassen. CPLEX ist eine Optimierungssoftware, die von ILOG, Inc. entwickelt wurde. Sie benutzt einen „Branch and Cut“ Algorithmus, um unter anderen LPs zu lösen.
Im Moment beträgt die maximale Problemgröße je nach Anzahl und Länge der Verhältnisconstraint bei ungefähr 20 Farhzeugen, sodass CPLEX das Problem in akzeptabler Zeit lösen kann. Die Anzahl der Variablen und damit auch die Anzahl der Nebenbedingungen steigen quadratisch mit der Anzahl der Fahrzeugen. Die handhabbare Größe des Problems lässt sich sicher durch geschicktes Umformulieren der Nebenbedingungen des ILPs noch steigern. Es ist ebenfalls möglich, zusätzliche einschränkende Nebenbedingungen einzuführen, die den Branch and Cut Prozess beschleunigen, indem sinnlose Verzweigungen eliminiert werden.

8 Komplexität

Theorem 8.1 *Das vorliegende Paintshop Problem ist NP schwer.*

In Folge wird diese Behauptung durch einen indirekten Beweis bestätigt, indem die Problemstellung auf ein anderes NP schweres Problem zurückgeführt wird.

Wir erinnern uns an das Problem in „Sequencing Bitvectors with Distance Constraints [8]“ (DCBS):

> Gegeben: Multimenge $F = \{b_1, \ldots, b_1, b_2, \ldots, b_2, b_\alpha, \ldots, b_\alpha\}$ von β-dimensionalen 0-1 Vektoren und Schranken $\underline{m}_j, \overline{m}_j \in N$ mit $\underline{m}_j < \overline{m}_j (j = 1 \ldots \beta)$.
> Gesucht: Eine Sequenz $b_{i_1}, b_{i_2}, \ldots, b_{i_n}$ der Vektoren in F, sodass für jede Komponente j zwischen je zwei „1“ mindestens $\underline{m}_j$ und höchstens $\overline{m}_j$ „0“ existieren.

Es gibt, wie bereits vorhin erwähnt, im Gegensatz zum aktuellen Problem eine Zusatzbedingung, die besagt, dass ein Maximalabstand zwischen zwei Fahrzeugen mit einer bestimmten Komponente existieren muss. Außerdem kann die $\underline{m}_j$ nur Constraints der Form $\frac{1}{N}$ ausdrücken.
Paper [8] bestätigt jedoch, dass das Problem durch das Vernachlässigen von den $\overline{m}_j$ Constraints und durch die Einschränkung von $\underline{m}_j = 1 \forall j \in J$ immer noch NP schwer ist:

> Nickel zeigt, dass auch durch starke Vereinfachungen das (DCBS) NP schwer bleibt, indem er eine Reduktion auf das „Hamiltonsche Pfad Problem“ macht.

- Gegeben ist ein Graph $G(V, E)$
- Existiert in G ein Pfad, der jeden Knoten in V genau ein Mal besucht?

Theorem 8.2 *(DCBS) bleibt NP schwer wenn die folgende Simplifizierung vorgenommen wird: Die Obergrenzen $\overline{m}_j$ werden alle ignoriert und die Untergrenzen sind auf $\underline{m}_j = 1$ fixiert.*

Beweis Sei $\overline{G} = G(V, \overline{E})$ mit $\overline{E} = (V \times V) \backslash E$ der Komplementärgraph zu G und $m := |\overline{E}|$ die Anzahl der Kanten.
Für alle Knoten $v \in V$ erzeugen wir einen Knotenvektor

$$b_v \in \{0,1\}^{m+2} \text{ mit } b_v^e = \begin{cases} 1 & v \notin e \\ 0 & v \in e \end{cases} .$$

Dann sind zwei Knotenvektoren v_1 und v_2 in der Reihenfolge $\ldots, v_1, v_2, \ldots$ in der Sequenz angeordnet, wenn und nur wenn eine Kante von v_1 nach v_2 in G existiert. D.h. Es existiert keine Kante von v_1 nach v_2 in $\overline{G}$ und somit nehmen die Knotenvektoren b_{v_1} und b_{v_2} nicht beide den Wert „0“ in den Komponenten $e = 1 \ldots m$ an.
Wenn es einen Hamiltonschen Pfad $v_1, \ldots, v_{|V|}$ in G gäbe, dann wäre

$$b_1, b_2, \ldots, b_{|V|-1}, b_{|V|}$$

offensichtlich eine zulässige Sequenz. Umgekehrt, wenn

$$b_1, b_2, \ldots, b_{|V|-1}, b_{|V|}$$

eine zulässige Sequenz wäre, dann müsste $v_1, \ldots, v_{|V|}$ ein Hamiltonscher Pfad sein.
Nun führen wir eine logische Invertierung aller Bitvektoren durch und der Beweis ist fertig.

Mit diesem Beweis können wir die folgende Schlussfolgerung ziehen:
Wenn im vorliegenden Problem die Farbwechselzählung weggelassen und alle Constraints auf $\frac{1}{2}$ vereinfacht werden, entsteht eine leichtere Aufgabe. Wenn diese Aufgabe nicht NP schwer wäre, dann könnten wir das vereinfachte Problem im Paper [8] auch in P lösen.
Das reduzierte Problem mit uniformen Constraints von $\frac{1}{2}$ ist äquivalent zum in [8] reduzierten Problem mit $\underline{m}_j = 1$ und $\overline{m}_j = 0$. Dies ergibt jedoch einen Widerspruch zum Beweis in [8].
Wenn das reduzierte Problem schon NP schwer ist, dann muss das ursprüngliche Problem mit Farbwechselzählung und beliebigen Constraints $\frac{Z}{N}$ auch NP schwer sein.

9 Zusammenfassung und Ausblick

Die gesamte Paintshop Applikation liegt dem HuGS Core zugrunde. Dies hat wesentliche programmiertechnische Vorteile, da viele essentielle Routinen wie z.B. die Suchalgorithmen nicht implementiert werden müssen. Die Schnittstellen sind sehr gut durchüberlegt und flexibel, wenn neue Applikationen in HuGS integriert werden sollen.
Um Paintshop in das System einzubetten, ist es lediglich notwendig, alle Schnittstellenklassen abzuleiten und diese zu implementieren. Die wichtigsten Überlegungen, die gemacht werden müssen, bestehen darin, die Auswertung der Lösung, den Aufbau der Nodes, die Wahl der Moves und die Visualisierung zu gestalten. Die Effizienz der Suchalgorithmen ist direkt von diesen designerischen Überlegungen abhängig, da das Interface der Suchalgorithmen sehr allgemein implementiert und auf die Schnittstellen angewiesen sind.

Ein weiterer Vorteil, der leider bei „ROADEF Challenge 2005“ leider nicht zum Tragen kommen kann, ist die menschliche Interaktion während des Optimierungsprozesses. In „Human-Guided Tabu Search [3]“ sind detaillierte Untersuchungen bezüglich des Vergleichs zwischen nonguided und guided Tabu Search gemacht worden, welche ergeben, dass die guided Version einen wesentlichen Vorteil bringt.
Der Script Modus bietet hierfür nur einen bedingten Ersatz, da nicht alle Möglichkeiten voraussagbar sind.

Die Nachteile liegen ebenfalls klar auf der Hand. Da HuGS eine Java Applikation ist und zudem alle Methoden in der Core sehr allgemein implementiert sind, ist die Laufzeit eher lang.

Es wäre interessant zu sehen, ob eine Portierung des gesamten HuGS Frameworks auf eine Programmiersprache wie z.B. C++ einen großen Gewinn in der Laufzeit bringen würde. Die Laufzeiteffizienz dieser beiden Programmiersprachen unterscheiden sich immerhin recht stark.

Weiteres könnte die interne Struktur von HuGS in dem Sinne modifiziert werden, dass sie bei der Ausführung der Moves effizienter arbeitet. Im Moment muss die alte Lösung in eine History abgespeichert werden, bevor die neue Lösung übernommen wird. Das bedeutet je nach Problemgröße ein nicht unwesentlichen Aufwand, da alle Bestandteile einer Lösung übertragen werden müssen.

Außerdem erfolgen viele Kopiervorgängen von Lösungen, wann immer ein neuer Move ausprobiert wird. Diese fest verankerte Routinen im HuGS Core bewirken auch eine Verschlechterung der Laufzeit, sind aber umgekehrt nicht unwichtig für die menschliche Interaktion. Die History Aufzeichnungen aller vergangenen Lösungen sind z.B. für den Benutzer von wesentlicher Bedeutung, wenn er glaubt, dass sich die Suche in einem lokalen Optimum verfangen hat und ein Backtracking einleiten möchte.

Ein ebenfalls interessanter Ansatz wäre eine kombinierte Lösung aus HuGS und CPLEX. Wir könnten das Problem in kleine Teile zerlegen und diese mittels ILP exakt lösen. Danach fügen wir die Teillösungen wieder zusammen. Momentan verspricht dieser Ansatz jedoch noch wenig Erfolg, da mittels dieser ILP Formulierung nur sehr kleine Teilaufgaben in angemessener Zeit lösen kann. Wie schon erwähnt liegt die handhabbare Problemgröße bei etwa 20 Fahrzeugen mit niedriger Constraintkomplexität. Wegen der starken Korrelation des Problems variiert die Qualität der Lösung auch bei kleinen Änderungen in der Reihenfolge der Farhzwuge sehr stark. Dadurch haben optimale Teillösungen von 20 Fahrzeugen leider wenig Aussagekraft. Interessanter wird es erst ab Teillösungsgrößen von hundert und mehr Fahrzeugen.

10 Anhang

In diesem Kapitel wird beschrieben, wie die Paintshop Oberfläche dargestellt wird und wie wir sie bedienen können. Die folgenden Seiten sollen eine Art Benutzerhandbuch darstellen und dem Benutzer helfen, mit der Applikation möglichst vertraut zu werden. Wie schon erwähnt, ist die menschliche Interaktion während des Optimierungsprozesses von großer Bedeutung. Dazu muss der Benutzer natürlich mit der Bedienungsoberfläche gut zurechtkommen.

10.1 Interface von HuGS-Paintshop

Das Interface sieht etwa wie in Abbildung 18 aus und gliedert sich in drei Abschnitte.

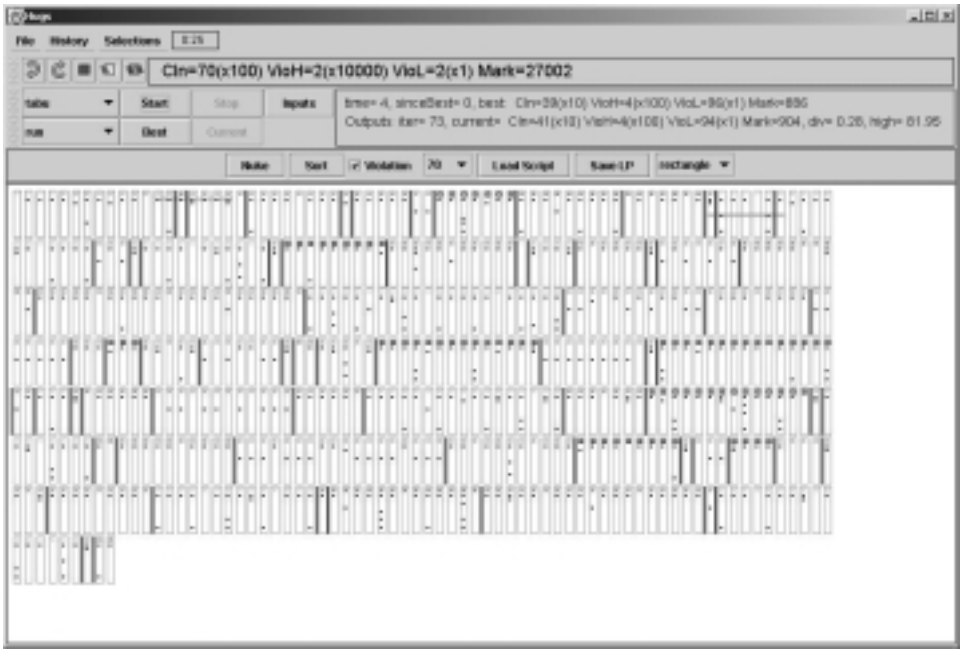


Abbildung 18: Interface

Allgemeines HuGS Bedienungsfeld

Diese Steuerelemente sind praktisch bei allen HuGS Applikationen identisch und erfüllen die nicht problemspezifischen Befehle.

Abbildung 19: HuGS Bedienungsfeld

- File
  save / load solution: Speichert / Lädt eine Lösungsinstanz.
  save / load problem: Speichert / Lädt eine Probleminstanz.

- History
  get best: Zeigt die beste Lösung an, die bis jetzt gefunden wurde.
  history: Zeigt alle bis jetzt gefundenen Lösungen an.

- Selections
  clear selection: Hebt alle Markierungen auf. (hotkey: Doppelklick auf die Applikationsfläche)
  set low / medium / high: Setzt die Mobility der markierten Nodes auf low / medium / high.
  set all low / medium / high: Setzt die Mobility aller Nodes auf low / medium / high.

- Die fünf Symbolbuttons (v.l.n.r.)
  zurück: Geht einen Schritt zurück zur vorigen Lösung.
  nach vorne: Geht einen Schritt nach vorne zur nächsten Lösung.
  low / medium / high Mobility: Genau wie „set low / medium / high“

- Die zwei Combo Boxen (v.o.n.u.)
  Algorithmus: Bestimmt den Algorithmus, der für die Optimierung verwendet werden soll.

Änderungen anzeigen: run (keine automatischen Änderungen anzeigen), step (pro Schritt eine Änderungen anzeigen), detail (alle Änderungen anzeigen), poll (Änderungen in Taktintervall anzeigen)

- Die fünf beschrifteten Buttons
  Start: Startet den Optimierungsprozess.
  Stop: Stoppt den Optimierungsprozess. (Start für fortsetzen)
  Best: Zeigt die beste Lösung an, die bisher gefunden wurde. Sie leuchtet grün, wenn eine bessere Lösung als die momentan angezeigte gefunden wurde und gelb, falls durch Backtracking eine lokal bessere Lösung gefunden wurde.
  Current: Zeigt die momentane Lösung an.
  Input: Bestimmt durch den Searchadjuster die Parameter für den Suchalgorithmus.

Applikationsspezifisches Bedienungsfeld

Diese Steuerelemente sind speziell für das Paintshop Problem angepasst.

Abbildung 20: Paintshop Bedienungsfeld

- Randomize
  Versetzt die markierten Nodes in eine zufällige Reihenfolge, wobei die farbliche Anordnung ignoriert wird.

- Nuke
  Versetzt die markierten Nodes in eine zufällige Reihenfolge, wobei nur gleichfarbige Knoten vertauscht werden.

- Sort
  Sortiert die markierten Nodes in eine farbliche Anordnung, sodass die Anzahl der Farbwechsel minimal wird.

- Violation
  Schaltet die Visualisierung der Constraintverletzungen und der Farbwechsel ein und aus.

- Darstellungsgröße
  Gibt an, wie viele Nodes pro Zeile dargestellt werden sollen. Je kleiner die Zahl, desto größer die Darstellung.
- Load Script
  Lädt eine Scriptdatei, die automatisch menschliche Interaktionen simuliert.
- Save Script
  Speichert die durchgeführten und somit registrierten Befehle in eine Scriptdatei, die mit „Load Script“ geladen werden kann.
- Save LP
  Speichert eine LP Formulierung der momentan geladenen Probleminstanz im CPLEX Format ab.
- Markierung
  Wechselt den Markierungsmodus.

Applikationsvisualisierung

Das ist das Hauptvisualisierungsfenster. Alle Fahrzeuge mit ihren Farben, Komponenten, etc. werden hier angezeigt.

Abbildung 21: Paintshop Applikationsvisualisierung

Die Fahrzeuge werden in der Lösungsreihenfolge von links nach rechts und von oben nach unten dargestellt. Die Umrandungsfarbe besagt folgende Information:

- grau: Das Fahrzeug stammt aus dem vorigen Tag und wird nur zur Berechnung der Constraintverletzung herangezogen. Die Position ist unveränderbar.
- grün: high Mobility, d.h. die Position des Fahrzeug kann von den Optimierungsalgorithmen beliebig verändert werden.
- gelb: medium Mobility, d.h. die Position des Fahrzeug kann zwar nicht von den Optimierungsalgorithmen verändert werden, aber sie können die Mobility wieder auf high setzen.
- rot: low Mobility, d.h. die Position des Fahrzeug kann nicht von den Optimierungsalgorithmen beliebig verändert werden. Die Mobility kann nur manuell auf low oder davon weg gesetzt werden.

In den Kästchen befinden sich die Angaben über die Farbe des Fahrzeugs und die benötigten Komponenten. Wenn der Mousecursor auf ein bestimmtes Fahrzeug zeigt, so werden alle Fahrzeuge, die dieselbe Farbe besitzen, durch eine zusätzliche Umrandung des Farbquadrates hervorgehoben. Wenn sich die Farbe zwischen zwei aufeinanderfolgende Fahrzeuge unterscheidet, dann ist ein senkrechter Strich dazwischen sichtbar, der diesen Farbwechsel anzeigt. Des weiteren werden Constraintverletzungen durch waagrechte Rechtecke über den verletzten Komponentenbereich dargestellt. Die Farbe ist rot bei einem wichtigen Constraint und blau bei einem unwichtigen Constraint.

10.2 Menschliche Interaktion

Hier wird beschrieben, wie der Benutzer im Optimierungsprozess interaktiven mitwirken kann. Um eine Aktion auszulösen, müssen lediglich die beschriebenen Vorgangsweisen benutzt werden.

Markierung und Mobilities
Um ein Fahrzeug zu markieren, klicken Sie auf den Node (das Kästchen), der dieses repräsentiert. Um mehrere zu markieren, können Sie entweder alle der Reihe nach durch einzelne Klicks markieren, oder Sie ziehen ein Rechteck um die zu markierenden Knoten. Die markierten Kästchen bekommen eine rote Umrandung.
Die Markierung kann einzeln rückgängig gemacht werden, indem ein markierter Knoten wieder angeklickt wird. Ein Doppelklick hebt alle Markierungen auf. Beachten Sie, dass die Knoten nur in aufeinanderfolgender Reihenfolge markiert werden können.

Um die Mobility der Knoten zu ändern, markieren Sie einfach diese und benutzen die Mobility Buttons am HuGS Bedienungsfeld.

Manuelle Moves
Der Benutzer hat die Möglichkeit, alle Aktionen (Moves), die von den Algorithmen ausgeführt werden, manuell einzuleiten.

- Swap Move: Markieren Sie einen Knoten und klicken Sie mit der rechten Maustaste den Knoten an, mit dem er vertauscht werden soll.
- Block Move: Markieren Sie eine oder mehrere Knoten, dann ziehen Sie mittels Drag and Drop mit der rechten Maustaste die markierten Fahrzeuge auf die gewünschte Position, um den ganzen Block von Knoten dorthin zu verschieben.
- Nuke Move: Markieren Sie eine Reihe von Knoten und klicken Sie auf „Nuke“, um die markierten Knoten in eine zufällige Reihenfolge zu versetzen, wobei nur gleichfarbige Knoten vertauscht werden.
- Sort Move: Markieren Sie eine Reihe von Knoten und klicken Sie auf „Sort“, um die markierten Knoten in eine farbliche Anordnung zu sortieren, sodass die Anzahl der Farbwechsel minimal wird.

Input Feld und Searchadjuster
Der Searchadjuster, der durch das Input Feld aufgerufen wird, hat die Funktion, den Suchraum manuell zu beeinflussen. Jeder der im HuGS Core enthaltenen Suchalgorithmen besitzt seine eigenen Suchparametern, aber alle enthalten folgende vier Einträge:

- swaps: Sollen Swap Moves überhaupt durchgeführt werden?
- Swap Moves: Wie viele Swap Moves sollen pro Suchiteration durchgeführt werden? (0 bedeutet alle möglichen Kombinationen der Instanz durchsuchen.)
- blocks: Sollen Block Moves überhaupt durchgeführt werden?
- Block Moves: Wie viele Block Moves sollen pro Suchiteration durchgeführt werden? (0 bedeutet alle möglichen Kombinationen der Instanz durchsuchen.)

Des weiteren enthält Tabu folgende zusätzliche Einträge:

- greedy: Soll die erste bessere Lösung gleich genommen werden?

- maxPly: Der maximale Verschachtelungsgrad der Moves. Defaultmäßig ist hier eins eingestellt (dadurch bleibt der Suchraum linear in Abhängigkeit von den aus der Move List generierten Moves), da Tabu höhere Kosten als Greedy besitzt und deswegen ein höherer Wert die Performance merkbar verschlechtert.

- memory: Wie viele Schritte soll sich Tabu merken, um Backtracking zu verhindern?

- minD: Wie schon im allgemeinen HuGS Kapitel erwähnt, ist das der minimum Diversity Wert, der einen Schwellenwert angibt, wann die „Diversify“ Funktion eingesetzt werden soll.

10.3 Lebenslauf

- 1980: Geboren in Shanghai, V.R. China
- 1986: Auswanderung nach Grieskirchen, Österreich
- 1986 - 1990: Volksschule Grieskirchen
- 1990 - 1992: Hauptschule Grieskirchen
- 1993: Umzug nach Wien
- 1993 - 1999: Bundesrealgymnasium 15, Henriettenplatz, Wien
- 1999: Matura am Bundesrealgymnasium 15, Henriettenplatz, Wien
- 1999 - Jetzt: Informatikstudium an der Technischen Universität Wien

Nebentätigkeiten

- 1995 - 1999: Teilnahme an der Mathematik Olympiade
- 1996 - 1999: Teilnahme an der Chemie Olympiade
- 1999 - 2000: Beschäftigung bei Löwa Ges.m.b.H.
- 2001: Organisationskomitee für Logic Colloquium 2001
- 2003: Organisationskomitee für ESSLLI/CSL 2003
- 2000 - Jetzt: Studienassistent am Institut für Computergraphik und Algorithmen, Abteilung für Algorithmen und Datenstrukturen

Literatur

[1] G. W. Klau, N. Lesh, J. Marks, M. Mitzenmacher: *Human-Guided Search: Survey and Recent Results.* Manuscript, 2003.

[2] J. P. Garcia-Sabater: *The Problem of JIT Dynamic Sequencing. A Model and a Parametric Procedure.* ORP^3, Paris, September 26–29, 2001.

[3] Klau, G. W., N. Lesh, J. Marks und M. Mitzenmacher: *Human-Guided Tabu Search.* In: *Proc. of AAAI 2002 (The Eighteenth National Conference on Artificial Intelligence, Edmonton, Alberta, Kanada, July/August 2002)*, Seiten 41–47, Menlo Parc, CA, USA, 2002. AAAI Press.

[4] Klau, G. W., N. Lesh, J. Marks, M. Mitzenmacher und G. T. Schafer: *The HuGS Platform: A Toolkit for Interactive Optimization.* In: *Proc. of AVI 2002 (International Working Conference on Advanced Visual Interfaces, Trento, Italien, May 2002)*, Seiten 324–330, New York, 2002. ACM Press.

[5] K. Smith, M. Palaniswami, M. Krishnamoorthy: *Traditional Heuristic versus Hopfield Neural-Network Approaches to a Car Sequencing Problem.* European Journal of Operational Research, 93(2):300–316, 1996.

[6] N. Brauner, Y. Crama: *Facts and questions about the maximum deviation just-in-time problem.* Discrete Applied Mathematics, Article 3174, 2003.

[7] R. Nickel, W. Hochstättler: *Gerechtigkeit am Fließband - Sequenzierung von Bitvektoren unter Nebenbedingungen.* Manuscript, 2003.

[8] R. Nickel, W. Hochstättler: *Sequencing Bitvectors with Distance Constraints.* Manuscript, 2003.