Fachhochschule Brandenburg

Informatik und Medien

D I P L O M A R B E I T

Lösung komplexer "Pickup and Delivery" Probleme unter Einbeziehung moderner Constrainttechniken

Vorgelegt von:
Mathias Lühr
Informatik / Intelligente Systeme
Matrikelnummer: 972128

Weberplatz 20
14482 Potsdam

Gutachter:
Prof. Dr. Jochen Heinsohn
Dipl.-Ing. (FH) Torsten Storrer

Potsdam, den 22.04.2002

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

„Mit innovativen Technologien und effizienten Lösungen setzen wir alles daran, dass Menschen, Güter und Daten von A nach B gelangen. Ohne Umwege und ohne unnötige Kosten.“ [IVU Web]

Das kann man im Profil der IVU Traffic Technologies AG lesen, an der die vorliegende Arbeit erstellt wurde. Güter von A nach B zu bringen, kann so schwer nicht sein. Die Meisten vermögen eine Tasse Kaffee von der Küche zum Schreibtisch zu transportieren – wenn sich beide Räume in der gleichen Wohnung befinden. Ebenso einfach ist es, ein Buch vom Nachttisch aufzunehmen und im Wohnzimmer im Bücherregal abzulegen. Viele sind in der Lage, mit dem Fahrrad ein Paket von einem Haus zum nächsten zu bringen. Die Schwierigkeiten, einen einzelnen Transport zu erledigen, sind relativ gering.

Aufnehmen – Transportieren – Ablegen.

Doch wie sieht es aus, wenn es um zehntausende Pakete geht, wenn größere Distanzen zwischen den Empfängern zurückgelegt werden müssen? Da geht es plötzlich um das Management der Fahrradflotte, um Einhaltung von Arbeits- und Pausenzeiten. Es geht um Kostenreduzierung durch kürzere Wege und Optimierung der Auslastung von Fahrrädern. Es geht um optimale Ausnutzung der Rucksackkapazitäten und darum, welche Pakete kombiniert werden dürfen. Es geht auch darum, wann, wo, welches Paket abholbereit ist und zu welcher Zeit es abgegeben werden kann. Und natürlich soll in Stadt- und Landfahrer unterschieden werden, die in der jeweiligen Umgebung besonders schnell oder besonders langsam sind.

Bezogen auf die IVU Traffic Technologies AG handelt es sich nicht um Fahrräder, sondern um Fahrzeuge, die beispielsweise Beton oder Kies transportieren. So liegen die momentanen Aufgaben in der Baubranche, sollen aber auch im Personenverkehr und in der Entsorgung gesucht werden.

Ein möglicher Anwendungsfall könnte aus einer Vielzahl von Baustellen bestehen, welche mit Kies versorgt werden sollen. In einem gewissen Umkreis einer jeden Baustelle liegen Kiesgruben oder einfach „Kiesbeladestationen“. Vielleicht befindet sich eine Beladestation näher an dieser Baustelle als eine andere, vermag aber nur kleine Fahrzeuge abzufertigen. Weiterhin muss die Verfügbarkeit von Fahrzeugen beachtet werden. Es ist durchaus möglich, dass der Fahrer nach vier Stunden Dienst eine Pause von mindestens fünfundvierzig aber höchstens sechzig Minuten einlegen

muss. Es kann auch sein, dass das Fahrzeug erst am Nachmittag zur Verfügung steht, weil vormittags eine TÜV-Plakette erneuert werden muss.

In einem zweiten Fall muss eine Menge von Mülltonnen entleert werden. Das Fahrzeug soll mehrere Ladungen aufnehmen, ehe es diese ablädt. Erst nachdem alle Mülltonnen in das Fahrzeug hinein entleert wurden, sollte der Müllplatz angefahren werden.

Den Fahrradkurier führt eine günstige Tour mit dem gefüllten Rucksack nacheinander auf dem kürzesten Weg zu den Empfängern und endet am Postzentrum, wo sie begonnen hat. Diese Art des Problems erinnert sehr stark an ein „Traveling Salesman“ Problem (kurz: TSP), bei dem ein Handelsreisender mehrere Geschäftskunden besuchen muss. Er soll an jeder Türe nur einmal klingeln und zum Schluss wieder an seiner Ausgangsposition zurückgekehrt sein. Erweitert um Zeitbeschränkungen, spricht man von einem „Traveling Salesman Problem with Time Windows“ (kurz: TSP-TW). Da sich die Entfernung zwischen den Städten $\overrightarrow{AB}$ von $\overrightarrow{BA}$ unterscheiden kann, wird das „Asymmetric“ vorangestellt: „Asymmetric Traveling Salesman Problem with Time Windows“ (kurz: ATSP-TW).

In der Literatur findet sich ebenfalls die Bezeichnung „Vehicle Routing Problem“ (kurz: VRP), die stärker den Einsatz einer Flotte von Fahrzeugen betont. Man könnte deshalb auch von einem „Vehicle Routing Problem with Time Windows“ sprechen. Jedoch würde der wahre Charakter nicht ganz getroffen werden, da bei dem vorliegenden Problem die Reihenfolge der einzelnen Anfahrtspunkte beachtet werden muss. Aus diesem Grund hat sich in der Literatur die Bezeichnung „Pickup and Delivery“ Problem etabliert. Eine Erweiterung mit „with Time Windows“ erscheint berechtigt, ist aber nicht gebräuchlich. Dennoch soll auf die nahe Verwandtschaft zum „Traveling Salesman Problem with Time Windows“ hingewiesen werden.

Man versucht solche Probleme zu lösen, indem man alle möglichen Reihenfolgen der zu durchwandernden Städte überprüft und miteinander vergleicht. Ein solcher Algorithmus erzeugt also jede mögliche Permutation. Und da hier von einem vollständigen Graphen ausgegangen wird, kann die Anzahl durch *n!* bestimmt werden. Die Reise mit dem kürzesten Weg muss schließlich die optimale sein.

Bei einem TSP mit zehn Städten, würde 10!=3.628.800 - mal eine Schleife durchlaufen werden. Wenn die Maschine zehn Durchläufe in einer Sekunde schafft, benötigt sie zur vollständigen Lösung 100,8 Stunden also 4,2 Tage. Sind es aber elf Städte, wäre der Rechner bereits 46,2 Tage beschäftigt! Nun ist es wenig akzep-

tabel, Probleme der Größenordnung zehn zu lösen. Die Unberechenbarkeit von 100 Städten auf der selben Maschine, ist so nicht zu vermeiden. Es wäre daher wünschenswert, dass der Algorithmus vorher bestimmte Berechnungen ausschließt. Beispielsweise macht es keinen Sinn, ein leeres Fahrzeug zu weiteren Entladestellen zu schicken. Von dieser Überlegung getrieben, können vielleicht weitere Informationen in die Problemlösung eingebracht werden, die überflüssige Teilaufgaben eliminieren helfen. Diese Informationen werden Randbedingungen oder Constraints genannt. Diese zu untersuchen und mit deren Hilfe ein komplexes Routingproblem zu lösen, ist Aufgabe der vorliegenden Arbeit. Dabei sollen moderne Constrainttechniken untersucht und analysiert werden. Die notwendigen Teilaufgaben gliedern sich in drei Bereiche:

1. Analyse der Fahrzeugplanung hinsichtlich der Anforderungen an die zu schaffende Softwarebibliothek,
2. Evaluierung geeigneter Constraint-Solver-Bibliotheken hinsichtlich des Fahrzeugroutens und Auswahl eines zweckmäßigen Constraint Solvers,
3. Entwurf und Implementierung einer Softwarebibliothek unter Einbeziehung des gewählten Constraint-Solvers, welche das gegebene „Pickup and Delivery“ Problem lösen kann.

Das wichtigste Kriterium ist jedoch, dass die Bibliothek von Delphi-Programmen aus nutzbar und auf dem Betriebssystem Windows NT 4 lauffähig ist.

Dazu werden im Kapitel 2 die wesentlichen Eigenschaften von Constraints erläutert. Nach grundlegenden Erläuterungen geht es im Kapitel 2.3 um Konsistenzprüfungen. Kapitel 2.4 beschreibt die elementaren und weiterführenden Suchverfahren. Danach werden grundlegende Heuristiken (Kapitel 2.5) und erweiternde Constraint-Techniken (Kapitel 2.6) erläutert. Im Kapitel 3 werden alternative Lösungsstrategien skizziert und das Kapitel 4 gibt einen Überblick und trifft eine Systemauswahl. Schließlich wird ab dem Kapitel 5 eine Lösung erarbeitet.

2 Constraints

In den folgenden Abschnitten wird der Constraintbegriff genauer untersucht. Nach einer Positionsbestimmung werden im Kapitel 2.2 wichtige Eigenschaften und Begriffe eines Constraintproblems erläutert. Danach finden sich Beschreibungen über Konsistenzalgorithmen und Suchmethoden. In Kapitel 2.5 werden zwei weitere Heuristiken erläutert und schließlich findet dieser Abschnitt in Bemerkungen über

erweiternde Initiativen seinen Abschluss.

2.1 Einleitung

Das Wort „Constraint“ deutet es bereits an: es handelt sich hierbei um eine Art von Beschränkung. So kann durch x<4 verlangt werden, dass x nicht größer als drei wird. Und mit y<x wird erzwungen, dass y nicht größer als x und damit nicht größer als zwei wird (mit x, y $\in$ $\mathbb{N}$). In [Frühwirth & Abdennadher 1997] wird ein Constraint als Relation zwischen Constraintvariablen angesehen. Es handelt sich dabei um Variablen, welche über einen Wertebereich oder eine Domäne verfügen. Diese ist endlich und kann gebrochene oder ganze Zahlen enthalten. Allerdings werden in bestimmten praktischen Fällen Domains mit unendlich großem Wertebereich erlaubt.

Da es sich jeweils um eine endliche Menge von Variablen und einer endlichen Menge von Constraints handelt, muss die Tiefe des Suchbaumes ebenfalls endlich sein und eine Tiefe von $|X|$ (Kardinalzahl der Menge aller Constraintvariablen) haben. Denn zur Lösung des Problems muss jeder Variable ein Wert zugewiesen werden, wozu nur $|X|$-Schritte notwendig sind. Diese Eigenschaft und die folgenden in [Barták 1998] zusammengefassten Merkmale können in Constraintlösern sehr gut verwendet werden:

1. Ein Constraint präsentiert Teilinformationen.
2. Ein Constraint ist richtungslos, es kann sich auf ein x auswirken und indirekt ein y beeinflussen und umgekehrt.
3. Constraints sind deklarativ.
4. Constraints sind additiv. Es ist möglich, mehrere Constraints auf eine Variable wirken zu lassen. Die Reihenfolge muss dabei nicht beachtet werden.

In dem folgenden Kapitel werden diese Eigenschaften um Definitionen zu Constraintproblemen erweitert.

2.2 Constraintprobleme

Ein Problem, welches nach der Erfüllbarkeit der Randbedingungen fragt, bezeichnet man als *Erfüllbarkeitsproblem* oder *Constraint Satisfaction Problem* (kurz: CSP). So lässt sich das bekannte Graphenfärbeproblem als CSP definieren.

Beispiel 1:

Eine Landkarte mit drei Kleinstaaten soll so eingefärbt werden, dass sich jedes Land von seinem Nachbarn unterscheidet. Zur Auswahl stehen vier Farben, sodass sich

das Problem folgendermaßen beschreiben lässt:
Die Länder werden durch drei Variablen repräsentiert. Der Wertebereich von A, B und C ist die Menge {rot; grün; blau; magenta}. Die gegebenen Einschränkungen, die durch das Problem vorgegeben sind, können durch folgende drei Constraints beschrieben werden:

$c_1 := A \neq B$

$c_2 := B \neq C$

$c_3 := C \neq A$

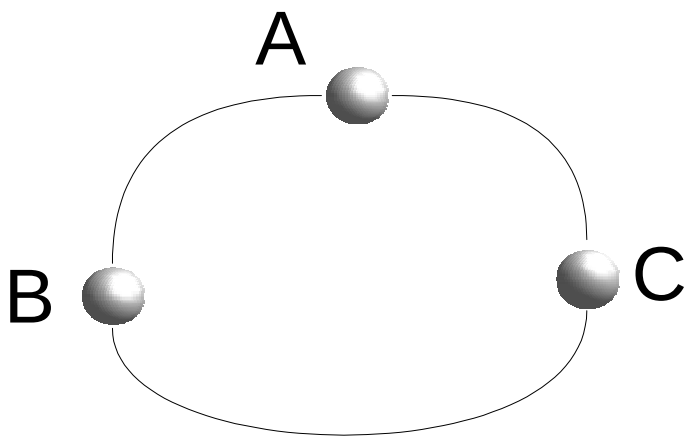


Abbildung 1 Graphenfärbeproblem

Die Suche beginnt damit, dem Staat A die Farbe rot zuzuweisen. Danach können die Constraints überprüft werden, um eine mögliche Verletzung aufzudecken oder den Wertebereich der Constraintvariablen einzuschränken. Das Constraint c_1 bewirkt, dass der Wertebereich D von B eingeschränkt wird:
D(B) = {grün; blau; magenta}.

Das Constraint c_3 schränkt C ein: D(C) = {grün; blau; magenta}.

Der Suchalgorithmus fährt fort und legt für den Staat B die Farbe grün fest. Das erneute Überprüfen der Constraints ergibt Folgendes: c_1 belässt D(A) auf {rot} und c_2 verringert D(C) auf {blau, magenta}.
Im letzten Schritt wird C auf blau festgelegt. Die nachfolgende Überprüfung der Constraints ergibt keine Unstimmigkeiten, sodass die Lösung akzeptiert werden kann.

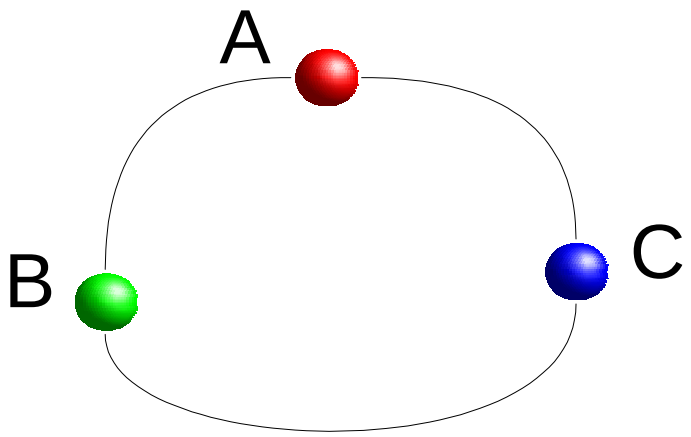


Abbildung 2 Graphenfärbeproblem (Lösung)

In dem obigen Beispiel wurden einige Begriffe stillschweigend eingeführt, die in folgender Definition festgehalten werden [Heinsohn & Socher 1999]:

Ein Constraintproblem besteht aus einer endlichen Menge X von Variablen und einer endlichen Menge C von Constraints. Jede Variable $x \in X$ ist mit einem Definitionsbereich oder einer Domain D(x) versehen.

Außerdem gilt: wirkt ein Constraint c auf k Variablen, so heißt c k-stellig. Bei einem Problem mit höchstens 2-stelligen Constraints spricht man von einem Constraintnetz (CN), welches sich als Graph darstellen lässt: CN = (X, C).

So sieht man eine Kante als 2-stelliges Constraint an und stellt eine Constraintvariable als Knoten des Graphen dar. Mehrstellige Constraints können ebenfalls durch Constraintnetze dargestellt werden, indem sie in 2-stellige Constraints transformiert werden [Barták 1998]. Weiter im Bereich der 2-stelligen Constraints verbleibend, spricht man von einem *vollständigen Constraintnetz*, wenn es zu jedem Variablenpaar ein Constraint gibt, welches sie „verbindet“ oder besser: sie gemeinsam beschränkt.

Die Lösung eines Constraintproblems mit n Variablen lässt sich als n-Tupel (L_1, L_2, ..., L_n) formulieren, wobei L_i ein Wert aus der Domäne der Constraintvariable x_i ist und jedes Constraint erfüllt sein muss. Das Tupel weist dabei jeder Variablen einen Wert oder ein Label zu. Eine Constraintvariable mit einem Label heißt instantiiert und manchmal auch gebunden. Man spricht von Markierung [Heinsohn & Socher 1999], wenn jede Constraintvariable ein Label besitzt. Wird durch eine Markierung kein Constraint verletzt, ist sie konsistent und stellt eine Lösung dar.

Somit ist das beschriebene Tupel von Labels eine konsistente Markierung.

Ist es für ein Constraintproblem nicht möglich, eine Lösung zu finden, spricht man von einem *überbestimmten* oder *overconstrained* Problem. Sollten aber mehrere Lösungen möglich sein, ist das Problem *unterbestimmt* oder *underconstrained.* Darüber hinaus werden Constraints als *hart* bezeichnet, wenn die Lösung nicht ohne sie auskommen kann, d.h. wenn dieselben erfüllt sein *müssen*. *Weiche Constraints* können dagegen vernachlässigt werden. Allerdings würde die zulässige Lösung eine höhere Qualität erhalten, wenn auch weiche Constraints erfüllt werden können.

Eine Markierung gilt zu einer anderen als benachbart, wenn sie durch eine Nachbarschaftsbeziehung oder auch Nachbarschaftsfunktion f_N in eine andere Markierung umgewandelt werden kann. Die Schritte, die dazu notwendig sind, werden mit s bezeichnet und sind damit gleichbedeutend zu der Entfernung zwischen den Markierungen.

Für das Beispiel 1 auf der Seite 4 kann eine weitere Lösung mit der Entfernung 1 gefunden werden. Denn der Variable C kann die Farbe Magenta zugewiesen werden, ohne das die anderen Variablen geändert werden müssen. Es wird also nur eine Änderung durchgeführt.

Viele einfache Constraintprobleme kommen mit einer Entfernung von 1 aus, um konsistente Nachbarn zu finden. Ein kleines s beschleunigt die Suche.

Das folgende „Traveling Salesman“ Problem hat zu jeder Nachbarlösung eine Entfernung von 1, wenn das Tauschen zweier Städte als eine Operation aufgefasst wird.

Beispiel 2:

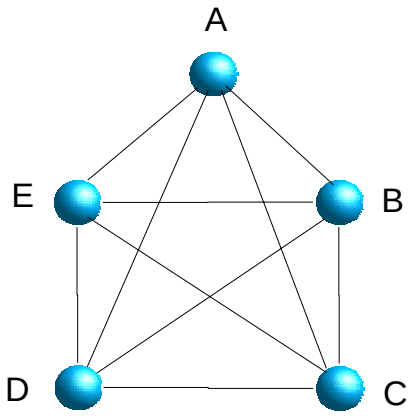


Abbildung 3 Traveling Salesman Problem im vollständigen Graphen

Markierung	1. Stadt	2. Stadt	3. Stadt	4. Stadt	5. Stadt
0	A	B	C	D	E
1	A	B	C	E	D
2	A	B	E	D	C
...					

Tabelle 1Traveling Salesman Problem

In [Barták 1998] werden Constraintprobleme nach ihrer Domaingröße klassifiziert. Besitzen Constraintvariablen finite Domains, so spricht man von *Constraint Satisfaction*. Kommen allerdings Domains zum Einsatz, deren Größe nicht vollständig begrenzt und somit unendlich ist, wird von *Constraint Solving* gesprochen. Weiterhin sind hier nichtlineare Zusammenhänge zwischen den Variablen erlaubt, die durch Constraints ausgedrückt werden. Die Methoden sind „mathematischer“, schließen automatische Differentiation und Taylor-Reihen-Entwicklung ein. Constraint Solver sind damit aufwendiger und komplexer, jedoch sind laut [Barták 1998] wahrscheinlich 95 % der industriellen Probleme durch Constraint Satisfaction zu lösen. Die Praxis nimmt von dieser Einteilung kaum Kenntnis, sodass hierauf nicht weiter eingegangen wird.

2.3 Konsistenzprüfungen und Constraintpropagierung

2.3.1 Knotenkonsistenz

Die folgenden Erläuterungen beziehen sich vornehmlich auf Constraintnetze oder zweistellige Constraints. In [Barták 1998] wurde gezeigt, dass jedes CSP in ein 2-stelliges CSP umgewandelt werden kann.

Eine Constraintvariable x wird als *knotenkonsistent* bezeichnet, wenn alle einstelligen Constraints erfüllt sind, die x beschränken. Folgerichtig wird dieser Zustand auch 1-konsistent (node consistent, NC) genannt [Heinsohn & Socher 1999].

Der folgende Algorithmus erzeugt in einem Constraintsnetz Knotenkonsistenz [Barták 1998].

Algorithmus 1 NC

```
procedure NC
  for each x in X do  //für jede Constraintvariable
    for each L in D(x) do  //für jedes Label der Variable x
      if any c(x(L)) is violated then //Verletzt L ein Constraint?
        remove L from D(x);  //entferne Label, falls inkonsistent
      endif
    endfor
  endfor
end NC
```

L bezeichnet ein Label einer Constraintvariablen $x \in X$ (Menge der Constraintvariablen).

Der Algorithmus untersucht jedes Label aller Constraintvariablen und entfernt gegebenenfalls ein Label, wenn ein einstelliges Constraint dadurch verletzt wird.

2.3.2 Kantenkonsistenz

Die Knotenkonsistenz betrachtet nur einstellige Constraints und vernachlässigt mehrstellige. Somit besteht die hohe Wahrscheinlichkeit, dass diese verletzt werden.

Beispiel 3:

Es seien die Variablen x und y gegeben mit D(x):={1; 2; 3} und D(y):={1; 3} sowie die Constraints $c_1(x) := x>1$, $c_2(x, y) := x>y$ und $c_3(y) := 2 \nmid y$. (zwei ist kein Teiler von y)

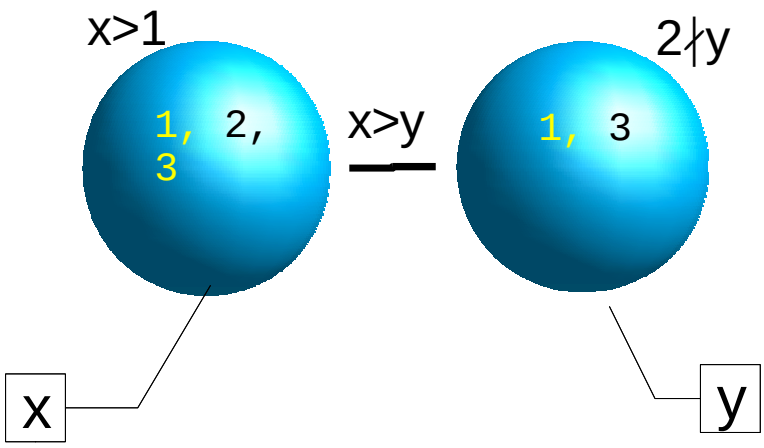


Abbildung 4 einfaches CSP

Offenbar ist y bereits knotenkonsistent, denn weder 1 noch 3 kann von 2 geteilt werden. Die Variable x wird nach dem Entfernen der 1 aus der dazugehörigen Domäne ebenfalls knotenkonsistent, sodass die Variablen mit D(x) = {2; 3} und D(y) = {1; 3} als 1-konsistent angegeben werden können. Es zeigt sich aber, dass $3 \in D(y)$ niemals zur Lösung beitragen kann, da hier das zweistellige Constraint c_2 verletzt wird.

Um eine gültige Lösung zu erhalten ist also mehr nötig, als Knotenkonsistenz herzustellen. Die Constraints, die die Constraintvariablen verbinden, müssen ebenfalls beachtet werden. Ein Constraintnetz, dessen mehrstellige Constraints erfüllt sind, heißt kantenkonsistent (arc consistent, AC und 2-konsistent).

Laut [Barták 1998] können in einem knotenkonsistenten CSP alle erfüllten einstelligen Constraints entfernt werden. Das ist auch einleuchtend, denn ein einstelliges Constraint verringert die Domain einer Variable, bis nur noch konsistente Werte enthalten sind. Wenn danach ein zweistelliges Constraint weitere Labels entfernt,

müssen sich immer noch konsistente Labels in der Domäne befinden. Sollte also eine Unstimmigkeit auftreten, dann kann dies nur durch die zweistelligen Constraints geschehen. Für die Konsistenzprüfung bedeutet dies, dass zunächst Knotenkonsistenz hergestellt werden muss, die einstelligen Constraints entfernt werden und danach die mehrstelligen Constraints untersucht werden können. Ein Algorithmus könnte folgendermaßen aussehen (vgl. [Barták 1998]):

Algorithmus 2 Revise

```
function revise(x, y)
  Deleted:=False
  for each L1 in D(x) do      //für jedes Label aus D(x)
    ConstraintViolated:=True
    for each L2 in D(y) do   //versuche ein passendes Label in D(y) ...
      if no Constraint c(x(L1), y(L2)) is violated then //... zu finden
        ConstraintViolated:=False
      endif
    endfor
    if ConstraintViolated then //falls kein passendes Label in ...
      remove L1 from D(x)      //... D(y) gefunden wurde ...
      Deleted:=True            //... entferne Label aus D(x)
    endif
  endfor
  return Deleted
end revise
```

Der Algorithmus revise beschränkt nur die Domain der ersten Variable. Zur Herstellung von Kantenkonsistenz zwischen zwei Constraintvariablen x1, x2 muss er zweimal aufgerufen werden:

```
revise(x1, x2)
revise(x2, x1)
```

Zur Verdeutlichung der Arbeitsweise soll das Constraint c_2 aus dem Beispiel 3 untersucht werden. Dazu wird davon ausgegangen, dass Knotenkonsistenz bereits hergestellt ist und der Wertebereich von x mit D(x) = {2; 3} und der von y mit D(y) = {1; 3} angegeben wird. Der Aufruf von revise(x, y) verursacht keine Änderung, denn zu dem Label x = 2 kann ein passendes Label aus D(y) gefunden werden. Und das Label x = 3 verletzt c_2 ebenfalls nicht. Dagegen verursacht revise(y, x) sehr wohl

eine Änderung:

1. L1=1
    1. L2=2
        1. ConstraintViolated:=False
    2. L2=3
        1. ConstraintViolated:=False
2. L1=3
    1. L2=2
    2. L2=3
    3. y=3 wird aus D(y) entfernt

In diesem Fall ist das Constraintnetz konsistent. Normalerweise muss revise solange ausgeführt werden, bis kein Wert mehr aus einer Domain entfernt wird, da sich die Constraints in einem Constraintnetz gegenseitig beeinflussen. Das Resultat dieser Überlegung ist der Algorithmus AC1 (vgl. [Barták 1998] oder [Heinsohn & Socher 1999]) führt.

Algorithmus 3 AC1

```
procedure AC1
  remove all single-digit constraints from constraintnet CN
  repeat
    Changed:=False
    for each Constraint C(x, y) in CN do
      Changed:= revise(x, y) or revise(y, x) //beide Funktionen ausführen
    endfor
  until not Changed
end AC1
```

Offensichtlich ist dieser Algorithmus nicht sehr effizient. Genauer gesagt hat er eine Zeitkomplexität von $O(n^3d^3)$ [Mackworth & Freuder 1993] wobei n die Anzahl der Variablen ist und d die maximale Domaingröße darstellt. Eine Änderung an einer Constraintvariable führt dazu, dass der gesamte Graph wiederholt durchsucht werden muss, obwohl nur ein gewisser Teil mit dieser Variable durch Constraints verbunden ist. Eine Änderung von AC1 geht auf diesen Umstand ein. Zunächst werden wieder alle einstelligen Constraints (nachdem sie erfüllt wurden) entfernt und daraus die Menge Q gebildet. Im Algorithmus AC3 wird Q durch eine Queue abge-

bildet. Q enthält nicht im eigentlichen Sinne zweistellige Constraints. Vielmehr werden in Q Variablenpaare gespeichert, die von einem Constraint betroffen sind. Um die korrekte Arbeitsweise zu gewährleisten müssen die Variablenpaare in beiden Reihenfolgen gespeichert werden. Für das Constraint c_2 aus dem Beispiel 3 hieße dies: Q:=(x; y),(y; x). AC3 entnimmt und entfernt aus Q ein Zahlenpaar (x, y) und übergibt es revise (Algorithmus 2). Sollte revise eine Änderung an der Domain von x vornehmen, werden Q alle Constraint-Variablenpaare hinzugefügt, die davon betroffen sein könnten; also (z, x). Wobei für die Variablen $z \neq x \wedge z \neq y$ gelten muss.

Formal stellt sich AC3 nach [Barták 1998] so dar:

Algorithmus 4 AC3

```
procedure AC3
  Q <--- put all arcs in the queue in both directions
  while Q not empty do
    Select and Delete an arc from the queue Q
      if revise(x, y) then
        Q:= Q append all (z, x) and z <> x and z <> y
      endif
  endwhile
end AC3
```

Dieser Algorithmus hat nur noch eine Zeitkomplexität von $O(n^2d^3)$ [Mackworth & Freuder 1993]. An dieser Stelle soll der Algorithmus AC3 genügen und von Ausführungen über den AC4 und folgenden abgesehen werden.

Oben wurde gezeigt, dass Knotenkonsistenz allein nicht genügt und nun stellt sich die Frage, ob Kantenkonsistenz allein ausreicht. Sie soll nach folgendem verändertem Beispiel 4 beantwortet werden.

Beispiel 4:

Das Beispiel 1 aus Kapitel 2.2 wird so abgeändert, dass nur noch zwei Farben zur Verfügung stehen. Somit lässt sich das Problem folgendermaßen beschreiben:

$D(A) = D(B) = D(C) = \{Rot; Blau\}$, $A \neq B$, $B \neq C$, $C \neq A$

Der AC3 (Algorithmus 4) nimmt keine Änderung an den Domains vor. Er überprüft die Länder A und B und findet keine Inkonsistenzen. Zwischen B und C, sowie C und A verhält es sich ebenso. Jedoch kommt die Inkonsistenz dann zum Tragen, wenn eine der Variablen auf ein Label festgelegt wird.

A wird rot also muss B blau sein. Und deshalb muss C rot sein wodurch A wiederum blau sein müsste, was aber nicht der Fall ist.

2.3.3 Pfadkonsistenz

„In einem Graphen (G= (V, E) ist ein Pfad p eine endliche Folge von Kanten.

$$p = (u_1, v_1), ..., (u_m, v_m)$$

$(u_i, v_i \in V$ mit $v_i = u_{i+1}$ mit $m \in \mathbb{N}_0$ und alle $i \in \{1, ..., m-1\})$“ [FH-Flensburg]

Die Kanten in einem Constraintnetz CN = (X, C) werden durch die zweistelligen Constraints dargestellt. Somit ist ein Constraintpfad *cp* eine Folge von zweistelligen Constraints, die n+1 Variablen miteinander verbinden. Die Länge des Pfades ergibt sich aus der Anzahl der Constraints, also *n*.

Man nennt einen Constraintgraphen *pfadkonsistent* (oder *k-konsistent*), wenn alle Werte aus den Domains $D(x_i)$ (mit $i \in \{1, ..., k-1\}$) alle zugehörigen Constraints erfüllen und es ein Label für eine k-te Variable gibt, sodass weiterhin alle Constraints zwischen diesen Variablen erfüllt sind [Barták 1998].

Ein Pfad ist *streng - k-konsistent*, wenn er für jedes $j \leq k$ j-konsistent ist.

Zur Einordnung der Konsistenztypen ist zu sagen, dass Knotenkonsistenz der 1-Konsistenz entspricht. 2-konsistent meint kantenkonsistent und für $k \geq 3$ spricht man von Pfadkonsistenz.

Algorithmen, die strenge Pfadkonsistenz herstellen, arbeiten sehr zeitintensiv und sind wenig praktikabel. So hat der PC-1-Algorithmus eine Zeitkomplexität von $O(n^3d^5)$ [Singh 1995]. Dieser Zeitaufwand ist für viele Probleme zu hoch, weshalb die Ansprüche an die Pfadkonsistenz etwas zurückgenommen werden müssen.

Ein Knoten x ist *eingeschränkt pfadkonsistent*, wenn Folgendes gilt:

Alle Constraints, die die Variable x direkt beschränken, sind erfüllt. Außerdem existiert zu jedem Label aus der Domain D(x) ein Label in der Domain einer benachbarten Variable y, sodass ein konsistentes Label zu einer anderen benachbarten Variable z gefunden werden kann. Zusätzlich müssen die Constraints über (x, z) und über (z, y) erfüllt sein. Das Constraint über (y, z) wird hier nicht berücksichtigt, obwohl es grundsätzlich anders definiert sein kann, als ein Constraint über (z, y)!

Glücklicherweise ist der Gebrauch eines PC-Algorithmus nicht notwendig. Die dynamische Anwendung eines AC-Algorithmus genügt, um inkonsistente Markierungen zu vermeiden. Dabei wird während der Suche einer Variablen ein Label

zugeordnet. Diese dynamische Anwendung von Konsistenzprüfungen wird hier Constraintpropagierung genannt. In der Literatur finden sich leider verschiedene Definitionen zur Constraintpropagierung, jedoch ist allen gemein, dass sich die Beschränkungen fortpflanzen. Ob nun alleinig durch einen Konsistenzalgorithmus oder durch den Verbund von Suche und Konsistenzprüfung kann der Literatur nicht entnommen werden. Die Praxis zeigt, dass zur Lösung eines CSP Konsistenzprüfungen allein nicht genügen.

2.4 Suchverfahren

2.4.1 Backtracking

Das Rückgrat der meisten Suchalgorithmen bildet zweifellos das Backtracking (kurz: BT). Dieser Algorithmus untersucht systematisch die Belegungen der Variablen. Dabei wird die Lösung Schritt für Schritt ausgebaut. Wird an einer Stelle ein inkonsistentes Label festgestellt, wird ein Rückschritt durchgeführt. Für ein Problem der Art (vgl. [Heinsohn & Socher 1999]):

$$x > y \wedge y > z$$
$$x, y, z \in \{0; 1; 2; ...; 10\}$$

kann aufgrund der transitiven Eigenschaft der Relation '>' sofort ausgesagt werden, das x > z sein muss und dass offenbar x = 0 kein gültiges Label sein kann. Backtracking hat davon aber keine Kenntnis und untersucht jedes Label systematisch:

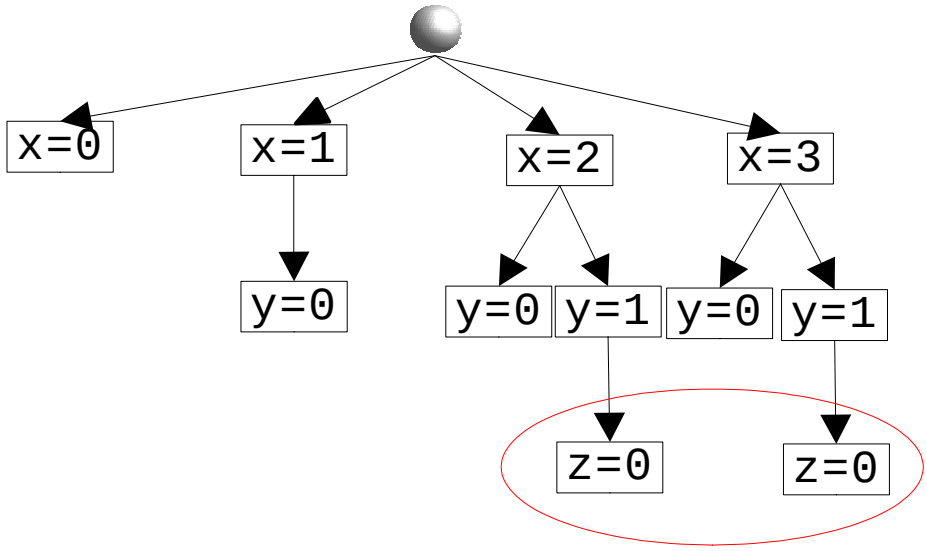


Abbildung 5 Backtracking

Für das geschilderte Problem könnte die Suche wie in Abbildung 5 dargestellt

werden. Allerdings ist der Suchbaum hier unvollständig und zeigt nur die ersten möglichen Lösungen, die von dem Kreis markiert werden. In welcher Reihenfolge Labels an Variablen gebunden werden, hängt von der Organisation der Variablen und deren Werte ab (siehe Kapitel 2.5).

Ein großer Nachteil des einfachen Backtrackings ist das *Thrashing* [Barták 1998]. Dabei gerät der Algorithmus immer wieder in dieselbe Sackgasse. In der Abbildung 5 wurde der Variable y zweimal das ungültige Label y = 0 zugewiesen, ehe eine gültige Lösung gefunden wurde.

2.4.2 Tiefensuche

Die Tiefensuche (depth-first search, DFS) legt für das Backtrackingverfahren eine Reihenfolge fest. So kann die Abbildung 5 auch durch Tiefensuche entstanden sein, wenn man die Grafik von links nach rechts liest.

Der DFS-Algorithmus untersucht zuerst die Knotennachfolger und dann deren

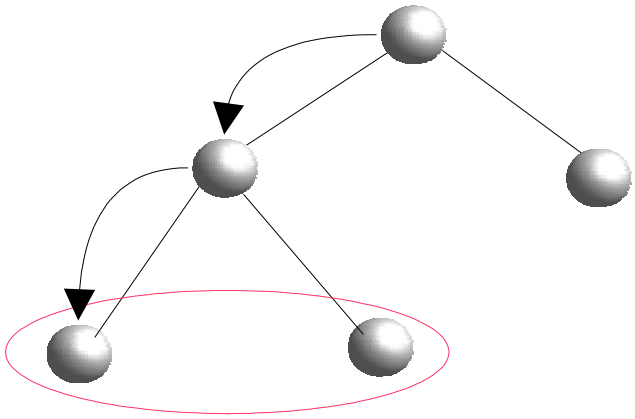


Abbildung 6 Tiefensuche

Nachfolger usw. Es geht also zunächst in die Tiefe des Suchbaumes, was auch den Namen erklärt. Die Implementierung bemüht meistens einen Stackspeicher, der die aktuellen ungetesteten Knoten beherbergt. Wird ein Knoten ausgewählt, so kommen seine Kinder auf den Stack und werden in einem anderen Schritt untersucht. Letztendlich befindet sich eine Lösung an einem Blatt und es ist sehr wünschenswert, so schnell wie möglich, dorthin zu gelangen. Für einige Problemstellungen kann dieses Vorgehen äußerst ungünstig ausgehen, wenn ein Ast des Suchbaumes eine extreme Tiefe besitzt und am Ende doch keine Lösung enthält. Der Suchbaum hat für ein Constraintproblem eine Tiefe von $|X|$, wobei X die Menge der Constraintvariablen ist.

Problematisch ist der Algorithmus nur dann, wenn er sehr früh einen Fehler macht und ihn erst sehr spät bemerkt (early mistake problem). Wenn man in der Abbildung 5, die auch durch Tiefensuche entstanden sein kann, das Label x = 1 betrachtet, stellt man fest, dass dieser Fehler erst durch y = 0 aufgedeckt wird, da kein z-Label mehr erlaubt ist.

2.4.3 Limited Discrepancy Search

Das oben beschriebene „early mistake“-Problem, das beim einfachen DFS-Algorithmus präsent ist, kann durch heuristische Verfahren, wie dem Limited Discrepancy Search (kurz: LDS) verhindert werden. LDS ist ein heuristisches Verfahren, vertraut aber nicht immer seiner Heuristik, sondern weicht auch davon ab. Diese Abweichung gibt dem Verfahren den Namen (Abweichung: engl. discrepancy). Dabei wird das Limit der Abschweifung schrittweise erhöht. Bei dem ersten Durchlauf wird keine Änderung erlaubt und die Heuristik befolgt. Die nächste Iteration untersucht alle Möglichkeiten mit einer maximalen Abweichung von 1, die übernächste lässt 2 Abweichungen zu usw. [Harvey & Ginsberg 1995]. Falls kein Ergebnis gefunden werden kann, wird trotzdem der gesammte Baum durchsucht.

2.4.4 Greedy Search

Wesentlich einfacher verhält sich der Greedy-Algorithmus. Er orientiert sich vollständig an seiner Heuristik und evaluiert zuerst den Knoten, der von der Heuristik ausgewählt wurde. Verglichen mit dem LDS arbeitet er nur eine Iteration ab. Die Effizienz ist sehr stark von der gewählten Heuristik abhängig. Anders als der LDS, findet dieser Algorithmus nicht jede Lösung, ist daher unvollständig und nur bedingt für Optimierungsprobleme geeignet.

2.4.5 Min-Conflicts-Heuristik

Die Min-Conflicts-Heuristik wählt zufällig eine Constraintvariable aus, die ein oder mehrere Constraints nicht erfüllt. (In der Praxis wird man dazu eine Konfliktmenge formulieren, in der genau die Variablen enthalten sind, die in irgend einer Weise Constraints verletzen. So ließe sich die Auswahl beschleunigen.)
Dann wird dieser Variablen ein Label zugeordnet, sodass so viele Constraints wie möglich erfüllt werden. Zumindest muss die Anzahl der Konflikte gleichbleiben. Falls mehrere Werte ein Konfliktminimum erzeugen können, wird eine zufällig bestimmte Wahl getroffen.

Leider kann sich diese Heuristik nicht aus einem lokalen Optimum befreien.

2.4.6 Hill-Climbing und Steepest-Descent Search

Der Hill-Climbing Algorithmus wählt zufällig eine unmarkierte Constraintvariable (Variable ohne Label) und bestimmt alle möglichen Belegungen dieser Variablen. Dann wird die Belegung gewählt, die den größten Gewinn bezüglich der Kostenfunktion verursacht. Vergleicht man den Suchgraphen mit einer Gebirgslandschaft, wird der Name des Algorithmus deutlich. Die Landschaft wird durch die Kostenfunktion beschrieben. Denn jedem Punkt kann mit seinem Höhenniveau eine bestimmte Güte und damit Kosten zugeordnet werden.
Der Algorithmus „wandert“ nur bergauf, wenn auch auf einem zufälligen Pfad und landet irgendwann auf einem Berggipfel. Meistens wird der Algorithmus in einem lokalen Optimum terminieren, weshalb man ihn in der Praxis an zufällig bestimmten Punkten mehrmals starten lässt.

Abgesehen von der Richtung arbeitet der Steepest-Descent Algorithmus ähnlich. Im Vergleich zu Hill-Climbing sucht er die nächste konsistente Markierung, die die beste Bewertung besitzt.
Der Steepest-Descent Algorithmus testet jede Variable und dessen Belegung und setzt schließlich das Label mit der größten Kostenverbesserung. Offensichtlich arbeitet dieser Algorithmus aufwendiger, als Greedy Search und Hill-Climbing, aber er führt meistens zu besseren Ergebnissen [DispatcherMan21].

2.4.7 Tabu Search

Anders als die oben vorgestellten Verfahren wirkt Tabu Search als Meta-Heuristik. Es leitet quasi eine lokale Suche. Da bei der Greedy-Suche immer nur der Schritt durchgeführt wird, der im Moment die beste Performanz hat, kann dieser Algorithmus schnell oszilliere

In Abbildung 7 soll der Greedy-Algorithmus die günstigste Lösung von Zustand A nach B bestimmen. Dabei kann A beispielsweise beim Rucksackproblem ein leerer und B ein voller Rucksack sein. Die einzelnen Knoten könnten eine bestimmte Konstellation der Rucksackfüllung darstellen.
Die Heuristik arbeitet hier so, dass sie immer den nächstgelegenen Knoten evaluiert. Dabei gerät sie in einen Kreis, der hier grün dargestellt ist. Es werden immer wieder die gleichen Knoten abgearbeitet. Tabu-Search führt zur Vermeidung dieses Problems eine Liste mit den letzten durchgeführten Aktionen ein. Alle Schritte, die sich

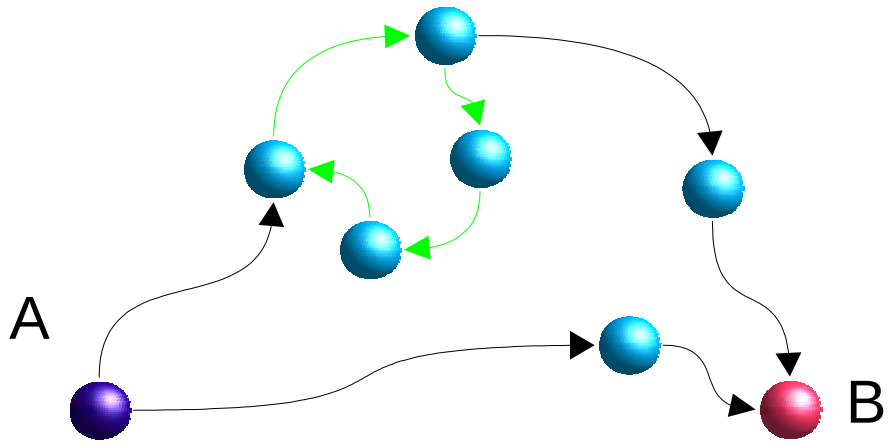


Abbildung 7 Zyklus im Greedy-Algorithmus

in der Tabu-Liste befinden, dürfen nicht verwendet werden. Da der Speicherplatz der Liste auf eine Zahl k beschränkt ist, fallen ältere Schritte irgendwann wieder heraus und stehen dem darunterliegenden Algorithmus zu Verfügung. Die Größe der Liste muss für jedes Problem durch empirische Untersuchungen bestimmt werden. Wird die Liste zu groß gewählt, werden zu viele Lösungen verboten, wird sie dagegen zu klein gewählt, besteht die Gefahr des Oszillierens. Tabu-Search kann sehr schnell sein und lokale Minima vermeiden. Leider ist es schwer, eine gute allgemeingültige Tabu-Größe zu finden.

2.4.8 Guided Local Search

Die Meta-Heuristik Guided-Local Search (kurz: GLS) „borgt“ sich die Idee der Bestrafung von den Algorithmen der Unternehmensforschung und kann durch geschicktes Manipulieren der Kostenfunktion einen lokalen Suchalgorithmus aus einem lokalen Optimum herausführen. Traditionell arbeitet der GLS mit der Hill-Climbing-Heuristik zusammen. Es ist aber denkbar, einen anderen lokalen Suchalgorithmus mit GLS zu kombinieren.

Die Heuristik definiert ein Feature als einen möglichen Bestandteil einer Markierung. Beispielsweise kann in einem TSP die Verbindung zwischen zwei Städten „fahre von A nach B“ ein Feature sein. (Dass dieses Feature nicht zwingend enthalten sein muss, wird durch die mögliche Reihenfolge A->C->B offensichtlich.)

Die Kostenfunktion von Hill-Climbing g(s) wird durch folgende Funktion h(s) ersetzt [Voudouris & Tsang 1998]:

$$h(s)=g(s)+\lambda \sum_{i=1,F} p_i \cdot I_i(s)$$

Hierbei ist s eine konsistente Markierung, λ ist ein regulierender Parameter von GLS, F ist die Anzahl der Features. Und p_i ist die Strafe für das Feature i. I_i ist ein Schalter für die Existenz des Features in der aktuellen Markierung.

$$I_i(s)= \begin{matrix} 1, \text{ falls Feature i in s enthalten ist} \\ 0 \text{ sonst} \end{matrix}$$

Die Idee von GLS ist es, die „schlechten Features“ in einem Optimum M (mit der Markierung s^*) zu bestrafen. Um die schlechten von den guten Features zu unterscheiden, werden Prioritäten bestimmt:

$$pri_i(s^*)=I_i(s^*)\cdot\frac{c_i}{1+p_i}$$

Der Wert für c_i stellt die Kosten des Features dar und könnte die Entfernung zwischen zwei Städten sein. Während der Suche werden die Features am häufigsten bestraft, die die höchste Priorität besitzen. Dabei wird p erhöht: $p_i = p_i+1$, wodurch die Priorität der Features wieder sinkt.

Somit lässt sich der Algorithmus wie folgt beschreiben [Voudouris & Tsang 1998]:

Algorithmus 5 GLS:

Gegeben: lokale Heuristik L mit der Kostenfunktion g

1. *Erzeuge eine konsistente Markierung, die als Startlösung fungiert.*
2. *Initialisiere die p-Werte mit 0.*
3. *Wiederhole, bis Abbruchkriterium (verstrichene Zeit, Anzahl der Iterationen ...) erreicht ist.*
   a) Arbeite L mit der Heuristikfunktion h solange ab, bis ein lokales Optimum M erreicht ist. s ist die lokal optimale Lösung.*
   b) Für jedes Feature, welches in M vorkommt wird die Priorität $pri_i(s^*)$ berechnet. (Dadurch ändert sich h!)
   c) Jedes Feature, welches die höchste Priorität besitzt, wird bestraft. $p_i=p_i+1$.
4. *Gib den besten Lösungskandidaten aus, der gefunden wurde.*

2.4.9 Branch-and-Bound

Sicher zählt das Branch-and-Bound-Verfahren mit zu den am stärksten diskutierten Algorithmen in der Constraintverarbeitung. Dabei stammt dieser Algorithmus aus dem Gebiet der Linearen Programmierung.

Hierbei wird angenommen, dass eine Zielfunktion zu minimieren ist und diese eine

berechenbare obere Schranke Z_U hat. Normalerweise nimmt Z_U den aktuellen Wert der besten zulässigen Lösung an. Jede weitere Lösung muss also besser sein als die *amtierende Lösung* [Hillier & Lieberman 1997].
Nach einer Initialisierung muss Z_U mathematisch gesehen, gegen unendlich streben. In der Praxis wird man Z_U meistens auf eine sehr große Zahl festlegen.
Im ersten Schritt (Branch) werden alle zulässigen Lösungen in einige Teilmengen aufgeteilt. Dies geschieht durch eine Verästelungsregel, die vom zu lösenden Problem abhängt. Die entstehenden Teilmengen müssen disjunktiv sein, um gemeinsame Lösungen zu verhindern. Im Bound-Schritt wird dann für jede Menge eine untere Schranke Z_L berechnet. Die untere Schranke zeigt quasi das Lösungspotential an. Ist Z_L sehr niedrig, scheint in dieser Teilmenge ein sehr gutes Minimum zu liegen. Nach dem Bound-Schritt folgt eine Sondierung der Teilmengen. Mengen, die keine besseren Lösungen enthalten können, werden ausgesondert.
Wenn also

1. $Z_L \geq Z_U$ oder
2. festgestellt wurde, dass die Teilmenge keine zulässigen Lösungen mehr enthalten kann oder
3. die beste mögliche Lösung der Teilmenge ausgemacht wurde (Z_L entspricht dem Zielfunktionswert),

wird die aktuelle Teilmenge entfernt. Danach wird mit einer neuen Menge begonnen.
Falls die Sondierungsregel 3 erfüllt wird und $Z_L < Z_U$ gilt, wird $Z_U = Z_L$ gesetzt und die amtierende Lösung durch die soeben gefundene abgelöst.
Wurde keine Lösung gefunden, muss ein neuer Branch-Schritt durchgeführt werden und dann wieder der Bound-Schritt usw.
Es sei hier angemerkt, dass im Unterschied zum Backtracking, der Suchzweig vor der eigentlichen Suche erstellt wird. Es muss also ein gewisser administrativer Aufwand betrieben werden, der aber dann relativ zielgerichtet zu einem Optimum führt.

2.5 Variablen-und Wertereihenfolge

Die oben beschriebenen Möglichkeiten lassen sich noch erweitern, um eine Lösung wesentlich schneller zu finden.
Man kann aufgrund der ursprünglichen Gleichverteilung von Variablen und Labels annehmen, dass die Optima in der Suchlandschaft ebenfalls gleichmäßig verteilt sind. Das ist natürlich ein Nachteil, da die Nachbarschaftsbeziehung f_N mehrere

Änderungen zulassen muss, um auf eine benachbarte konsistente Lösung zu wechseln. Viel günstiger wäre es, wenn nur genau eine Änderung eine neue zulässige Lösung hervorbringen könnte.

Wenn man nach einem Maximum sucht, lässt sich die Metapher von dem höchsten Gipfel gut anwenden. Demnach wäre ein Minimum ein Tal. Um nun zu erreichen, dass ein Berg nicht nur von Tälern, sondern von weiteren Bergen umgeben ist, kann man versuchen, die Variablen in eine bessere Ordnung zu bringen. Möglicherweise hilft das „First-Fail“ Prinzip, wobei die härtesten Fälle zuerst in die Suche einbezogen werden. Zum Beispiel könnte man vorab die Variable verwenden, die in den meisten Constraints involviert ist.

Nichtsdestotrotz können zwei Grundsätze bestimmt werden [Barták 1998]: *statische Reihenfolge* und *dynamische Reihenfolge*.

Die statische Reihenfolge legt vor der Suche ein einziges Mal eine Ordnung fest, wohingegen durch die dynamische Reihenfolge in jedem Suchschritt, abhängig vom aktuellen Status, die nächste Variable bestimmt wird.

Leider kann die dynamische Ordnung nicht mit jedem Suchalgorithmus kombiniert werden (insbesondere nicht mit den uninformierten Verfahren, wie Breiten- und Tiefensuche).

Eine weitere Möglichkeit, auf die Suche Einfluss zu nehmen, ist die Werteordnung. Hierbei werden die Labels der Variablendomains in solch eine Reihenfolge gebracht, dass schneller zu einer Lösung gefunden werden kann. So könnte man den Wert einer Variablen bevorzugen, der die wenigsten Constraintpropagierungen auslöst und entsprechend sortieren. Es gibt noch weitere Value-Ordering-Heuristiken, jedoch muss man feststellen, dass für zufällig generierte Probleme, eine Festlegung auf eine bestimmte Heuristik äußerst schwer fällt.

2.6 Constrainterweiterungen

2.6.1 Partielle CSP

Manche Constraintprobleme sind überbestimmt und in anderen Fällen steht zur Lösung des Problems zu wenig Zeit zur Verfügung. In solchen Fällen bietet es sich an, das Problem aufzuweichen oder abzuschwächen, sodass nur noch eine partielle Constrainterfüllung erreicht wird. Problematisch erweist sich die Art und Weise, in der die Abschwächung erreicht wird. Von den verschiedenen Möglichkeiten haben sich vier als bedeutsam erwiesen ([Barták 1998] und [Freuder & Wallace 1992]):

1. Vergrößere die Domäne einer Variable.
2. Weiche ein Constraint auf (statt x < 3 verwende x < 6).
3. Entferne eine Variable aus der Problematik.
4. Entferne ein Constraint.

Um sich nicht zu sehr von dem eigentlichen Problem zu entfernen, wird eine Distanzfunktion definiert. Sie legt die Constraints fest, die unbedingt erfüllt werden müssen und definiert gleichsam die Nähe zum eigentlichen Problem.
Laut [Guesgen 1997] werden der Partiellen Constrainterfüllung verschiedene Teilbereiche zugeordnet. Dadurch werden hierarchische Constraints (Kapitel 2.6.2) und Fuzzy Constraints (Kapitel 2.6.3) diesem Thema ebenfalls zugeschrieben.

2.6.2 Constrainthierarchien

Constrainthierarchien werden ebenfalls genutzt, um überbestimmte Constraintprobleme zu lösen. In gewissem Sinne sind sie mit partiellen CSP's vergleichbar oder als deren Bestandteil anzusehen, allerdings werden hier lediglich Constraints entfernt, Variablen verbleiben im Problem. Um zu entscheiden, welche Beschränkungen entfernt werden können, um dennoch eine Lösung zu erhalten, werden die Constraints zu Kategorien zusammengefasst oder mit Prioritäten versehen. Das Entfernen von Constraints mit niedrigen Prioritäten, verändert das eigentliche Problem am Wenigsten. Kann nach dem Löschen der Constraints mit der geringsten Priorität noch immer keine Lösung gefunden werden, sind im nächsten Schritt Constraints höherer Priorität aus dem Problem zu entlassen.

2.6.3 Fuzzy Constraints

In Fuzzy-CSP's werden unscharfe Constraints zugelassen. Ein Constraint wird vollständig gelöst, wenn die Mitgliedsfunktion (*membership function*: f_{MS}) des Constraints eine 1 liefert. Wie in der Fuzzy-Logik auch, dürfen die Funktionswerte nur zwischen 0 und 1 liegen, wobei eine Null bedeuten würde, dass das Constraint überhaupt nicht erfüllt wird. Somit setzt sich ein Fuzzy-CSP aus einem generischen Constraint Satisfaction Problem (bestehend aus Constraintvariablen V und Constraints C) sowie n Mitgliedsfunktionen zusammen, welche n Constraints einen eindeutigen Funktionswert zuweisen.
Die Lösung eines CSP's ist eine Markierung, die alle Constraints voll erfüllt, wohingegen die Lösung eines Fuzzy-CSP's eine Markierung ist, die die Summe aller

Funktionswerte von $f_{MS}(c_i)$ maximiert.

2.6.4 Probabilistische Constraints

Probabilistische Constraint Satisfaction Probleme (kurz: Prob-CSP) zeichnen sich dadurch aus, dass jedes Constraint $c_i \in C$ mit einer Wahrscheinlichkeit $\rho(c_i)$ belegt wird. Dadurch wird ausgedrückt, dass die Situation mit einer Wahrscheinlichkeit ρ eintritt in der das Constraint c_i wirkt. Anders formuliert tritt mit der Wahrscheinlichkeit ρ die Beschränkung c_i ein. Die Wahrscheinlichkeiten geben Auskunft, inwieweit eine konsistente Markierung die Lösung eines wirklichen Problems darstellt. Man kann nun entscheiden, inwieweit unwahrscheinliche Constraints berücksichtigt werden. Es mag die Problematik nicht übermäßig verändern, wenn man ein Constraint mit einer geringen Wahrscheinlichkeit entfernt und somit das Problem einfacher gestaltet.

2.7 Trends

Laut [Barták 1999] scheint das Modellieren von Constraintproblemen mit das wichtigste Entwicklungsthema zu sein. Danach werden globale versus lokale Constraints diskutiert.

Auf der ECAI 1996 (European Conference on Artificial Intelligence) befasste sich etwa die Hälfte der Vorträge mit Modellierung, wobei sich davon ein Großteil den Constraint Hierarchien zuwandte [Stützle 1996]. Auf der IJCAI 2001 (International Joint Conference on Artificial Intelligence) waren nur noch zwei von neunzehn Vorträge zum Thema „Suche, Erfüllbarkeit und Constraint Erfüllbarkeitsprobleme“, die das Modelling als wesentlichen Bestandteil zum Thema haben.

Toby Walsh [Walsh 2001] untersucht darin hochgradige Knoten in Suchnetzen und stellt fest, dass in zufällig generierten Netzwerken hochgradige Knoten kaum auftauchen, in konkreten Problemen aber sehr häufig zu finden sind. Er untersucht den Einfluss des Knotengrades auf das Graphenfärbeproblem (Beispiel 1). Weiterhin wird von Jean-François Baget und Yannic S. Tognetti [Baget & Tognetti 2001] versucht, den Backtrackingalgorithmus durch eine veränderte Darstellung des Problemgraphen zu verbessern. Der BCC-Algorithmus (BCC – Biconnected Component of a Constraint Graph) kann dabei auf globales Constraintwissen zurückgreifen. Wurden sonst vornehmlich lokale Informationen verwertet, kann nun globales Wissen benutzt werden.

In der Praxis liegt der Fokus der Entwicklungsarbeit ebenfalls in der Modellierung, jedoch mit einem anderem Ziel. Verschiedene Firmen versuchen Constraintlöser in

bestehende Programmiersprachen zu integrieren. Dabei sind naturgemäß Constraint-Solver in den logikbasierten Sprachen am stärksten vertreten. Laut [Frühwirth & Abdennadher 1997] gehören zu den ersten Constraint-Logic-Programmiersprachen CLP(R), CHIP und Prolog III.

Inzwischen werden auch hybride Ansätze untersucht und seit spätestens 2001 industriell verwendet. Mathematische Programmierung und Constraintprogrammierung haben beispielsweise in den ILOG Bibliotheken Solver 5 und CPLEX 7 durch Concert 1 eine gemeinsame Schnittstelle gefunden. Hybride Algorithmen kombinieren Techniken der Künstlichen Inteligenz und der Unternehmensforschung. In [Rodosek & Wallace 1998] wurde gezeigt, wie ein „Hoist Scheduling Problem“ effizient durch den Einsatz zweier Ansichten gelöst werden kann. Dabei wurde ein generisches Modell entworfen, welches dann in ECL^{i}PS^{e} durch Mixed-Integer-Programming (kurz: MIP; wird im Operations Research angewendet) und Constraint-Logic-Programming gelöst wurde. Auf diese Weise können sich beide Verfahren gegenseitig „befruchten“ und ihre Performanz steigern.

Geschwindigkeitssteigernd wirkt sich auch parallele Verarbeitung aus. Jedoch ergeben sich aus der parallelen Abarbeitung weitere Konsistenzprobleme, die berücksichtigt werden müssen. Eine einfache Zergliederung des Constraintnetzes hilft hier nicht weiter. Die Inkonsistenzen, die dabei entstehen, müssen über die Netzgrenzen hinweg aufgedeckt werden. In [Baumgärtel 1997] wird ein erfolgreiches Verfahren beschrieben, welches ein Constraintnetz zerlegt und die so entstandenen Teilprobleme löst. Die Lösung aller Teilprobleme konnte zu einer Gesamtlösung kombiniert werden.

Seit Anfang 1990 sind Bestrebungen in Gange, Constraints in Relationale Datenbanken zu integrieren [Revesz 1998]. Dabei werden Constraints als Basisdatentypen angesehen. Die Abfragesprache wird erweitert und ermöglicht umfassendere Abfragen wie: „Gib mir einen Termin für ein Meeting, an dem Person A und Person B teilnehmen können.“ Die möglichen Termine sind dabei jeweils für beide Personen gespeichert und werden auf Konsistenz überprüft.

Die Bilanz der kommerziellen Lösungen ist beachtlich. Ende der achtziger Jahre waren bereits industrietaugliche Solver verfügbar. Laut [Frühwirth & Abdennadher 1997] wird seit Anfang der neunziger Jahre Constraintprogrammierung kommerziell eingesetzt. Der Umsatz mehrerer weltweit agierender Firmen, wurde 1996 auf 100 Millionen US-Dollar geschätzt.

2.8 Zusammenfassung

Kurz gesprochen, lassen sich mit der Hilfe von Constraints Bedingungen festlegen, die in der Lösung zementiert sein müssen.

Der vorhergehende Abschnitt hat gezeigt, dass Constraints in vielen Bereichen eine Rolle spielen können. Constrainttechniken werden in Datenbanken, oder auch bei der Verarbeitung der natürlichen Sprache verwendet. Sie können dazu beitragen, bestimmte Problemklassen, die sehr schwer zu bewältigen sind (NP-vollständige Probleme), schneller zu lösen, denn sie verkleinern wirksam den Suchraum.

3 Alternativen zur Constraintprogrammierung

In [Tsang u.a. 1999] wird auf die enge Verbindung zwischen den Forschungsgebieten des Operation Research (kurz: OR; Unternehmensforschung) und der Künstlichen Intelligenz im Bereich der Kombinatorischen Optimierung hingewiesen. Beide Gebiete haben voneinander profitiert obwohl ihnen unterschiedliche Ansätze zugrunde liegen.

An dieser Stelle soll nur kurz auf die Lineare Programmierung und das Simplex-Verfahren eingegangen werden. Lösungsansätze zur Simulation wie das simulierte Ausglühen, das im Abschnitt 3.2 vorgestellt wird, führen in bestimmten Bereichen zu sehr guten Ergebnissen. Im Abschnitt 3.3 werden Genetische Algorithmen vorgestellt.

3.1 Lineare Programmierung

In einigen Fällen lässt sich aus den praktischen Problemen ein mathematisches Modell entwickeln. Eine Ansammlung von Gleichungen und Ungleichungen dient der Beschreibung des Problems und stellt das Modell dar. Gewissermaßen bezeichnet die Mathematische Programmierung das Erstellen eines Modells.

Sollte es nur lineare Gleichungen und Ungleichungen umfassen, können die Mittel der Linearen Programmierung angewandt werden. Sollte dies aber nicht möglich sein, müssen weiterführende Methoden, wie die der Quadratischen Programmierung verwendet werden. Diese würde zumindest quadratische Terme in der Zielfunktion zulassen. Modelle mit nichtlinearen Funktionen sind weitaus komplizierter zu handhaben, sodass man versucht, nur mit linearen Funktionen auszukommen.

In einigen Fällen wird ein ganzzahliges Ergebnis verlangt, welches mit Hilfe der Linearen Programmierung nicht immer gefunden werden kann. Tritt dies auf, werden die Methoden der Integerprogrammierung angewandt.

Hier soll nur auf das Simplex-Verfahren eingegangen werden, welches auf dem Gebiet der Unternehmensforschung eine fundamentale Bedeutung erlangt hat. Eine große Anzahl wichtiger Probleme können durch die Simplex-Methode gelöst werden.

Den Anfang des Linearen Programmiermodells macht die Zielfunktion:

$$Z = c_1x_1 + c_2x_2 + ... + c_nx_n,$$

die im Laufe der Lösungssuche maximiert wird. Jede Variable x_j der Zielfunktion hat einen Koeffizienten c_j. Die Anzahl der Variablen wird hier mit n angegeben. Eine Reihe von Nebenbedinungen, die als lineare Gleichung oder Ungleichung ausgeprägt sind, beschränken den Lösungsraum.

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \leq b_1$$

$$a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \leq b_2$$

.

.

.

$$a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n \leq b_m$$

$$x_1 \geq 0, x_2 \geq 0, ..., x_n \geq 0$$

Das Modell besitzt m Funktionen, wodurch sich der Index der Konstanten b_i ($i \in \{1; ...; m\}$) und a_{ij} ($i \in \{1; ...; m\}$ und $j \in \{1; ...; n\}$) ergibt. Der Koeffizient a_{23} steht somit vor der Variable x_3 und gehört der zweiten Funktion an zu der auch b_2 gehört.

Die Variablen x_j nennt man *Entscheidungsvariablen* des Systems und die Konstanten a_{ij}, b_i, c_j sind die *Parameter* des Modells. Dabei finden sich für die ersten m Ungleichungen in [Hillier & Lieberman 1997] häufig der Begriff der *funktionalen Nebenbedingungen*. Die letzten n Ungleichungen ($x_1 \geq 0$, $x_2 \geq 0$ ...) sind die *Nichtnegativitätsbedingungen*.

Je nach Modell können verschieden geartete Lösungen gefunden werden. Im Kontext der Linearen Programmierung wird jede Belegung der Variablen als Lösung betrachtet, unabhängig davon, ob eine Nebenbedingung verletzt wird oder nicht. Laut [Hillier & Lieberman 1997] ist eine *zulässige Lösung* eine Variablenbelegung, die alle Nebenbedingungen einhält. Eine *optimale Lösung* weist darüber hinaus einen maximalen Wert der Zielfunktion auf.

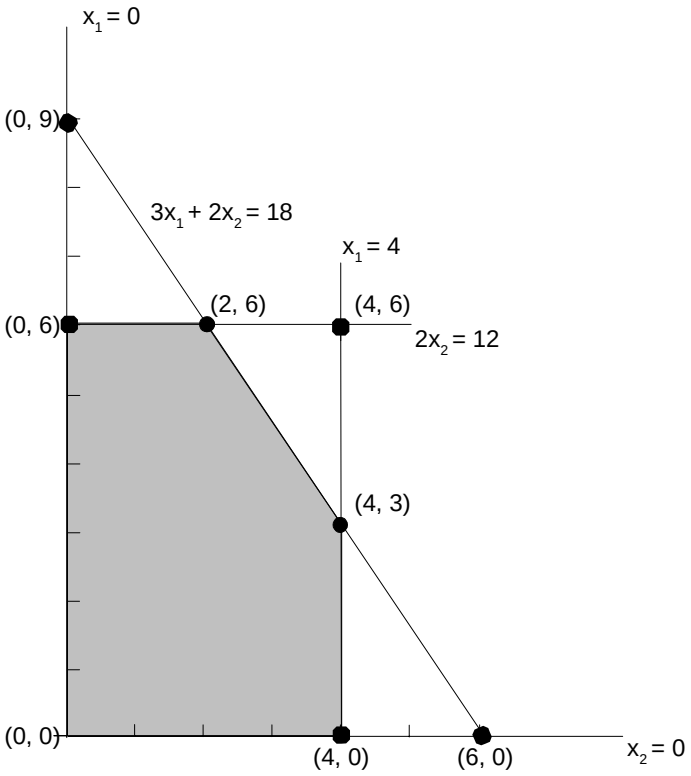


Abbildung 8 Simplex (aus [Hillier & Lieberman 1997])

Betrachtet man ein Modell mit zwei Variablen, lässt sich leicht eine geometrische Deutung finden, bei der der Lösungsraum durch eine mehr oder weniger stark deformierte Figur dargestellt werden kann. Der grau gezeichnete konvexe Bereich in Abbildung 8 wird Simplex genannt. Eine *zulässige Eckpunktlösung* (z.B. (4, 3)) ist eine zulässige Lösung, die sich an einer Ecke des Lösungsraumes befindet. Dagegen befindet sich eine *unzulässige Eckpunktlösung* (z.B. (4, 6)) auf einem Schnittpunkt zweier Geraden außerhalb des Lösungsraumes. Es gelten zwei *zulässige Eckpunktlösungen* als *benachbart,* wenn sie durch eine Begrenzungslinie verbunden sind.

In [Hillier & Lieberman 1997] werden weitere Eigenschaften genannt, die die wichtigsten Grundlagen für das Simplexverfahren bilden:

1. Wenn genau eine optimale Lösung existiert, ist es eine zulässige Eckpunktlösung.
2. Existieren mehrere optimale Lösungen, sind mindestens zwei von ihnen benachbart.

3. Es gibt nur eine endliche Zahl von zulässigen Eckpunktlösungen.
4. Ist eine zulässige Eckpunktlösung gleich oder besser als *alle benachbarten* Eckpunktlösungen, dann ist sie auch gleich oder besser als *alle anderen* Eckpunktlösungen.

Diese vier Eigenschaften werden von dem Simplexverfahren genutzt. Die Eigenschaft 1 verwendend, müssen nur Eckpunktlösungen betrachtet werden. Die Eigenschaft 3 besagt, dass nur eine endliche Zahl von Lösungen untersucht werden muss. Somit wird folgende Vorgehensweise möglich [Hillier & Lieberman 1997]:

1. Initialisieren mit einer zulässigen Eckpunktlösung.
2. Wiederhole bis Abbruchbedingung erreicht
a) Gehe zu einer besseren benachbarten Eckpunktlösung.
b) Optimalitätstest.

Der Optimalitätstest nutzt die Eigenschaft 4. Wird keine bessere benachbarte Eckpunktlösung gefunden, muss die aktuelle eine optimale Lösung sein.

Das Simplexverfahren benötigt zur besseren Handhabung die Umwandlung der Ungleichungen in Gleichungen. Dazu werden sogenannte Schlupfvariablen (slack variables) eingeführt. Aus einer Ungleichung mit nur einer Variable $x_i \leq b_j$ wird $y_j = b_j - x_i$. Eine weitere Umformung ergibt $x_i + y_j = b_j$. Die Gleichungen und Ungleichungen sind äquivalent, wenn $y_j \geq 0$ gilt. Entsprechend kann mit den Ungleichungen mit mehreren Variablen umgegangen werden. So wird mit jeder Ungleichung verfahren, bis das alle Ungleichungen umgestellt ist. Die Zielfunktion wird ebenfalls verändert, sodass sich eine angepasste Form des Schemas ergibt:

$$
\begin{aligned}
Z - c_1x_1 + c_2x_2 + \ldots + c_nx_n &= 0 \\
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n + y_1 &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n + y_2 &= b_2 \\
&\vdots \\
a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n + y_m &= b_m
\end{aligned}
$$

$$x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m \geq 0$$

Diese neue Form (*Gleichungsform*) bringt einige neue Bezeichnungen mit sich [Hillier & Lieberman 1997]:

Eine *erweiterte Lösung* ist eine Lösung mit Schlupfvariablen. Eine *Basislösung* ist

eine erweiterte Eckpunktlösung und eine *zulässige erweiterte Basislösung* ist eine erweiterte zulässige Eckpunktlösung. Eine *Nichtbasisvariable* ist eine Variable, die auf Null gesetzt wurde. Die übrigen Variablen werden *Basisvariablen* genannt. Sind alle Basisvariablen nicht negativ, liegt eine zulässige Basislösung vor. Zwei zulässige Basislösungen befinden sich in Nachbarschaft, wenn sie sich lediglich um eine Variable unterscheiden. Somit sind gute Voraussetzungen für eine Abbruchbedingung geschaffen.
Das Simplex-Verfahren arbeitet ähnlich dem Gauß-Jordan-Eliminationsverfahren. Dafür wird die Gleichungsform in eine Matrix-Form überführt. Da n + 1 Variablen und m + 1 Gleichungen vorhanden sind, existieren (bei der Annahme, dass n > m) n - m Freiheitsgrade. Somit lassen sich einige Variablen beliebig wählen, solange der Wert der Variablen größer Null ist. Das Verfahren startet mit der Initialisierung, die eine erste Lösung bestimmt. Meistens können die Schlupfvariablen als Basisvariablen festgelegt werden. Eine erste zulässige Lösung wäre, bei nur zwei Entscheidungsvariablen und der geometrischen Deutung des Problems, der Koordinatenursprung.
Während eines Iterationsschrittes tauschen eine Nichtbasisvariable und eine Basisvariablen die Plätze und ändern dabei jeweils ihren Status von Nichtbasisvariable auf Basisvariable und von Basisvariable zu Nichtbasisvariable. Die Wahl fällt auf die mit dem größten Koeffizienten, sodass man annehmen kann, dass sich der Wert der Zielfunktion am stärksten vermehrt. Der Optimalitätstest prüft, ob durch einen weiteren Schritt mit einer beliebigen Nichtbasisvariablen eine Verbesserung des Ergebnisses herbeigeführt werden kann. Ist dies nicht der Fall endet das Verfahren.

So direkt das Simplex-Verfahren auch arbeitet, so schwierig ist es, ein geeignetes Modell für das reale Problem zu finden. Die Nebenbedingungen müssen linear sein und in eine Funktion hineinformuliert werden können. Dies macht die Bearbeitung zwar sehr effizient, schränkt den Programmierer in der Freiheit des Modellentwurfs aber stark ein. Außerdem lassen sich gewisse Probleme nicht immer durch lineare Funktionen ausdrücken. Oft müssen nichtlineare Zusammenhänge beschrieben werden. Durch stückweise lineare Approximation wird das Modell aber sehr unhandlich und zu ungenau. Auch können logische Funktionen nur durch ganzzahlige 0/1-Ausdrücke repräsentiert werden, was sich nur mit Hilfe der Integerprogrammierung lösen lässt.

3.2 Simulation

Da mathematische Modelle oft sehr komplex sind, entstand die Idee, den Problemfall zu simulieren. Für eine Simulation wird das Problem in logische Komponenten zerlegt, sodass es möglich ist, bestimmte Reaktionen auf Ereignisse vorherzusagen. Die Zusammenführung der einzelnen Komponenten des Modells entspricht der Simulation des Gesamtsystems. Das Auftreten bestimmter Ereignisse und Beobachtungen wird einer Wahrscheinlichkeitsverteilung unterworfen. So sind Simulationen über Verkehrsfluss mit Ampelanlagen vorstellbar beziehungsweise bereits umgesetzt worden. Auch die Simulation umfangreicher Lagerhaltungssysteme wurde realisiert.

Aus der Simulation eines physikalischen Prozesses ist eine Heuristik entstanden, die eine große Bedeutung Mitte der Achtziger Jahre erlangte: das *Simulated Annealing* (Simulierte Ausglühen).

Sie nutzt dabei die Simulation des Abkühlens von Metall und den Metropolis-Monte-Carlo-Algorithmus. Beim Übergang vom flüssigen in den festen Aggregatzustand bilden die Metallatome Gitterstrukturen aus. Eine perfekte Anordnung zeigt sich in einer niedrigen Energiebilanz, die hier erwünscht ist. Eine gestörte Anordnung beeinträchtigt die Energiebilanz.

In der Werkstoffphysik wird für gute Ergebnisse das Metall langsam abgekühlt, was zur Folge hat, dass sich die Atome noch lange bewegen können und nur langsam in ihrer Mobilität eingeschränkt werden. Die Boltzmann-Verteilung

$$\frac{n_j}{N} = \frac{g_j}{g_0} \mathrm{e}^{\frac{-E_j}{kT}}$$

mit

- n_j Teilchenzahl im j-ten angeregten Zustand,
- N Teilchengesamtzahl,
- g_j Entartungsgrad,
- g_0 Statistisches Gewicht des Grundzustandes,
- E_j Anregungsenergie des j-ten angeregten Zustandes und
- T Temperatur

bestimmt die Besetzung der Energieniveaus der Atome. Damit sind obere Energieniveaus weniger wahrscheinlich besetzt als niedrigere. Aus dieser Wahrscheinlichkeitsverteilung ergibt sich die Besetzungsdichte der Energieniveaus. Außerdem verhält sich die Wahrscheinlichkeit, dass im Metallkristall ein bestimmter Gitterplatz

nicht besetzt ist, zum Boltzmannfaktor $e^{\frac{-E_j}{kT}}$ proportional.

Diese Überlegungen wurden in [Kirkpatrick u.a. 1983] mit dem Metropolis-Algorithmus in Verbindung gebracht. In jeder Iteration wird für ein Atom eine Änderung der Elektronenpositionen vorgegeben. Ist die Energieänderung im Vergleich zu der vorigen Elektronenposition kleiner oder gleich Null, wird sie akzeptiert. Sollte eine Änderung den Anstieg der Energiebilanz bedeuten, wird eine Zufallszahl im Intervall von [0, 1] erzeugt und mit dem Boltzmannfaktor verglichen. Genauer gesagt, wird

$$P(\Delta E) = e^{\frac{-\Delta E}{kT}}$$

berechnet. Sollte die gefundene Wahrscheinlichkeit größer sein als der Zufallswert, wird die neue Konfiguration akzeptiert. Diese Schritte werden immer wieder ausgeführt. Allmählich wird die Temperatur verringert, sodass das Metall letztendlich erstarrt.

Bezogen auf die kombinatorische Optimierung wird die Temperatur als Parameter verwendet. Die Werte für die Energiebilanzen werden aus einer Kostenfunktion gewonnen. Außerdem wird die Verteilungsfunktion als Akzeptanzfunktion bezeichnet. Der gesamte Algorithmus sieht damit wie folgt aus:

Algorithmus 6 Simulated Anealing

1. *Erzeuge eine zufällige neue Konfiguration*
2. *Wenn $E_{neu} \leq E_{alt}$*
   a) Ersetze alte Konfiguration durch die neue
   b) Gehe zu 1
3. *Wenn $E_{neu} > E_{alt}$ generiere Zufallszahl z im Intervall [0, 1]*
   a) Wenn $z < P(E_{neu}-E_{alt})$
   - ersetze alte Konfiguration durch die neue
4. *Wenn irgendein Ereignis eintritt, verringere den Parameter T*
5. *Wenn irgendein Ende-Kriterium erreicht ist, beende den Algorithmus.*
6. *Gehe zu 1.*

Diese Heuristik vermag aus einem lokalem Minimum wieder herauszufinden und die Suchgeschwindigkeit kann auf Kosten der Ergebnisqualität durch ein schnelleres „Abkühlen“ erhöht werden.

Ist ein Maximum statt eines Minimums notwendig, wird das Ergebnis der Kostenfunktion schlicht negiert.

3.3 Evolutionäre Algorithmen

In der Spiegelausgabe 23/98 meinte der Computerspielentwickler von „Creatures“: „Die Natur hat uns 3,7 Milliarden Jahre Forschungszeit voraus. Wir wären dumm, wenn wir versuchten, etwas anderes aus dem Boden zu stampfen, das auch nur annähernd so interessant ist.“ Die Evolution (zumindest in Teilen) nachzuahmen und die langjährige Forschungszeit zu nutzen, ist erklärtes Ziel evolutionärer Verfahren. Genetische Algorithmen implementieren grundlegende Prinzipien der Evolution. Genauer gesagt werden Selektion, Rekombination und Mutation als Grundlage betrachtet.

Der Algorithmus startet mit einer Population von möglichen Lösungen, wobei eine Lösung als Zeichenkette repräsentiert werden kann. Sie wird hier als Chromosom bezeichnet. Die Menge aller Chromosomen, die Population, wird in eine neue Population umgewandelt, indem einzelne Chromosomen der Mutation unterworfen werden oder zwei Chromosomen durch Rekombination Nachkommen (er-)zeugen. Die Häufigkeit und Stärke der evolutionären Maßnahmen wird durch äußere Parameter und den Fitnesswert (gibt Auskunft über die Güte einer Lösung) der Chromosomen festgelegt. Die Funktion, die den Fitnesswert bestimmt, übernimmt somit die Selektion. Nur die Besten sollen überleben. Getrieben von der Hoffnung, dass sich die Fitness der Chromosomen und damit die Güte der Lösungen von Population zu Population verbessert, stellt man als Abbruchkriterium eine gewisse Generation ein. Beispielsweise könnte die tausendste Population die letzte sein. Da nur die Besten überleben, muss sich folgerichtig in der letzten Generation der bis dahin beste Lösungswert befinden.

Lösungen oder Chromosomen können im Bereich der Genetischen Algorithmen nicht falsch, höchstens sehr schlecht sein. Es ist nicht möglich, harte Constraints direkt einzubeziehen. Statt dessen behilft man sich damit, einem entsprechenden Chromosom einen besonders schlechten Fitnesswert zuzuordnen. Beispielsweise würde beim Erraten der Zahl einhundert die Lösung drei besonders schlecht sein. Eine neunundneunzig wäre dann schon ziemlich gut, doch in Wirklichkeit noch immer falsch.

Bei dem hier betrachteten Problem können Lösungen, die beinahe richtig sind, nicht akzeptiert werden. Beispielsweise wird es schwierig Tomatenkisten auf einen Betonfahrmischer zu verladen. Mag sein, dass solche Lösungen sehr schlechte Fort-

pflanzungschancen haben, doch können sie bei sehr schlechten Populationen nicht ausgeschlossen werden.

3.4 Dynamische Programmierung

Die Vorteile des Teile-und-Herrsche-Prinzips versucht die Dynamische Programmierung weiter auszubauen. Das Problem wird in mehrere Teile zerlegt und anschließend versucht, unter Einbeziehung vorheriger Ergebnisse kleinerer Teile, das Gesamtproblem zu lösen. In bestimmten Fällen wird zunächst ein optimales Ergebnis für den Ausschnitt eines Problems gesucht. Dann wird ein größer Ausschnitt gewählt und dessen Problem unter Einbeziehung der früheren Ergebnisse gelöst. Dabei wird laut [Hillier & Lieberman 1997] eine vollständige Enumeration vermieden.

3.5 Zusammenfassung

Die verschiedenen Ansätze haben ihre Berechtigung und sind sehr erfolgreich. Doch kann lineare Programmierung schnell sehr aufwendig und damit teuer werden. Das Validieren des Simplex-Modells sollte keinesfalls unterschätzt werden. Wird mit einem falschen oder zu ungenauen Modell gerechnet, zeigen sich Fehler unter Umständen erst, wenn die Lösung in der Realität erprobt wird. Ebenso muss das Modell, das zur Simulation herangezogen wird, genau geprüft werden. Oftmals wird dieser Bereich der Unternehmensforschung als Experimentierlabor bezeichnet, lassen sich doch bestimmte Konstellationen ausprobieren und testen. Die evolutionären Algorithmen werden immer ernster genommen. Beispielsweise wurde 1996 berichtet [c't 1996], dass die Universität Hildesheim beabsichtige, die Fahrpläne der Deutschen Bundesbahn mit genetischen Algorithmen zu optimieren.

Möglicherweise kann man noch auf völlig andere Optionen hoffen. Ein Quantencomputer könnte Aufgaben in einer Zeit linear zur Größe des Eingabevektors bewältigen, für die ein von-Neumann-Computer exponentiell viel Zeit benötigt. Denn ein Qubit, der kleinster Informationsträger des Quantencomputers, kann mehrere Zustände zugleich annehmen. Dadurch sind Aussagen wie: *„Factoring a number with 400 digits - a numerical feat needed to break some security codes - would take even the fastest supercomputer in existence billions of years. But a newly conceived type of computer, one that exploits quantum-mechanical interactions, might complete the task in a year or so, thereby defeating many of the most sophisticated encryption schemes in use.“* [sciam 1998] möglich. Bezogen auf kombinatorische Optimie-

rungsprobleme wäre der Vorteil enorm. Man könnte immer ein optimales Ergebnis liefern und müsste sich nicht mit einem beinahe optimalen Ergebnis zufrieden geben. Bis es jedoch soweit ist, werden andere Ansätze benötigt. Solange wird Constraintprogrammierung noch eine sehr wichtige Rolle spielen.

4 Systemübersicht und Auswahl

Um möglichst eine fundierte Aussage über die Lösbarkeit des Problems mit einem gewählten System zu machen, werden im Folgenden mehrere Constraintlöser untersucht. Die Zahl beschränkt sich auf fünf, da die meisten Constraint-Solver als Erweiterung zu einem Prolog-System entwickelt wurden. Diese können hier aber nicht eingesetzt werden, da die Anbindung an Delphi problematisch ist. Das gesuchte System muss bestimmte Hilfestellungen für das Fahrzeugrouten bieten, sodass das Problem in kurzer Zeit implementierbar ist. Die Lösung muss in irgendeiner Form eine Anbindung an Delphiprogramme erfahren. Es werden somit folgende Eigenschaften benötigt:

1. Mit Hilfe der Constraintbibliothek muss eine Windows-NT-Dynamic-Link-Library (DLL) erstellt werden können. Die Aufrufkonventionen müssen mit denen von Delphi vereinbar sein.
2. Die zu erstellende DLL muss kostengünstig zu warten sein. Die Programmiersprache sollte deshalb entsprechend populär sein (C, C++ oder Object Pascal).
3. Die Problemrepräsentation sollte im Bereich des Fahrzeugroutens unterstützt werden. Dies könnte bereits eine Beispielimplementierung eines kleinen Routingproblems sein. Im günstigsten Fall ist es eine Bibliothek von Klassen, die speziell für das Fahrzeugrouten entworfen wurde.
4. Es muss möglich sein, eine robuste Version zu erstellen.

4.1 ECL^{i}PS^e

Das System wurde am Imperial College London entwickelt und befindet sich inzwischen in den Händen von Parc Technologies Ltd. ECL^{i}PS^e (ECRC Common Logic Programming System) stellt sich als „Constraint Logic Programming System“ [Parc] vor. In der Tat konzentriert sich das System auf die Kompatibilität zu Prolog und entsprechenden Constraints. Die Entwicklungsumgebung existiert für UNIX-Derivate und Windows. Die Sprache ist eine Erweiterung von Prolog und beherrscht den Umgang mit Arrays und Strukturen. Darüber hinaus besteht die Möglichkeit ECL^{i}PS^e-

Programme mit C/C++-Code zu erweitern oder ECL^{i}PS^e-Bibliotheken von C/C++-Programmen aus zu benutzen. Da die Daten von ECL^{i}PS^e nach C konvertiert werden, lässt sich ECL^{i}PS^e als Bibliothek verwenden. Eine bidirektoriale ereignisgesteuerte Kommunikation ist über die Schnittstelle zu C möglich. Leider verfügt ECL^{i}PS^e nicht über eine Unterstützung für das Fahrzeugrouting.

4.2 ILOG Bibliotheken

ILOG ist seit vierzehn Jahren in der Optimierung tätig. Dabei liefert das Unternehmen Bibliotheken für mathematische Constraints und KI-Constraints. ILOG selbst gibt an, dass es zahlreiche Unternehmen mit seinen Bibliotheken unterstützt hat. Darunter finden sich so klangvolle Namen, wie Oracle Corporation, Deutsche Telekom AG, Nokia Corporation, SAP AG oder PSA Peugeot-Citroën.

Durch die Einführung der Trennung zwischen Modell und Algorithmus, seit der Version 5 der Solver-Bibliothek, bildet die Concert-Bibliothek eine Verbindung zwischen der CPLEX- und Solver-Bibliothek. CPLEX bildet ein breites Spektrum der Linearen Programmierung ab und zielt eher in den Bereich der Mathematischen Programmierung. Ein Modell, welches mit der Concert-Bibliothek erstellt wurde, kann sowohl mit dem Solver, als auch mit der CPLEX-Bibliothek gelöst werden. Dieser hybride Ansatz mag für einige Problemstellungen sehr wertvoll sein, den Fragen des Fahrzeugroutens kommt es wenig entgegen. Routingbelange finden in der Dispatcher-Bibliothek ihre Unterstützung, die auf die Solver-Bibliothek aufbaut.

Bei Verwendung der ILOG-Bibliotheken wären somit nur die Concert-, Solver- und die Dispatcher-Bibliothek interessant. Mit Hilfe der Concert-Bibliothek wird ein sogenanntes Modell erstellt. Dieses Modell wird in diesem Fall von dem Solver gelöst. Die Dispatcher-Bibliothek baut auf die Solver-Bibliothek auf und erweitert dessen Klassen um routingtypische Elemente, wie Fahrzeuge oder Knoten. Da die ILOG-Bibliotheken in C++ implementiert wurden und eine C++ Schnittstelle besetzen, bieten sie die besten Voraussetzungen, eine entsprechende DLL zu erstellen.

4.3 CHIP

CHIP (Constraint Handling in Prolog) ist ursprünglich als Erweiterung zu Prolog in München am European Computer Industry Research Centre (kurz: ECRC) entstanden. Die Weiterentwicklung wurde von Cosytec übernommen und hat inzwischen die Version 5.4 erreicht.

Chip ist eine ausgereifte Constraint-Bibliothek, die über umfassende Möglichkeiten

verfügt, Constraints zu deklarieren. Neben der Behandlung von Constraints mit endlichen Domains können auch endlose Variablendomains verwendet werden. Deren Einsatz in mehreren System wie EVA erprobt worden. Mit dem System EVA wird der Transport von Atommüll zur Wiederaufbereitungsanlage nach La Hague geplant. Dabei muss die Verfügbarkeit von passenden Transportfahrzeugen beachtet werden. Möglicherweise kommt dieses System dem der zu erstellenden DLL sehr nahe. Es ist also möglich, mit CHIP eine robuste Lösung zu erstellen. Doch leider konnte keine konkrete Unterstützung für Vehicle-Routing-Probleme gefunden werden. Das einzige Constraint, dass Fahrzeugrouten irgendwie unterstützt, heißt „cycle“, mit dem sich Rundrreisen berechnen lassen. Die Repräsentation des Vehicle-Routing-Problems scheint zwar möglich aber dennoch schwierig und zeitaufwendig.

Neben einer C/C++-Schnittstelle existiert ein Prologinterface. Außerdem bestehen Versionen für OpenVMS, UNIX und Windows NT. Interessant ist auch, dass das System eine Unterstützung für Datenbanksysteme bietet.

4.4 Mozart/OZ

OZ ist eine „multi-Paradigmen“ Sprache, die Eigenschaften der Objektorientiertheit, der funktionalen und der logischen Programmierung beherbergt. Die Entwickler selbst rücken OZ für parallele Abläufe, soft-real-time Prozesse und reaktive Anwendungen ins Rampenlicht. Weiterhin liegt ein Schwerpunkt der Programmiersprache in der Unterstützung zur Constraintprogrammierung. In OZ bleibt der Forschungscharakter leider nicht verborgen. So finden sich in der Dokumentation Erklärungen zur internen Behandlung von Variablen.

Die aktuelle Version 3 kann in Verbindung mit einer Entwicklungsumgebung namens Mozart aus dem Internet [OZ] kostenfrei bezogen werden. Mozart läuft auf verschiedenen UNIX-Derivaten und auf Windows. OZ unterstützt das verteilte Lösen von Constraintproblemen. Die Constraints müssen über eine endliche Domain verfügen. Sie werden in einem Speicherraum, „Store“ genannt, abgelegt, wobei die effiziente Verwaltung der Speicherzustände von Variablen durch diese Stores erreicht wird. Ein Store ist dabei mehr, als nur eine Sammlung von Speicherzuständen. Verbunden mit einem „Propagator“, wird die Variablendomain im Store verändert. Ein Propagator stellt zumindest Kantenkonsistenz her, wobei auch hier zwischen Domain-Propagation und Intervall-Propagation unterschieden wird. Was im

Kontext von OZ als Domain-Propagation bezeichnet wird, wurde bereits im Kapitel 2.3.2 beschrieben. Das Intervall-Propagieren beschränkt sich nur auf die Grenzen einer Domäne, es werden also keine Werte zwischen den Domaingrenzen entfernt. Dafür ergibt sich aber eine geringere Prozessorbelastung. Durch einen „Distributor“ werden jeweils eine Strategie zum Auswählen einer Variable und zur Auswahl eines Wertes festlegt. Gewisse grundlegende Distributoren, wie „First-Fail“, sind bereits implementiert und lassen sich erweitern. Ein Explorer erlaubt die Visualisierung von Suchfortschritten und hat damit einen gewissen Wert für das Debugging und Profiling.

Interessant ist auch, dass overconstrained Probleme von OZ berücksichtigt werden können. Somit lassen sich durch die Anwendung von „Reified Constraints“ Verbindungen zu logischen Constraints herstellen und primäre und sekundäre Constraints implementieren. Wie im Kapitel 2.6.2 beschrieben, werden die Constraints je nach Priorität beachtet oder gestrichen. Was allerdings negativ durchschlägt, ist die maximale Integer-Größe von 134.217.726 die von dem Constraintmodul FD unterstützt wird. Außerdem wirkt die Sprache durch die vielen verschiedenen Paradigmen recht unhandlich. Ein leichter Einstieg lässt sich nur sehr schwer finden. Dazu kommt, dass das C/C++-Interface nur in eine Richtung funktioniert. So kann OZ zur Zeit nicht als C++-Bibliothek verwendet werden. In [Chew u.a. 2000] wird, was eine C++-Bibliothek für Oz betrifft, auf die Zukunft verwiesen.

4.5 StarFlip++

StarFlip wurde unter dem Gesichtspunkt der Wiederverwendbarkeit für ein weites Feld der Optimierung an der Wiener Universität entworfen. In der Tat sind die Spezialitäten nur knapp gehalten und beschränken sich auf grundlegende Klassen. Allerdings kann StarFlip++ mit Fuzzy-Sets und Fuzzy-Constraints umgehen.

Vages Wissen und Beschränkungen finden in den Fuzzy-Constraints ihre Fortsetzung und zur Unterstützung der Optimierung wurden Tabu-Search und genetische Algorithmen implementiert. Die StarFlip++ Bibliotheken wurden in C++ geschrieben und existieren als Quellcode für UNIX-Derivate und Windows-Compiler.

Klassen, die höher in der Abstraktionsebene liegen, wie z.B. Fahrzeug- oder Ladungsklassen sucht man bei StarFlip++ leider vergebens.

4.6 Fazit

Alle fünf hier vorgestellten Bibliotheken, haben ihre Vorteile. Besonders im reinen

Umfeld der Constraint-Programmierung können sie ihre Stärken ausspielen. Für OZ spricht die Möglichkeit, parallele Lösungen zu entwerfen. Allerdings muss für die Lösung eine eigene Programmiersprache verwendet werden, was hinsichtlich der Einarbeitungszeit und der Softwarewartung sehr aufwendig zu sein scheint. Die Unterstützung für VRP's fehlt, wodurch OZ nicht verwendet werden kann.

StarFlip++ unterstützt vor allem Fuzzy-Constraints. Positiv ist, dass hier die C++-Quellen vorliegen und somit eine C++-Schnittstelle existiert. Negativ ist wiederum die fehlende Unterstützung von VRP's.

Die CHIP-Bibliotheken können ebenfalls aus C++-Programmen aufgerufen werden. Die lange Erfahrung der Entwickler lässt auf eine stabile Version schließen, die im industriellen Bereich eingesetzt werden kann. Auch kann die Datenbankunterstützung sehr wertvoll sein. Allerdings fehlt auch hier eine Unterstützung für Routingprobleme.

ECL^{i}PS^{e} bietet keine Datenbankunterstützung und konzentriert sich nur auf Constraints. Leider fehlt auch hier eine Unterstützung für Vehicle Routing Probleme.

Einzig und allein ILOG unterstützt dieses spezielle Feld der Optimierung mit einer eigenen Bibliothek. Die C++-Bibliotheken lassen sich in Form einer DLL oder eines COM-Objektes von Delphi aus nutzen.

	ECL^{i}PS^{e}	ILOG	**CHIP**	**OZ**	**StarFlip++**
DLL-Erstellung	+	++	++	--	++
Programmiersprache	C (C++)	C/C++	C/C++/Prolog	OZ	C++
Problemrepräsentation	0	++	0	--	--
Robustheit	+	+	+	+	0

Tabelle 2 Systemvergleich

Die Werte für die Robustheit beruhen auf persönlichen Einschätzungen und gehen nur geringfügig in die Wertung ein. Sie spiegeln eher die Erfahrung wieder, die mit diesen Systemen gesammelt wurde. Die Repräsentation des „Pickup and Delivery“ Problems ist maßgeblich für die Auswahl der Bibliothek verantwortlich. Die meisten Objekte der Problemstellung sind in den ILOG-Bibliotheken bereits vorhanden, so dass die Entwicklung mit der Solver- und Dispatcher-Bibliothek eine schnelle Implementierung verspricht.

5 Klassifizierung des „Pickup and Delivery“ Problems

Wie bereits im Kapitel 1 beschrieben, wird eine Fahrzeugflotte mit dem Transport von Gütern betraut. Jedes Fahrzeug hat seine Eigenschaften und kann daher nicht jeden Auftrag erfüllen. Die zu schaffende Bibliothek hat die Aufgabe, auf diese Problematik einzugehen und eine gute Lösung zu erzeugen.
Die Güter müssen an einer Beladestation auf ein Fahrzeug aufgeladen und an einer Entladestation abgeladen werden. Es versteht sich von selbst, dass das Beladen vor dem Abladen stattfindet und dass ein leeres Fahrzeug nicht entladen werden kann.
Ein Auftrag gibt neben dem Entladeknoten eine Reihe von möglichen Beladeknoten vor. Von welchem Beladeknoten die Ladung abgeholt wird, muss das System entscheiden. Aber jede Station, ob nun Depot, Belade- oder Entladestation hat seine eigenen Öffnungszeiten. Diese müssen eingehalten werden und erklären die nahe Verwandtschaft des Problems zum ATSP-TW.

Leider erklärt dies nur, dass es fast unmöglich ist, eine Lösung innerhalb einer kurzen Zeit zu finden. Aber eigentlich sollte es nach n-Schritten (bei n Städten) möglich sein. Ein Algorithmus müsste nur immer genau „wissen“, welche Stadt für die optimale Lösung die nächste in der Kette sein soll. Einen solchen „Hellseher-Algorithmus“ vermag man auf einen normalen PC, nur schwerlich zu konstruieren. Doch könnte wenigstens als theoretisches Modell eine nicht deterministische Turingmaschine verwendet werden. Ein darauf erzeugter Algorithmus wäre ebenfalls nicht deterministisch und es kann daher behauptet werden: Ein Problem, das mit einem nicht deterministischen Algorithmus in polynomieller Zeit gelöst werden kann, liegt in der Klasse NP. Demgegenüber steht die Klasse P, die alle Probleme enthält, die deterministisch in polynomieller Zeit gelöst werden können. Gewiss liegen manche Probleme gleichzeitig in NP und in P.

Der in Kapitel 1 (Seite 2) vorgestellte Algorithmus „Vollständiges Suchen“ hat eine Komplexität von der Größenordnung O(n!). Und somit gehört ein TSP *mit diesem Algorithmus* nicht der Klasse P an. Da momentan kein Algorithmus bekannt ist, der ein TSP in polynomieller Zeit löst, vermutet man, dass dieses Problem nicht in P liegt sondern nur in NP. Ein Beweis wäre möglich, indem *alle* möglichen Algorithmen untersucht werden, die das Problem lösen. Ist von ihnen keiner in der Lage, das Problem in polynomieller Zeit zu lösen, muss das Problem in NP liegen, ohne in P zu sein. Der Haken bei der Sache ist nur, das nicht alle möglichen Algorithmen bekannt

sein können.

Um die Probleme aus NP zu vergleichen, wendet man einen Transformationsalgorithmus T(x) an. Man möchte wissen, ob das Problem A schwerer ist, als das Problem B. Dazu wandelt T die Eingabe x für B in eine Eingabe für A um. Wenn das möglich ist, ist B nicht schwerer als A oder B <= A. [c't 2001]

Man kann also entscheiden, ober ein Problem mindestens so schwierig ist wie ein anderes. Doch welches ist das schwierigste Problem oder welches sind die schwierigsten Probleme?

Eine Sammlung, die darüber Aufschluss gibt, wurde beispielsweise von Garey und Johnson zusammengestellt [Garey & Johnson 1979]. Sie enthält die schwierigsten Probleme die in NP liegen und als „NP vollständig“ gelten.

Wie in [Ascheuer u.a. 1999] gezeigt wurde, zählt das „Asymmetric Travelling Salesman Problem with Time Windows“ zu den NP-vollständigen Problemen und man kann zeigen, dass das vorliegende Problem ebenfalls NP vollständig ist.

Dazu wird jede Ladung auf eine Größe von 0 gesetzt. Die Auswahl der Beladepunkte wird auf einen verkürzt. Die Reihenfolge Beladeknoten -> Entladeknoten kann aufgrund der verschwindend geringen Transportlast aufgehoben werden. Die Flotte besteht nur noch aus einem Fahrzeug, sodass lediglich ein ATSP-TW vorliegt. Werden n Fahrzeuge verwendet, vergrößert sich die Problematik, da vielleicht n-mal der Algorithmus des ATSP-TW verwendet werden muss. Wie oft der Algorithmus aufgerufen wird, hängt von der Auslastung der Fahrzeuge ab. Wird nur eines benötigt, genügt auch ein Durchlauf.

Somit muss angenommen werden, dass das vorliegende Problem mindestens so schwierig ist, wie ein „Asymmetric Travelling Salesman Problem with Time Windows“ und man kann deshalb davon ausgehen, dass es ebenfalls NP-vollständig ist. Daraus ergibt sich die hohe Wahrscheinlichkeit, dass kein Algorithmus existiert, der das Problem in polynomieller Zeit lösen kann. Aber vielleicht lässt sich in angemessener Zeit eine gute Lösung finden.

6 Analyse der Aufgabenstellung

Für das gegebene „Pickup and Delivery“ Problem wird ein gerichteter Graph

$$D:=(V \cup \{nw\}, E)$$

benötigt. Dabei stellt *V* die Menge aller Knoten und das Element *nw* einen Knoten „nowhere“ dar, der keine Koordinaten besitzt. Die Menge der Kanten E enthält die

Verbindungen zwischen zwei Knoten v_i, $v_k \in V$. Somit ist der Graph im üblichen Sinne definiert. Es wird aber nötig sein, von einem vollständigen Graphen auszugehen. Denn jeder Knoten soll von jedem anderen erreichbar sein. Die Menge *DP* bezeichnet alle Depots und ist eine Teilmenge aller verfügbaren Knoten im System.

$$DP \subseteq V$$

Für das System soll sich ein Fahrzeugstandort immer an einem Depot befinden. Man könnte sich V auch als Menge aller Städte Deutschlands vorstellen. Die Menge der Depots enthielte dann nur die Städte, in denen real ein Fahrzeugstützpunkt (Depot) wäre. Da ein Fahrzeug immer von seinem Depot startet, und dort wieder zurückkehrt, kann von einem Heimatstandort des Fahrzeuges gesprochen werden. Dieser lässt sich als Start und Ziel definieren. Der Start- und Endpunkt einer Route hängt somit vom Fahrzeug ab.

Jede Kante ist mit einer Bewertung oder Wichtung

$$c_{ij} > 0 \mid Kosten\ zwischen\ den\ Knoten\ v_i\ und\ v_j \in V$$

versehen, die in der Lösungssuche mit berücksichtigt wird. Sie kann hier als Wegstrecke zwischen den einzelnen Knoten oder als Zeitverbrauch angesehen werden. Weil die Fahrzeuge unterschiedliche Eigenschaften haben, und somit kein konstanter Faktor zwischen dem gefahrenen Weg und der Zeit gefunden werden kann, müssen zwei Kantenbewertungen definiert werden:

1. für die Entfernung: $length_{ij} \mid für\ die\ Knoten\ v_{i,}\, v_j \in V$ und

2. für den Zeitverbrauch: $time_{ij} \mid v_{i,}\, v_j \in V$

Offensichtlich hängt der mathematische Zusammenhang zwischen den Wichtungen *time* und *length* von den Fahrzeugeigenschaften ab.

Die Öffnungszeiten eines Depots können als Menge von Zeitfenstern betrachtet werden, die wiederum Mengen von Uhrzeiten sind. Somit gilt für ein Zeitfenster TW:

$$\mathrm{TW} = \{ z_{1,}\, z_{2,} ..., z_n \mid z_i \text{ ist Uhrzeit} \}$$

Die Öffnungszeiten können als Vereinigung aller Zeitfenster aufgefasst werden:

$$OH = TW_1 \cup TW_2 \cup ... \cup TW_n$$

Zur einfacheren Handhabung werden die Zeiten als Sekunden aufgefasst, die von einem definierten Zeitpunkt an gezählt werden. So könnten für jeden Tag die Sekunden neu gezählt werden. 6:00 Uhr würde dann 21600 Sekunden entsprechen.

Da es aber möglich sein muss, bestimmte Tage zu verplanen, ist es besser, den *universal time code* zu implementieren. Dieser stellt die Anzahl der Sekunden seit der ersten Minute vom 1. Januar 1970 dar.

Die Öffnungszeiten für eine Baustelle (Entladeknoten) und ein Werk (Beladeknoten) wird in der gleichen Weise definiert. Die Menge der Entladeknoten sei

$$DNO \subseteq V$$

und die Menge der Beladeknoten sei

$$PNO \subseteq V$$

Jedem Entladeknoten können mehrere Beladeknoten durch einen Auftrag zugeordnet sein. In der Lösung darf es aber je Entladung nur eine Beladung geben, wodurch der Algorithmus aus mehreren Möglichkeiten einen Knoten wählen muss. Ein Entladeknoten (dno $\in$ DNO) mit seinem Beladeknoten (pno $\in$ PNO) muss augenscheinlich vom identischen Fahrzeug angefahren werden. Sie liegen auf der selben *Tour T*. Hier wird eine Tour als Permutation von Be- und Entladeknoten angesehen.

Wenn *R* der *Rang* eines Knotens in einer Tour T ist, so gilt:

$$R(pno) < R(dno)$$

Hierin wird noch nicht festgelegt, dass ein Entladeknoten direkter Nachfolger eines Beladeknoten ist. Vielmehr soll $R(pno)+1=R(dn)$ nur dann gelten, wenn kein Transportgut gefunden wird, mit dem das Fahrzeug zusätzlich beladen werden kann. Sollte ein Auftrag das gleiche Produkt enthalten, kann es bei Beachtung der Ladekapazität hinzugeladen werden. Andernfalls muss das Fahrzeug den aktuellen Auftrag vollenden, ehe es einen neuen ausführen kann.

Für jeden Be- und Entladeknoten gilt eine Verzögerungszeit $td \geq 0$. Diese Verzögerungszeit ist durch den Ort selbst bedingt und kann nicht durch die Art der Fahrzeuge beeinflusst werden. Allerdings ist für jede Be- und Entladestelle ein Geräuschpegel $G>0$ in db(A) zu beachten, welcher bestimmte Fahrzeuge ausschließen kann. Die maximale Masse eines Fahrzeugs muss ebenfalls beachtet werden, und wird mit $m>0$ angegeben. So kann beispielsweise einem Vierzig-Tonner verboten werden, über eine für maximal zwei-Tonner zulässige Brücke zu fahren.

Um die Kennzahlen der „Objekte“ unterscheiden zu können, werden die Kennbuchstaben der Knoten groß geschrieben und die der Fahrzeuge klein. Somit wird die

maximale Masse eines Knotens durch *M>0* festgelegt. *L*, *B* und *H* legen die maximale Ausdehnung eines Fahrzeuges für einen Knoten fest.

F bildet die Menge aller Fahrzeuge, die zur Verfügung stehen. Jedes Fahrzeug *vh∈F* zeichnet sich durch eine Reihe von Eigenschaften aus. Die Masse *m 0*, die Breite *b>0*, die Länge *l>0* und die Höhe *h>0*, Ladevolumen *V≥0*, zuladbare Masse *vlm≥0* oder die allgemeine Verzögerungszeit des Fahrzeugs *vdt≥0.* Sie muss zu jeder Be- und Entladezeit hinzugerechnet werden. Die Startverzögerung v*ds≥0*, Endeverzögerung v*de≥0*, die Beladeverzögerung v*dl≥0* und die Entladeverzögerung v*dul≥0* bilden die zeitlichen Strafen des Fahrzeugs, die immer einzurechnen sind. Die Reinigungszeit v*dc≥0* muss nur berücksichtigt werden, wenn es im Auftrag entsprechend gekennzeichnet ist. Um eine möglichst freie Wahl der Parameter zu erhalten werden Be- und Entladezeiten je Wareneinheit eingeführt: Beladezeit je m³ *vl≥0*, Entladezeit je m³ *vul≥0*, Beladezeit je Tonne *vg≥0* und Entladezeit je Tonne *vug≥0.* Und der Schallpegel des Fahrzeugs *g>0* in db(A).

Die Eigenschaften m, b, l und h sind physikalische Attribute, die positiv sein müssen. Aus Gründen der Kompatibilität sollen sie ganzzahlig sein und Gramm- und Millimeterwerte darstellen. Die Ladekapazitäten V und vlm werden in den Grundeinheiten Liter und Gramm behandelt.

Jedes Fahrzeug darf nur eine bestimmte Dauer des Tages verwendet werden, sodass die tägliche Arbeitszeit wiedergespiegelt werden kann. Dies wird durch das Einsatzzeitintervall *vt:=[Anfang, Ende]* und die Einsatzdauer *vd≥0* ausgedrückt, die vollständig innerhalb der Einsatzzeit liegen muss. Es kann also eine grundlegende Einsatzbereitschaft über einen gewissen Zeitraum vorliegen, jedoch darf der Fahrer nur eine maximale Zeit arbeiten.

Örtlich ist jedes Fahrzeug an ein Depot gebunden, von dem es morgens abfährt und am Abend zurückkehrt. Dieser Heimatstandort wird durch *vo∈DP* beschrieben.

Für das einmalige Einsetzen des Fahrzeugs je Tag wird *vc≥0* berücksichtigt. Die Fahrzeugkosten je Km im unbeladenen Zustand legt *vcul≥0* fest, im beladenen *vcl≥0*.

Offensichtlich muss *vcl≥vucl* gelten, denn ein Fahrzeug ist im beladenen Zustand schwerer und verbraucht in der Regel mehr Sprit. Der Einfachheit halber wird hier eine lineare Beziehung angenommen, sodass man die Werte der Spritkosten als

Funktionswerte auffassen kann. Denn wenn das Fahrzeug voll beladen ist, gilt $f_S(x)$ = vcl und wenn es völlig leer ist gilt $f_S(x)$ = vucl. Die lineare Beziehung erlaubt folgende Herleitung:

$$\frac{x-x_1}{x_2-x_1}=\frac{y-y_1}{y_2-y_1} \qquad |\,(y_2-y_1)$$
$$\frac{(y_2-y_1)\cdot(x-x_1)}{x_2-x_1}=y-y_1 \quad |+y_1$$
$$\frac{xy_2-xy_1+x_1y_1-x_1y_2}{x_2-x_1}+y_1=y$$
$$x\frac{(y_2-y_1)}{x_2-x_1}+\frac{x_1y_1-x_1y_2}{x_2-x_1}+y_1=y$$

Die y-Werte bezeichnen die Kosten des Fahrzeugs, abhängig von der geladenen Masse. Die x-Werte stehen für die Kapazität. x_1 hat dabei den Wert für die minimale Kapazität des gegebenen Fahrzeugs und x_2 bezeichnet die maximale Kapazität. Somit nimmt x_1 immer den Wert 0 an, wodurch sich die Formel weiter vereinfachen lässt:

$$x\frac{(y_2-y_1)}{x_2-0}+\frac{0\cdot y_1-0\cdot y_2}{x_2-0}+y_1=y$$
$$x\frac{(y_2-y_1)}{x_2}+y_1=y$$

Für die Kosten je Zeit kann die gleiche Funktion mit den gegebenen Parametern (unbeladen für x_1) $vcult \geq 0$ sowie (beladen für x_2) $vclt \geq 0$ verwendet werden.

Die Fahrzeuge werden als Klasse und Typ (VHC und VHT – beides Mengen von Fahrzeugen) betrachtet. Jede Menge beschreibt die Fahrzeugflotte unterschiedlich und bildet einen anderen Blickwinkel ab.

So könnte man beispielsweise allen Mountainbikes die TypeID 1 zuordnen und allen Rennräder die TypeID 2. Die ClassID 1 erhalten alle Fahrräder mit Hängerkupplung und die ClassID 2 wird an alle Räder mit Kindersitz vergeben. So beschreibt in diesem Fall die Klasse die Transportmöglichkeit und der Typ sagt etwas über die Schnelligkeit aus.

Ein Auftrag besteht im Wesentlichen aus einem Entladeknoten (dno $\in$ DNO) und einer Liste von Beladeknoten (pno $\in$ PNO). Außerdem enthält er ein Liste von bevorzugten Fahrzeugen, deren TypeID und ClassID. Und natürlich wird die Menge des Transportgutes und dessen ID mitgegeben.

Das Transportgut wird durch eine Zahl beschrieben. Die Liefermenge wird in Litern und in Gramm berücksichtigt $ol \geq 0$ und $og \geq 0$.

Das Fälligkeitsdatum (*od*) und das Zeitfenster der Lieferung ($ot \in TW$) können durch ein einziges Zeitfenster angegeben werden, wenn der utc als Grundlage benutzt wird. Die Vorgabe einer Fahrzeugklasse ($vhc \in VHC$), eine Menge vorgegebener Fahrzeugtypen ($VHT_O \subseteq VHT$), eine mögliche Vorgabe eines Fahrzeuges ($vh \in F \cup \emptyset$), Reinigungsflag ($cf \in \{0; 1\}$) sowie Vorladungsflag ($plf \in \{0; 1\}$)[1] machen zusätzlich einen Auftrag aus. Das Reinigungsflag entscheidet darüber, ob ein Fahrzeug nach einem Auftrag gesäubert werden muss. Aus Gründen der Einfachheit wird hier angenommen, dass sich das Fahrzeug direkt am Entladeknoten säubern lässt.

Das Bedürfnis, Fahrzeuge für mehrere Stunden zu sperren, wird durch Pausen ausgedrückt. Dadurch können Freistunden oder Reparaturzeiten beschrieben werden. Eine Pause hat dabei mehrere Attribute – Datum *pda*, Zeitfenster der Pause $ptw \in TW$, Dauer $pdr > 0$, maximale Verletzung der Pause $pi \geq 0$ in Sekunden sowie die beladene Menge in Litern und Gramm, die zulässig sind: $pll \geq 0$, $plg \geq 0$.

Die Angabe inwieweit das Fahrzeug beladen sein soll, mag etwas merkwürdig anmuten. Jedoch ist es dadurch möglich, Reparaturen und gewöhnliche Pausen durch die gleiche Klasse abzubilden. Wenn das Fahrzeug in eine Werkstatt fahren soll, wird verlangt, dass es unbeladen ist. Legt der Fahrer seine Mittagspause ein, spielt dies aber keine Rolle. Auch ist der Ort unwichtig, an dem er seinen Imbiss zu sich nimmt. Dagegen ist die richtige Werkstatt sehr wohl von Bedeutung. Somit gilt für den Pausenstandort $pn \in V \cup \{nw\}$. Ist pn=nw kann die Pause irgendwo stattfinden. Natürlich muss das Fahrzeug $pvh \in F$ für die Pause festgelegt sein, da eine Pause nur für einen bestimmten Fahrer samt seines Fahrzeugs gelten kann.

Bevor die objektorientierte Analyse- und Designphase beginnt, soll die Funktionsweise und die wichtigsten Merkmale der ILOG-Bibliotheken im folgenden Abschnitt erläutert werden.

7 Funktionsweise der ILOG-Bibliotheken

Während der Suche nach einer Lösung wird grundsätzlich Backtracking (Kapitel 2.4.1) praktiziert. Es werden also Zustände angenommen und auf Gültigkeit über-

1 Sollte der Wert 1 sein, so wurde ein Fahrzeug mit diesem Material beladen und kann ohne Beladeverzögerung das Werk verlassen. Dazu ist nötig, dass das Fahrzeug angegeben wurde.

prüft. Bei Erfolg wird versucht, einen neuen erweiterten Zustand anzunehmen, bei Misserfolg wird ein früherer Zustand restauriert und in anderer Weise als zuvor ausgebaut. Um dieses Backtracking zu realisieren muss im Allgemeinen eine umfangreiche Speicherverwaltung vorhanden sein, die das Beschriebene leistet und Backtrack-Zustände verwalten kann. Der Bestandteil, der dies innerhalb der ILOG-Bibliotheken leistet, heißt „IloEnv“.

Mit Hilfe eines IloEnv-Objektes kann ein „IloModel“ erstellt werden, welches seit der Version 5 der Solver-Bibliothek möglich ist. Die starke Trennung zwischen Modell und Algorithmus hat durchaus Vorteile. So kann ein Modell für eine bestimmte Gruppe von Fahrzeugen generiert werden oder alle Constraints eines Fahrzeugs enthalten.

Doch muss die Trennung zwischen Modell und Algorithmus spätestens an der Stelle durchbrochen werden, an der eine Lösung verlangt wird und das Modell berechnet werden soll. Hier kommt der IloSolver ins Spiel. Er kann die Algorithmen-Objekte aus einem Modell extrahieren und sie in einem Goal (dazu weiter unten mehr) verwenden. Die Methode der Extraktion, ist in der Klasse IloExtractable implementiert, weshalb die meisten Modellklassen von IloExtractable erben.

Wie in Kapitel 4.2 bereits erläutert, werden die Klassen, die zum Fahrzeugrouten nötig sind, durch die Dispatcher-Bibliothek bereitgestellt. Eine wichtige Rolle spielt hierbei die Klasse IloNode, welche durch ihre Objekte einen Punkt auf einer Landkarte markiert und gewissermaßen die Knoten des in Kapitel 6 beschriebenen Graphen darstellt. Die Kanten werden durch sogenannte IloDimension2-Klassen realisiert. Da zwischen zwei Knoten verschiedene Kosten und damit Wichtungen auftreten können, sind sie eher allgemein gehalten. Um beispielsweise den Zeitverbrauch zu definieren, würde man schreiben:

```
IloEnv env;
IloModel model(env);
IloDimension2 time(env, IloEuclidean, „Time“);
model.add(time);
```

Mit diesen Zeilen hat man den Zeitverbrauch zwischen allen Knoten definiert, der aufgrund ihrer jeweiligen Entfernung zustande kommt.

Nebenbei wird in diesem Beispiel euklidischer Abstand angenommen. Natürlich ist dies bei dem vorliegenden Problem nicht möglich. Schon allein deshalb nicht, weil dort Längen- und Breitengrade zum Einsatz kommen. Um den Abstand zwischen

zwei geographischen Punkten zu berechnen, bietet die Dispatcher-Bibliothek die Funktion IloGeographical an. Diese berechnet eine Näherung basierend auf einer Kugel zwischen zwei Knoten. Allerdings kommt dabei nur der Einheitskreis zum Zuge, sodass die Funktion mit dem durchschnittlichen Erdradius multipliziert werden muss. Statt „IloEuclidean“ und „IloGeographical“ können auch eigene Entfernungsfunktionen installiert werden. Diese könnten dann beispielsweise fahrende Baustellen berücksichtigen.

An einer IloVisit wird ein Fahrzeug be- oder entladen. Die Veränderung der freien Ladekapazität des Fahrzeugs und seiner Zuladung wird durch die Klasse IloDimension1 realisiert. So kann das Beladen eines Fahrzeugs folgendermaßen beschrieben werden:

```
IloDimension1 weight(env, „Masse“);
IloNode nPickup(env, x, y);
IloVisit vPickup(nPickup, „Tomaten“);
model.add(vPickup.getTransitVar(weight) == 200);
model.add(vPickup);
```

Diese Zeilen sagen aus, dass irgendein Fahrzeug 200 Kilogramm Tomaten aufladen soll. Da ein reales Fahrzeug nicht unendlich viel Kapazität besitzt, muss entsprechend eine Grenze gesetzt werden:

```
IloNode fNode(env, x1, y1);  //Startknoten
IloNode eNode(env, x2, y2);  //Endknoten
IloVehicle aVehicle(fNode, eNode, „Leichter LKW“);
aVehicle.setCapacity(weight, 300);
model.add(aVehicle);
```

Hier wird ein Fahrzeug erzeugt, welches vom Knoten „fNode“ losfährt und am Knoten „eNode“ ankommt. Bezüglich der Masse besteht eine Grenze von 300 Kilogramm. Gewiss lassen sich hier andere Maßeinheiten annehmen, da sie nirgends festgelegt werden. Es ist also wichtig, dass sich ein Programm nur auf eine einzige Maßeinheit beruft.

Die Modell-Klassen haben meistens das Präfix „Ilo“ wohingegen den Algorithmenklassen ein „Ilc“ vorangestellt wurde. Somit findet die Implementation eines Suchalgorithmus in einem IlcGoal statt. Goals stellen gewissermaßen einen Algorithmus dar. Sie erben von der Klasse IlcGoal beziehungsweise IloGoal und definieren

bestimmte Funktionen, die zur Handhabung von Goals notwendig sind. Die wichtigste Funktion ist wahrscheinlich IlcGoal::execute des Algorithmenobjektes, denn sie implementiert den Algorithmus. Die Verarbeitung von Goals geschieht mittels eines Stapelspeichers. Er gibt durch seine Struktur die Reihenfolge zur Abarbeitung der Goals vor. Das oberste Goal wird immer als nächstes ausgeführt. Die Kontrolle darüber hat die Funktion IloSolver::next. Diese beginnt eine Suche oder führt sie fort. Wird IloSolver::next das erste Mal aufgerufen, werden alle Goals auf den Goalstack gelegt. Danach wird das oberste Goal vom Stack genommen und ausgeführt. Sollte es Subgoals besitzen werden diese auf den Stack gelegt. Danach wird wiederum das oberste Goal vom Goalstack entfernt und ausgeführt. Dies kann das Subgoal eines vorherigen Goals sein oder ein komplett neues. Die Funktion beendet ihre Arbeit, wenn der Stack leer ist.
Um ein einzelnes Goal auszuführen kann auch die Funktion IloSolver::solve verwendet werden. Sie arbeitet ähnlich, nur dass sie immer eine neue Suche beginnt.

Die Punkte, an denen der Algorithmus zurückkehrt, wenn sich ein Zweig als unlösbar herausgestellt hat, werden hier „choice points“ genannt. Ein solcher Choice Point ist eine Disjunktion von Goals. Das spezielle Goal IlcOr erzeugt eine solche Verknüpfung. Alle Goals, die an IlcOr übergeben wurden, sind Subgoals von IlcOr. Während des Abarbeitens eines IlcOr-Goals geschieht Folgendes [SolvManual51]:

1. Der Zustand des Solvers wird gespeichert. Dazu gehört auch der Goal-Stack.
2. Das erste Subgoal wird dem Goalstack hinzugefügt.
3. Alle anderen Subgoals (der Disjunktion) werden als ungetestete Goals für den Choice Point gespeichert.
4. Das erste Goal wird von dem Goalstack genommen und ausgeführt. Sollte es fehlschlagen, wird der Zustand des Solvers wiederhergestellt und das erste ungetestete Subgoal wird auf den Stack gelegt. Sollten alle Subgoals von IlcOr fehlschlagen, war auch IlcOr nicht erfolgreich.

Leider ist es sehr selten möglich, durch eine erste Lösung ein gutes Ergebnis zu erhalten. Doch kann ein schlechtes Ergebnis verbessert werden. Dazu bietet der Dispatcher einige Heuristiken, wie GLS, Tabu Search und Greedy Search an, die zur zweistufigen Lösungsfindung verwendet werden können:

1. Eine erste Lösung erzeugen.

2. Diese Lösung verbessern.

Nach dieser kurzen Sammlung von ILOG-Eindrücken kann die Analysephase beginnen.

8 Objektorientierte Analyse und Design

Im folgenden Abschnitt werden die Klassen identifiziert, die letztendlich implementiert werden sollen.

8.1 OOA

In der Objektorientierten Analyse werden Objekte der realen Welt und deren Zusammenwirken abgebildet. Die Suche nach solchen Objekten und die für die Programmierung wichtigen Klassen kann durch Evaluierung jedes Substantives der Problemstellung durchgeführt werden. In diesem Fall wird die Suche nach Klassenkandidaten durch die Aufgabenstellung in Form der Eingangsdaten beschleunigt.
Man kann bereits an dieser Stelle „Beladeknoten“, „Entladeknoten“, „Depotknoten“, „Fahrzeug“, „Auftrag“ und „Pause“ als Kandidaten identifizieren.

Die Be- und Entladeknoten erhalten alle Attribute, die in dem Kapitel 6 auf der Seite 43 genannt wurden. Doch können sie aufgrund ihrer breiten Übereinstimmung in der Klasse „Besuch“ zusammengefasst werden. Ein Depot kann im Dispatcher durch ein IloNode-Objekt repräsentiert werden, sodass hierfür keine eigene Klasse definiert wird. Allerdings müssen die Öffnungszeiten durch entsprechende Constraints repräsentiert werden. Etwa in dieser Form:

```
IloNode fNode(myEnv, x, y); IloNode eNode(myEnv, x, y); //Depot-Knoten
IloVisit start(fNode); IloVisit end(eNode); //Depot
IloVehicle(myEnv, start, end, „LKW“); //beginnt am depot und endet dort
myModel.add(start.getCumulVar(time) >= minTime); //nach minTime abfahren
myModel.add(end.getCumulVar(time) <= maxTime); //vor maxTime zurück
```

Ein Fahrzeug ist in einem „Vehicle Routing“ Problem ein zentraler Bestandteil und muss durch eine Klasse repräsentiert werden. Die vielfältigen Attribute können leider nicht komplett durch die Klasse IloVehicle aufgefangen werden, sodass sie entweder erweitert oder um eine Klasse ergänzt werden muss. Um die Algorithmen der Dispatcher-Bibliothek weiter nutzen zu können, wird hier eine ergänzende Klasse geschaffen. Eine Vererbung findet nicht statt.

Ein Auftrag repräsentiert ein Fülle von Daten und sollte durch eine Klasse

vertreten sein. Von diesem müssen Informationen wie Ladungsmenge, Transportgut oder mögliche Beladeknoten bereitgestellt werden. Sie können dann durch Constraints abgebildet werden.

Der Sachverhalt einer Unterbrechung oder Pause findet in der Dispatcher-Bibliothek durch IloVehicleBreakCon eine Entsprechung. Durch Objekte dieser Klasse kann die Fahrt eines Fahrzeugs unterbrochen werden. Es kann auch durch ein weiteres Constraint zu einer Pause an einem bestimmten Ort gezwungen werden.

Um nicht mit mehreren Systemen zur Namensvergabe zu arbeiten, sondern nur mit einem, erhalten alle Klassen, die nicht den ILOG-Bibliotheken angehören das Präfix „Ml“. Außerdem werden alle Klassennamen ins Englische überführt.

Die Anzahl der Fahrzeuge übersteigt gewiss die 1 und die Art des Problems lässt darauf schließen, dass immer wieder ein bestimmtes Fahrzeug abgerufen werden muss und nicht iterativ alle hintereinander bearbeitet werden. Alle Fahrzeuge in einem Array zu speichern ist daher zu kurzsichtig, weshalb hier für die Klasse MlVehicle eine Verwaltungsklasse MlVehicles geschaffen wird. Diese kann auch Objekte der Klasse die IloVehicle auf Anfrage erzeugen. Wendet man diese Erkenntnisse auf alle anderen Klassen an, findet sich auch für diese die Notwendigkeit, Verwaltungen zu schaffen, so dass das abgewandelte Klassendiagramm erstellt werden kann:

Für das Interface der Bibliothek sollen möglichst nur wenige Funktionen definiert werden. Nach Möglichkeit soll ein „lade Problem“ und „berechne Problem“ genügen, um den Nutzer von den Interna weitestgehend abzuschirmen. Es stellt sich somit die Frage, wie die Objekte der so gefundenen Klassen erzeugt und verwaltet werden können. Es erscheint an dieser Stelle sinnvoll, eine Managerklasse zu definieren, die über das Leben und den Tod von Ml- und Ilo-Objekten bestimmt. Dazu muss ein Manager ein IloEnv-Objekt enthalten, um Ilo-Objekte erzeugen zu können. Ein Ilo-Model, IloSolver und ein IloDispatcher ist ebenfalls notwendig.

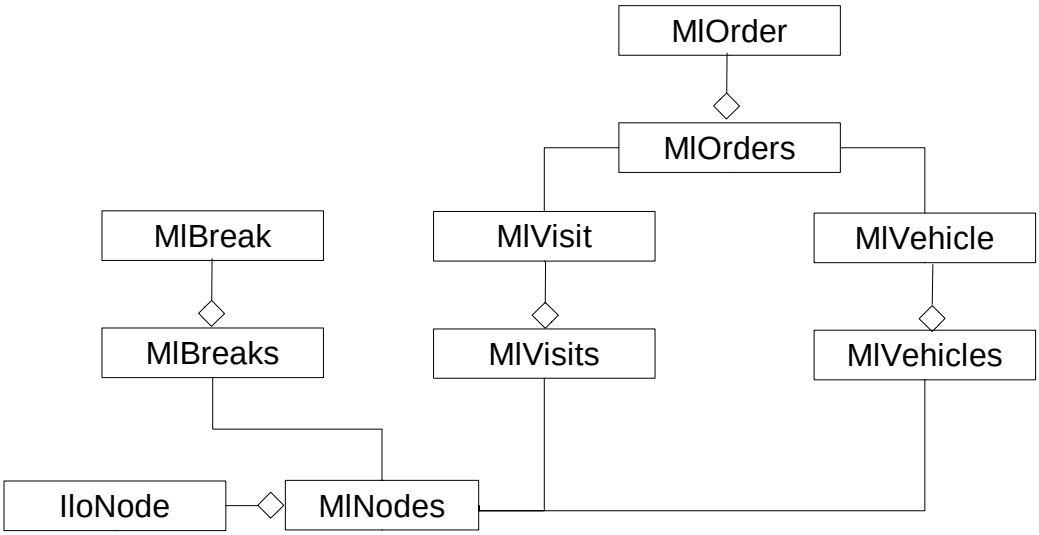


Abbildung 9 Entwurf UML-Klassendiagramm

Wie im Kapitel 6 erläutert wurde, kann ein Zeitfenster als Menge von Sekunden aufgefasst werden. Ein Fahrzeug, dass einen Knoten besuchen will, muss nur seine Ankunftszeit innerhalb einer Menge TW finden. Eine Menge TW enthält leider jede Sekunde des Zeitfensters, so dass eine Programmierung bezüglich des Speicherplatzes und der Performanz schwierig sein würde. Für die hiesigen Zwecke genügt die Speicherung der Zeitfenstergrenzen.

Ein Zeitintervall besteht immer aus einem Anfangszeitpunkt und einem Endzeitpunkt. Die Zeiten sollen in der gesamten Bibliothek sekundengenau betrachtet und ebenso verarbeitet werden. Deshalb wird hier das Zeitformat utc (siehe Seite 43) verwendet. Günstig erscheint eine Klasse, die die Zahlenpaare aufnimmt und eine Klasse, die diese Intervall-Objekte verwaltet. Eine Klasse, die diese Aufgabe übernimmt heißt MlTimeWindow.

Die Detailstufe des Diagramms gibt bereits Auskunft über gewisse Zusammenhänge. Demnach ist ein MlManager nur dann ein Manager, wenn ihm ein IloEnv, ein IloSolver, IloDispatcher und ein IloModel angehören. Ebenso wird die Existenz von MlOrders, MlBreaks, MlVisits, MlNodes und MlVehicles verlangt. Dagegen kann beispielsweise eine Instanz der Verwaltungsklasse MlBreaks auch ohne ein Objekt der Klasse MlBreak sein, welches eine einfache Pause darstellt. Bei manchen Klassen

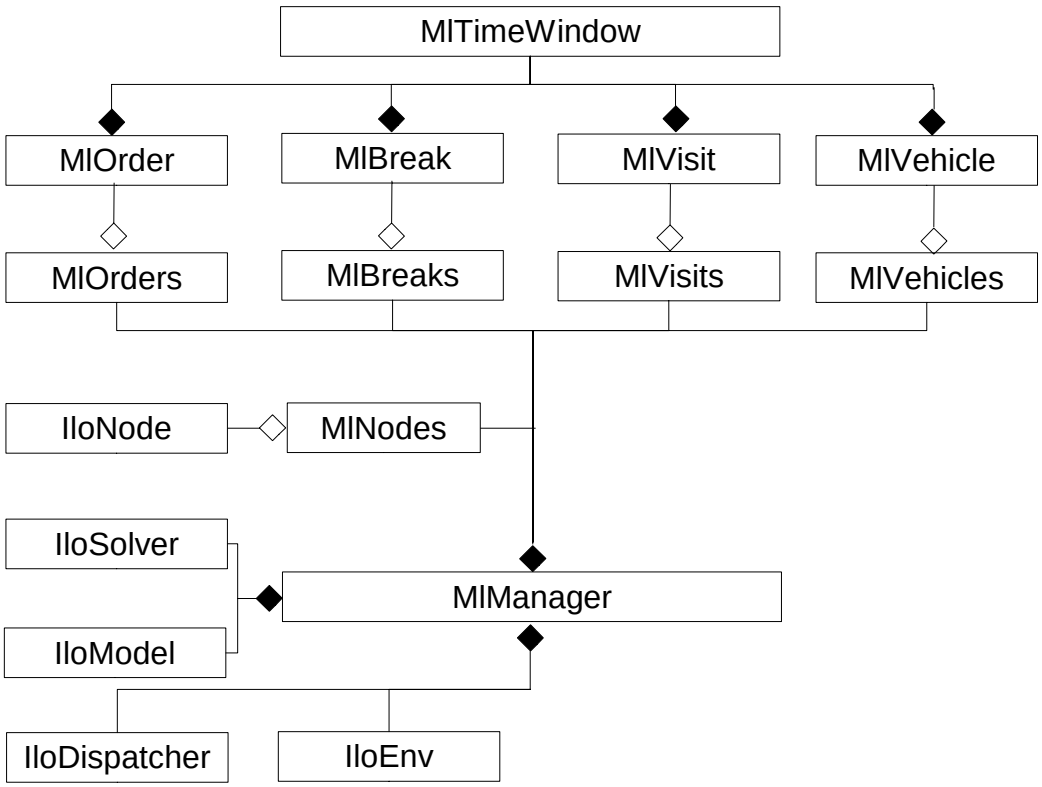


Abbildung 10 erweitertes UML-Klassendiagramm

kann darüber gestritten werden, inwieweit eine Verwaltungsklasse ohne eine entsprechende Klasse Sinn macht. Doch können durch die Verwendung von Pointern bezüglich der Performanz in einer Verwaltungsklasse Vorteile entstehen.

8.2 OOD

In diesem Abschnitt werden die Ergebnisse der Analyse zu einem Design mit seinen Schnittstellen weiterentwickelt. Auf die detailierte Erläuterung aller Header-Files wurde hier verzichtet, da sich die Vorgehensweise wiederholt. Der genaue Inhalt der Schnittstellen befindet sich im Anhang B – Klassendeklarationen.

8.2.1 MlTimeWindow

Günstig erscheint eine Klasse, die die Intervallgrenzen eines Zeitfensters aufnimmt und eine Klasse, die diese Zahlenpaare verwaltet. Hier soll zur Verwaltung der Zeitfensterobjekte die C++-Templateklasse std::list herangezogen werden (Anhang B – Klassendeklarationen - MlTimeWindow).

Die Memberfunktionen MlTimeWindow::add, MlTimeWindow::get und MlTime-

Window::del dienen dem Zugriff auf Zeitfensterobjekte. Da es sich dabei um eine Liste von Objekten handelt, muss auch geprüft werden können, ob sich der Iterator (Zeiger auf das aktuelle Objekt) am Anfang oder Ende befindet. Dazu dienen die Memberfunktionen MlTimeWindow::begin und MlTimeWindow::end. Ebenso notwendig ist das Zurücksetzen des Iterators durch MlTimeWindow::setToBegin und MlTimeWindow::setToEnd.

8.2.2 MlNodes

Die Knoten *V* des Graphen *D:=(V∪{nw}, E)* bilden die Grundlage für die Positionen von Beladestationen, Entladestationen, Pausen oder Depots. Diese Aufgabe wird, wie in der Analyse (Kapitel 8.1) bereits festgestellt wurde, durch die Klasse IloNode erfüllt. Somit enthält ein Knoten die geographischen Koordinaten und eine Identifikationsnummer.

Eine Möglichkeit eine Vielzahl von Knoten zu verwalten bestünde darin, die Klasse IloAnySet der ILOG-Solver-Bibliothek zu nutzen. Die Objekte dieser Klasse können allgemeine Zeiger verwalten, also könnte man durch einen Typecast einen void-Pointer erzeugen und diesen dann einem IloAnySet-Objekt übergeben. Etwa so:

```
IloAnySet allNodes;
IloNode *aNode = new IloNode(x, y, name);
allNodes.add((void*)aNode);
```

Dieses stark vereinfachte Code-Beispiel macht deutlich, dass der Weg etwas umständlich ist. Besser erscheint die Nutzung, der C++ Standardtemplateklasse std::list. Um jedoch das Suchen und Einfügen von Knoten zu kapseln, soll hierzu eine eigene Klasse entworfen werden, welche std::list und „IloNode“ benutzt (Anhang B – Klassendeklarationen).

Die Klassenbibliotheken von ILOG machen sich die Technik von Handle- und Implementationsklassen zu Nutze. Dabei wird die Funktionalität einer Klasse in einer Implementationsklasse verborgen. Die Klassen, die zur Programmierung verwendet werden, sind Handleklassen. Deren Objekte besitzen einen Pointer auf das Implementationsobjekt. Das hat auch Einfluss auf das Kopieren von Objekten. Die Kopie einer Handleklasse zeigt immer noch auf das gleiche Implementationsobjekt!

Es muss also gefragt werden, in wie weit die Speicherverwaltung des Solvers die eigenen Listen-Objekte beeinflussen kann. In der Solver Version 4.4 wurde ausdrücklich darauf hingewiesen, dass man keinesfalls Pointer auf ILOG-Objekte halten

soll. Stattdessen sollten Pointer auf Implementationsobjekte verwendet werden. Weil ein entsprechender Constructor für IloNodes existiert, sollen IloNodeI-Pointer von der Liste verwaltet werden.

8.2.3 MlBreaks

Die auf der Seite 46 beschriebenen Unterbrechungen können ebenfalls in einer Klasse zusammengefasst werden. So soll ein eingrenzender Zeitraum und die Dauer berücksichtigt werden, sowie eine mögliche Zeitverletzung.

Um das Zeitfenster für den Beginn der Pause darzustellen kann hier auf die in Kapitel 8.2.1 beschriebene Klasse zurückgegriffen werden. MlBreak::beginning ist somit vom Typ MlTimeWindow.

Für die Klasse MlBreaks muss berücksichtigt werden, dass sie eine Unterbrechung auf einem gegebenen Fahrzeug deklarieren muss. Die Pause kann an einen bestimmten Ort gebunden sein, was aktuelles Wissen über MlNodes nötig macht. Dies soll durch einen Pointer auf das MlNodes-Objekt erreicht werden, der durch den Constructor übergeben wird.

Weil Pausen immer fahrzeugbezogen sind, wird die Memberfunktion MlBreaks::locateNextOf(vehicleID) eingeführt. Sie soll die nächste Pause eines gegebenen Fahrzeugs ermitteln.

Um die Möglichkeit zu schaffen, Unterbrechungen eines Fahrzeugs zu erzeugen, wird die Funktion MlBreaks::insertBreaks deklariert. Dabei soll das Modell des Fahrzeugs, mit all seinen Constraints und einem IloDimension2-Objekt übergeben werden. Dieses IloDimension2-Objekt enthält für gewöhnlich den zeitlichen Verbrauch zwischen zwei Knoten. Um alle notwendigen Informationen aus dem Modell zu erhalten, benötigt man ein IloDispatcher-Objekt und dazu wiederum ein IloSolver-Objekt. Es genügt sogar ein IloSolver, da ein Dispatcher aus einem Solver erzeugt werden kann. Also erhält die Klasse MlBreaks ein Attribut namens mySolver (Anhang B – Klassendeklarationen – MlBreaks).

8.2.4 MlVisits

Die vordefinierte Klasse IloVisit genügt sicher den meisten Anforderungen. Für das bestehende Klassengerüst genügt sie aber leider nicht. Verschiedene Attribute finden darin keinen Platz. Etwa kann das Constraint „Fahrzeug darf nicht breiter sein als x“ nicht innerhalb eines Modells mit IloVisit-Objekten umgesetzt werden. Man benötigt zur Deklaration zusätzlich ein IloVehicle-Objekt. Aus diesem Grund werden

solche Constraints von dem Manager deklariert. Die Informationen dazu muss die Klasse MlVisit bereitstellen (Anhang B – Klassendeklarationen – MlVisits).

8.2.5 MlVehicles

Eine C++-Klasse, die ein Fahrzeug darstellt, existiert bereits in den ILOG-Bibliotheken. Dabei hat ein Fahrzeug einen festen Anfangspunkt und einen festen Zielpunkt zu dem es am Abend zurückfährt. Es muss daher möglich sein, auf die bestehenden MlNodes zurückzugreifen. Dazu wird das Attribut MlVehicles::myNodes eingeführt.

Ein IloVehicle hat eine bestimmte Kapazität und verursacht bestimmte Kosten. Die Klasse IloVehicle eignet sich deshalb hervorragend zur Abbildung eines realen Fahrzeugs. Leider können die ILOG-Klassen nicht vererbt und erweitert werden. Einige Methoden hätten dann Schwierigkeiten mit der erweiterten Klasse umzugehen. Eine Möglichkeit besteht darin, Objekte an ein ILOG-Objekt zu binden. Dies geschieht durch einen Pointer auf das entsprechende Objekt und der Methode IloVehicle::setObject.

```
MlInfo *myVehicleInfo = new MlInfo();
IloVehicle aVehicle;
aVehicle.setObject((void*)myVehicleInfo);
  ...
myVehicleInfo = (MlInfo*)aVehicle.getObject();
```

In einem Algorithmus könnten diese Informationen entsprechend verarbeitet werden. Eine andere Möglichkeit besteht darin, die unterschiedlichen Eigenschaften durch Constraints abzubilden. Die Informationen für diese Constraints (Breite, Höhe ...) könnten in einer zweiten Klasse parallel zur IloVehicle-Klasse gepflegt werden. Die Handhabung von zwei Klassen (IloVehicle und Mlvehicle) kann durch die Klasse MlVehicles verborgen werden.

Somit können bestimmte Eigenschaften wie Geschwindigkeit oder Kapazität direkt im IloVehicle-Objekt gesetzt werden, zum Anderen können weitere Constraints in Verbindung mit IloVisit oder MlOrder deklariert werden.

Bleibt noch die Frage, wie eine Menge von Fahrzeugen verwaltet werden soll und wie Fahrzeugklassen und -typen behandelt werden sollen (Klasse hier als Menge von realen Fahrzeugen mit bestimmten Eigenschaften; Kapitel 6, Seite 44). Dafür ist zu klären, in welcher Weise auf diese Menge zugegriffen werden könnte. In

einem Algorithmus würde man wahrscheinlich so verfahren, dass man das Fahrzeug „fragt“, ob es zu dieser Klasse gehört und dann entsprechend die Kosten setzt. Ebenso würde man mit der Fahrzeugklassenzugehörigkeit umgehen.
Da die Eingangsdaten auch die Aussage „Alle Fahrzeuge dieser Klasse“ beinhalten können, ist hier gut abzuwägen, welche Datenstrukturen man wählt. Man könnte zu jedem Fahrzeug die Klasse und den Typ speichern und ein entsprechendes Attribut hinzufügen. Dadurch ließe sich die Frage nach dem Typ leicht beantworten, aber es würde schwierig werden, alle Fahrzeuge eines gewählten Typs oder einer bestimmten Fahrzeugklasse zu verplanen. Aus diesem Grund wird hier wiederum auf die std::list-Klasse zurückgegriffen, um eine Menge von Typen und Klassen abzubilden. Ob ein Fahrzeug von einem bestimmten Typ ist, lässt sich mit Hilfe der Klasse MlVehicle feststellen, in der ein entsprechendes Attribut enthalten ist.

Die Klasse MlVehicles muss einen Pointer auf MlNodes besitzen, um Fahrzeuge mit einem gültigen Start und Zielknoten zu erzeugen.

MlVehicles::getIloVehicleAt gibt einen Zeiger auf ein IloVehicle zurück, das sich an der angebenen Position in der Liste befindet. Es erscheint für stark arrayausgerichtetes Arbeiten notwendig, eine solche Funktion mit aufzunehmen. Somit können alle Fahrzeuge mit einem IloVehicleArray verarbeitet werden. Die Funktion MlVehicles::getIloVehicle kann dies nicht leisten, da hier die ID des Fahrzeugs übergeben wird. Diese wiederum hat nur die Einschränkung, numerisch – aber nicht fortlaufend zu sein.

8.2.6 MlOrders

Als weitere Klasse wird das Analogon zum realen Auftrag deklariert. Informationen über Lieferzeitpunkt, Warenart oder zu verwendende Fahrzeugklasse können hieraus entnommen werden. Ebenso sind alle möglichen Beladeknoten und der eine Entladeknoten enthalten. Außerdem liefert ein Auftrag Aufschluss darüber, welche Menge einer bestimmten Ware geliefert werden soll.
Da je Auftrag immer nur eine Fahrzeugklasse festgelegt werden kann, wird dies beispielsweise durch eine Integer-Variable geschehen. Die Fahrzeugtypen können mehrfach auftreten, weshalb hier wiederum auf das std::list-Template zurückgegriffen wird. Ebenso ergeht es den Fahrzeugen des Auftrags und den Beladeknoten. Der Lieferzeitraum wird durch ein MlTimeWindow (Kapitel 8.2.1) repräsentiert.

Die Tatsache, ob das Fahrzeug bereits beladen ist und ob es nach dem Auftrag gereinigt werden muss, wird durch einen Bool'schen Wert realisiert.

8.2.7 MlManager

Die Klasse MlManager enthält Objekte der Ml-Verwaltungsklassen sowie einige Ilo-Objekte wie IloEnv, IloSolver oder IloDispatcher. MlManager bestimmt über den Lebenszyklus all dieser Objekte und kann als zentrale Figur identifiziert werden. Die zahlreichen IloDimensions-Objekte werden in ein eigenes IloModel-Objekt gekapselt.

Die Funktionen der Schnittstelle kommunizieren ausschließlich mit dem MlManager. Er leitet die Berechnung (MlManager::computeSolution) und die Erstellung (MlManager::createAllObjects) eines Modells.

8.3 Format der Eingangsdaten

Die Frage, in welcher Form die Daten eingelesen werden können, erscheint etwas schwierig, vor allem deshalb, weil Datenelemente (wie beispielsweise der Ort einer Pause) einfach weggelassen werden können. In einem Auftrag mögen unbestimmt viele aber mindestens ein Aufladeknoten enthalten sein. Somit scheint eine Datei mit gewöhnlichen Records, ein falscher Gedanke zu sein. Am besten könnte dies durch eine eigne Backus Nauer Form (BNF) abgebildet werden, wobei dann ein BNF-Parser notwendig wäre.

Wenn man bestehende Standards berücksichtigen will, die eine breite Unterstützung erfahren, scheint XML in Verbindung mit XML Schema die beste Lösung zu sein. Mit einem Schema können sehr einfach Datenstrukturen definiert werden, die im XML-File eingehalten werden müssen. Natürlich könnte man auch auf DTD's (Document Type Definition – Files) zurückgreifen, doch bietet ein Schema mehr Freiheiten für die Festlegung der Typen und wird selbst in XML verfasst, wohingegen sich DTD's einer eigenen Definitionssprache bedienen.

Die Existenz des freien XML-Parsers[2] Xerces, der auch XML Schemas verarbeiten kann, erleichtert die Wahl. Somit können die Eingangsdaten sehr gut definiert und vom System eingelesen werden. Eine sehr gute Unterstützung bietet hier das standardisierte DOM (Document Object Model), auf dem verschiedene Funktionen zum Einlesen definiert wurden. DOM präsentiert eine XML-Datei als Baum und stellt gewisse Methoden zum Abrufen der Informationen bereit. Die komplette Definition der Schemas befindet sich im Anhang A - Eingangsdaten.

2 Xerces auf http://xml.apache.org/xerces-c/index.html

8.4 Bemerkungen zu OOA/OOD

In der Analyse wurde ein möglichst schlankes Interface vorgeschlagen. Und es spricht vieles dafür, die Details der Implementation zu verbergen. Deshalb wird sich hier auf ein „loadProblem“ und „computeProblem“ beschränkt. Eine Anwendung der zu erstellenden Bibliothek muss zunächst die Eingangsdaten laden, um sie anschließend berechnen zu lassen.

Die Ladereihenfolge ist äußerst wichtig und wird wie folgt festgelegt: Geographische Knoten->Fahrzeuge->Beladeknoten->Entladeknoten->Aufträge->Unterbrechungen.

Der Manager wird durch MlManager::insertNode, MlManager::insertVehicle usw. gebeten, die entsprechenden DOM-Objekte in eigene Objekte zu wandeln und abzuspeichern. Wenn die Funktion „computeProblem“ aufgerufen wird, werden die eben erzeugten Objekte benutzt, um alle notwendigen Constraints zu erzeugen und die Suche anzustoßen. Das Ergebnis wird durch eine passende Struktur übergeben.

Dieser grobkörnige Algorithmus soll nun im nachfolgenden Abschnitt in einigen Bereichen weiter betrachtet werden. Dabei wird vor allem auf die Erstellung der ersten Lösung und eines Constraints wert gelegt. Abschließend wird die Einbindung des Goals MlGeneratePlan und der zahlreichen Constraints erläutert.

9 Implementierung

Der Rahmen ist zu knapp bemessen, als dass man auf alle Details der Implementierung eingehen könnte. Zu schnell würden die Erläuterungen von Oberflächlichkeit geprägt sein. Statt dessen werden wichtige Höhepunkte des C++-Quellcodes untersucht. Allen voran das Goal zur Erzeugung einer ersten Lösung MlGeneratePlan.

9.1 MlGeneratePlan

Im Kapitel 7 wurde die zweistufige Herangehensweise der ILOG-Bibliotheken festgestellt. Diese Art der Problemlösung erzeugt eine Lösung, und verbessert sie dann. Eine erste Lösung muss noch nicht perfekt sein, sie muss nur gültig sein. Natürlich darf die Herausforderung der anschließenden Verbesserung nicht zu groß sein. Andererseits sollte die erste Lösung schnellstmöglich gefunden werden.

Die Dispatcher-Bibliothek bietet eine kleine Sammlung an Heuristiken, die eine erste Lösung erzeugen können. Leider zeigte sich während der Implementierungsphase, dass keine praxistauglich ist. Bei bestimmten Eingangsdaten funktionieren die vordefinierten Goals sehr gut, wenn aber nur ein Auftrag hinzukommt,

verlangsamt sich die Suche von zwei auf dreißig Minuten! Eine Vorsortierung an verschieden Stellen kann dieses Problem leider nicht wesentlich entschärfen. Auch stört hierbei ein Bug in den Goals sehr. Unter bestimmten Umständen, wird die gesammte Lösung ungültig. Das Problem tritt vor allem in Verbindung mit dem IloIf-Then-Constraint auf. Dieses Constraint setzt die Logikformel $A \rightarrow B$ um. Man kann auf diese Weise fordern, dass ein Entladeknoten nur dann besucht wird, wenn ein Beladeknoten besucht wurde.

```
IloIfThen(myEnv, A.performed(), B.performed());
```

(Natürlich fehlt hier noch eine Aussage darüber, was mit B passiert, wenn A nicht ausgeführt wurde, doch soll das Beispiel schlank bleiben.)
Ohne Anwendung dieser Wenn-Dann-Regel kann eine Lösung gefunden werden. Mit ihr scheitert das Goal. Eigentlich sollten lediglich die IloVisits A und B fehlen. Statt dessen ist das gesamte Ergebnis ungültig.
Es ist etwas traurig, dass dieser Fehler schon in der Vorgängerversion vorhanden war und noch immer nicht ausgebessert wurde. In der Vorgängerversion äußerte er sich nur etwas anders. Dort wurde B einfach verschluckt und tauchte nicht im Ergebnis auf – weder als „nicht erfüllt“ noch als „erfüllt“.

Ein eigenes Goal zur Findung der ersten Lösung ist also notwendig. Wie bereits aus dem Kapitel 7 bekannt ist, verlangen die ILOG-Bibliotheken nach einer Modell- und Algorithmenklasse. Zusätzlich wird jeweils eine Handle- und eine Implementationsklasse benötigt. Somit sind insgesamt vier Klassen nötig.
Für Modell und Algorithmus existiert jeweils eine Klasse IloGoalI bzw. IlcGoalI von denen die Grundfunktionalität geerbt wird. So muss für die Modellklasse die Extraktion des Algorithmenobjektes programmiert werden. Die virtuelle Memberfunktion IlcGoalI::execute wird überschrieben und implementiert den Algorithmus.

Glücklicherweise kann die Handleklasse durch eine Funktion ersetzt werden, die der Implementationsklasse Speicher zuweist, sodass sich für die Modellklasse folgende Schnittstelle ergibt:

```
// Modellklasse
class MloGeneratePlanI : public IloGoalI
{
private:
        IloInt    delivCount;
```

```
public:
         MloGeneratePlanI(IloEnvI*, IloInt);
         IlcGoal extract(const IloSolver) const;
         IloGoalI* makeClone(IloEnvI*) const;     //Copy-Operator
         void display(ILOSTD(ostream&)) const;    //Ausgabe für Debug
};
```

Das private Attribut *delivCount* wird vom Algorithmus benötigt und muss deshalb auch in dieser Klasse ein Gegenstück besitzen. Die Methoden IloGoalI::makeClone und IloGoalI::display sind in diesem Zusammenhang nicht weiter wichtig. Erst durch die Methode IloGoalI::extract wird ein Algorithmenobjekt erzeugt, welches auf den Goalstack gelegt und ausgeführt werden kann. Im Grunde genommen wird, nachdem das Goal vom Stack genommen wurde, die Funktion IlcGoalI::execute ausgeführt. Innerhalb der Methode kann der eigentliche Algorithmus implementiert werden. Die privaten Attribute, die der Algorithmus benötigt, werden weiter unten erläutert. Nichtsdestotrotz zeigt sich die Klassendeklaration wie folgt:

```
// Algorithmenklasse
class MlcGeneratePlanI : public IlcGoalI
{
private:
         IlcIntSet myDeliveries;
         IlcIntArray myVehicles;
         IlcBool  createVisitSet(void);
         IlcBool  createVehicleSet(void);
         IlcBool  createDeliveryListOn(IloVehicle, IlcIntArray);
         IloVisit getNearestDelivery(IloVisit, IloVehicle, IlcIntSet);
public:
         MlcGeneratePlanI(IloSolver, IlcInt);
         IlcGoal execute();
};
```

Die Arbeitsweise der Memberfunktion MlcGeneratePlanI::execute kann in etwa so dargestellt werden:

1. *Erzeuge ein Array mit Entladestationen-ID's.*
2. *Erzeuge ein Array mit Fahrzeug-ID's, wobei das kostengünstigste Fahrzeug auf Index 0 steht.*

3. Wähle das erste unbenutzte Fahrzeug aus und markiere es als benutzt. Setze aktVisit auf den Start-Knoten des gewählten Fahrzeugs. Falls kein unbenutztes Fahrzeug vorhanden ist, gehe zu 7.
4. Erzeuge ein Array mit den übrigen Entladestationen-ID's, die auf diesem Fahrzeug zu diesem Zeitpunkt möglich sind.
5. Falls noch ein ungetesteter Entladeknoten für das Fahrzeug vorhanden ist.
a) Nimm ersten ungetesteten Entladeknoten aus dem fahrzeugbezogenen Array und markiere ihn als getestet.
b) Rufe alle zugehörigen Beladeknoten ab und teste der Reihe nach einen Beladeknoten mit dem Entladeknoten in Verbindung mit aktVisit.
- Bei Erfolg entferne die aktuelle Entladeknoten-ID aus dem Array der gesamten verfügbaren Entladeknoten und setze aktVisit auf die soeben verplante Delivery-Visit.
c) Gehe zu 5.
6. Falls kein ungetesteter Entladeknoten für das Fahrzeug vorhanden ist, gehe zu 3.
7. Schließe jede Tour eines Fahrzeugs.
8. Verplane die Unterbrechungen mit IloInstantiateVehicleBreaks.
9. Markiere alle offenen Visits als „unperformed“.

Die einzelnen Schritte des Algorithmus sollen nun näher betrachtet werden.

9.1.1 Erster Schritt

Um die Fahrzeug-ID's und Visit-ID's zu verwalten bietet sich ein Stack an. So könnte das oberste Element automatisch als nächstes betrachtet werden. Die C++ Templateklasse std::stack scheint dafür wie geschaffen zu sein, doch geht es schneller, eine eigene Stackverwaltung zu benutzen, die auf einem Array beruht. Also wird ein Zeiger auf das aktuelle Fahrzeug und den aktuellen Entladeknoten benötigt.

Zu Beginn werden Objekte vom IloSolver, IloDispatcher, IloEnv und den verschiedenen Stackzeigern initialisiert.

```
IloSolver mySolver = getSolver();
IloDispatcher myDispatcher(mySolver);
IloEnv myEnv = myDispatcher.getEnv();
IlcInt vehiclePointer = 0;
IlcInt visitPointer = 0;
IlcInt state = 0;
```

Danach kann ein Array mit allen Entladeknoten-ID's erstellt werden. Die ID's, die im Modell durch den Solver festgelegt wurden, unterscheiden sich für gewöhnlich von denen der Eingabedaten, sodass hier Obacht gegeben werden muss. Es müssen sogar alle IloVisits erfolgreich extrahiert sein, bevor man auf die ID-Nummer des Objektes zugreifen kann. Sollte von einem nicht extrahierten Objekt die ID-Nummer

verlangt werden, wird eine Exception ausgelöst.

Entsprechend diesen Randbedingungen füllt MlcGeneratePlanI::createVisitSet das Array MlcGeneratePlanI::myDeliveries mit den richtigen ID's der Entlade-Visits.

```
CreateVisitSet();
```

Nun werden die Objekte für das aktuelle Fahrzeug und die aktuelle Visit initialisiert. Ebenso muss das Array für die aktuellen Entladeknoten definiert werden.

```
IloVehicle myVehicle;
IloVisit   myVisit;
IlcIntArray currentDeliveries(mySolver, myDeliveries.getSize());
```

9.1.2 Zweiter Schritt

In diesem Schritt werden die ID's aller verfügbaren Fahrzeuge in ein Array sortiert. Die Ordnungsheuristik (Kapitel 2.5), die dabei angewendet wird, orientiert sich an den Kosten der Fahrzeuge. Dementsprechend werden preiswerte Fahrzeuge (statische Kosten pro Tag) zuerst verplant. Eigenschaften, wie Geschwindigkeit oder Spritkosten werden nicht berücksichtigt.

Die private Memberfunktion MlcGeneratePlanI::createVehicleSet übernimmt das Füllen des Arrays MlcGeneratePlanI::myVehicles.

```
createVehicleSet();
```

Im folgenden Schritt wird ein Fahrzeug vom Stack genommen und entsprechend seines Startknotens die Variable für die aktuelle Visit gesetzt.

9.1.3 Dritter Schritt

Die Implementierung des Algorithmus wird nah an eine Statusmaschine angelehnt. Zu diesem Zweck wird in der Variable „state“ der Status festgehalten. Die verschiedenen Status werden durch eine switch-Anweisung koordiniert. So wird der Schritt 3 durch `state=0` vertreten. Schritt 4 entspricht `state=1` usw.

```
while (true)
{
  switch (state)
  {
  case 0:
```

Sollten Entladeknoten noch unverplant und ungenutzte Fahrzeuge vorhanden sein, wird ein Fahrzeug vom Stack genommen.

Der Startknoten des Fahrzeugs wird zur aktuellen Visit „myVisit“, an die weitere Be- und Entladeknoten geknüpft werden können. Danach wird zum nächsten Schritt gewechselt (`state=1`; Schritt 4).

```
if (vehiclePointer >= myVehicles.getSize())  //alle Fahrzeuge getestet
  state = -1;           //Schleife verlassen
else if (myVehicles[vehiclePointer]<0)  //keine Fahrzeuge mehr da
  state = -1;           //Schleife verlassen
else if (myDeliveries.getSize()>0)  //noch unverplante Aufträge
{
  myVehicle = myDispatcher.getVehicle(myVehicles[vehiclePointer]);
  myVisit   = myVehicle.getFirstVisit(); //aktuelle Visit
  ++vehiclePointer;     //Stackpointer aktualisieren
  state = 1;
}
```

Sollten keine Entladeknoten mehr vorhanden sein, wird die Schleife verlassen (`state = -1`).

```
else
  state = -1;  //Schleife verlassen
break;
```

9.1.4 Vierter Schritt

Die Zuordnung und Sortierung der Entladeknoten für ein gegebenes Fahrzeug übernimmt die Memberfunktion MlcGeneratePlanI::createDeliveryListOn. Sie erzeugt für ein Fahrzeug eine Liste von Entladepunkten, wobei dieselben nach ihrem Spielraum bezüglich der Fahrzeugwahl aufsteigend sortiert werden. Der Entladeknoten mit den wenigsten zulässigen Fahrzeugen steht also auf Index 0. So werden auch Aufträge, die nur ein Fahrzeug zulassen, an das richtige Fahrzeug vergeben. Denn die Constraint-Variable, die die Fahrzeug-ID's in ihrer Domain enthält, wurde bereits beschränkt. Eine Visit, dessen Fahrzeugvariable nur eine Domaingröße von eins hat, steht in jedem Falle vorn und wird als erste verplant. Da die Constraints vor der Ausführung der Goals propagiert werden, ist diese Strategie erfolgreich. Die hier angewandte Heuristik entspricht weitestgehend dem First-Fail-Prinzip.

Das dazu notwendige Array muss an die Funktion übergeben werden. Dabei ist es nicht interessant, ob „by value“ oder „by reference”, denn es wird nur die

Handleklasse übergeben. Der Zeiger auf die Implementationklasse wäre in beiden Fällen identisch. Zur besseren Lesbarkeit wird hier aber per Referenz übergeben.

```
case 1:
  createDeliveryListOn(myVehicle, currentDeliveries);
  visitPointer = 0;
  state = 2;
  break;
```

Im nachfolgenden Schritt werden mögliche Kombinationen, die durch die Mehrzahl von Beladeknoten entstehen, der Reihe nach getestet.

9.1.5 Fünfter und sechster Schritt

Zunächst muss ein möglicher Entladeknoten aus dem Array entnommen werden. Sollte kein Knoten mehr verfügbar sein, wird ein neues Fahrzeug gewählt und erneut der Algorithmus durchlaufen. (`state = 0`; Schritt 3)

```
case 2:
  if (visitPointer >= currentDeliveries.getSize())//keine Aufträge da
    state = 0;   //neues Fahrzeug
  else if (currentDeliveries[visitPointer]<0)//keine Auftrage mehr da?
    State = 0;   //neues Fahrzeug
  else if (currentDeliveries.getSize()>0)  //neuen Entladeknoten
  {
    IloVisit delVisit = myDispatcher.getVisit(
                                  currentDeliveries[visitPointer]);
    ++visitPointer;       //Stackpointer aktualisieren
```

Anschließend wird diese ID aus dem Array mit allen Entladeknoten-ID's entfernt. Dadurch kann dieser Auftrag von keinem anderen Fahrzeug mehr gefahren werden.

```
    myDeliveries.remove(myDispatcher.getIndex(myVisit));
```

Nun kann der Versuch unternommen werden, diese Visit mit seinem Beladeknoten an eine bestehende Kette von Visits anzuhängen. Das Subgoal „MlcLinkPickup“ realisiert dies. Da an jedem Entladeknoten ein Array der zugehörigen Beladeknoten angedockt ist (siehe Seite 56; „setObject“), kann das Subgoal das Array der Beladestationen aus dem Entladeknoten gewinnen. Es ruft dann wiederum ein Subgoal auf, um es zu sortieren.

Eine mögliche Ordnungsheuristik für die Beladestationen könnte diese Stationen

entsprechend ihrer Entfernung zur eigenen Entladestation sortieren. Damit nimmt man an, dass ein zum Entladeknoten naher Beladeknoten günstig für das Gesamtergebnis ist. So würde das Fahrzeug in die Nähe des Entladeknotens fahren, dort beladen werden und erst dann, relativ schwer, die kurze Strecke zum Entladeknoten fahren. Auf diese Weise könnten die Spritkosten verringert werden. Doch leider kann sich der gewählte Beladeknoten fernab einer günstigen Route befinden.

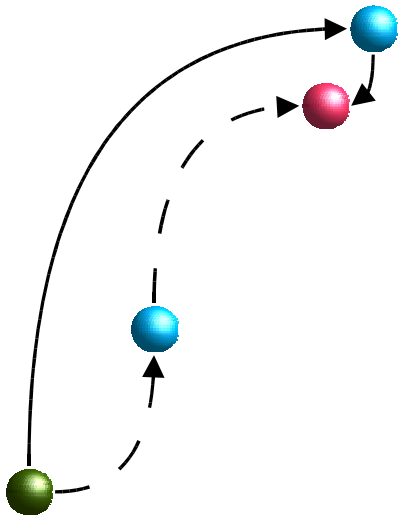


Abbildung 11 Problem der Orderheuristik

Die Abbildung 11 zeigt genau dieses Problem. Der grüne Punkt soll ein bereits verplanter Entladeknoten sein, an dem ein neuer Beladeknoten (blau) und danach der zugehörige Entladeknoten (rot) gekoppelt werden soll. Die gestrichelte Route ist dabei günstiger, als die gewählte durchgezogene Linie.

Eine andere Heuristik könnte sich nach der wirklichen Entfernung zwischen den grünen und roten Punkten richten. Je nachdem, wo der blaue Punkt liegt, wird sortiert. Die Teilroute mit den geringsten Kosten soll als erste untersucht werden. Das Goal „MlcOrderPickups“ verwirklicht genau diese Strategie und sorgt dafür, dass zunächst die gestrichelte Linie verwendet wird. Dazu werden die Entfernungen grün -> blau -> rot verglichen und die Beladeknoten (blau) entsprechend sortiert.

Es sei an dieser Stelle angemerkt dass es sich bei dieser Heuristik um eine in Kapitel 2.5 beschriebene dynamische Ordnungsheuristik handelt. Alle anderen bisher angewandten Ordnungsheuristiken sind statisch. Der notwendige Mehrauf-

wand wird aber belohnt und rechtfertigt sich, wenn eine geringe Anzahl von Beladestationen vorliegen.

Das Goal, welches die die Sortierung benutzt und die Teilrouten einfügen möchte, benötigt neben dem Fahrzeug, der aktuellen Visit und der neu gewählten Delivery-Visit, den Index für die erste Pickup-Visit im Pickup-Array (in diesem Falle die 0).

```
      if (mySolver.solve(MlcLinkPickup(mySolver,
                              myVehicle,         //aktuelles Fahrzeug
                              myVisit,           //grüner Knoten
                              delVisit,          //roter Knoten
                              0)))               //erster blauer Knoten
```

Durch das Goal MlcLinkPickup wird ein Choice Point gesetzt. Das erste Subgoal MlcLinkVisit versucht die aktuelle Teilroute an die bestehende Route anzuhängen. Schlägt dies fehl wird einfach MlcLinkPickup mit dem nächsten Index des Pickup-Arrays aufgerufen, da hier der Choice Point greift.

```
++pickup;

return IlcOr(          //Choice Point erzeugen
  MlcLinkVisit(solver, //zuerst versuchen, Route um Teilroute zu erweitern
    thisVehicle,       //auf diesem Fahrzeug
    firstVisit,        //grüner Knoten
    nextVisit,         //ein blauer Knoten
    delivVisit),       //roter Knoten
  MlcLinkPickup(solver, //rekursiver Aufruf
    thisVehicle,       //Fahrzeug
    firstVisit,        //grüner Knoten
    delivVisit,        //roter Knoten
    pickup));          //neuer Index für blauen Knoten
```

Kann MlcLinkPickup erfolgreich abschließen, wird die letzte Visit der Route zur aktuellen Visit. In Hinblick auf Abbildung 11 wird der rote Punkt grün gefärbt.

```
      {
        myVisit = delVisit; //letzte Visit zur aktuellen machen
      }
```

Sollte MlcLinkPickup fehlschlagen, muss die Visit-ID des gewählten Entladeknotens den verbleibenden Fahrzeugen wieder zur Verfügung gestellt.

```
      else
      {
        myDeliveries.add(myDispatcher.getIndex(myVisit));
      }
    }
```

Welches Ergebnis dieses Goal auch liefert, es wird solange im Status 2 verblieben, bis alle Entladeknoten des Fahrzeugs getestet wurden. Danach muss ein neues

Fahrzeug gewählt und in den Status 0 gewechselt werden:

```
    else           //alle Delivery-Visits auf diesem Fahrzeug untersucht
      state = 0;   //neues Fahrzeug
    break;
  }
```

9.1.6 Siebenter Schritt

Wenn alle Fahrzeuge und deren Entladeknoten samt Beladeknoten untersucht wurden, kann die While-Schleife verlassen werden.

```
  if (state == -1)
    break;
}
```

An dieser Stelle wurden alle möglichen Aufträge entsprechend der Ordnungsheuristik untersucht. Die Touren der Fahrzeuge sind in gewisser Weise offen, denn es fehlt die Verbindung von der letzten Entladestation zum Heimatstandort des Fahrzeugs.

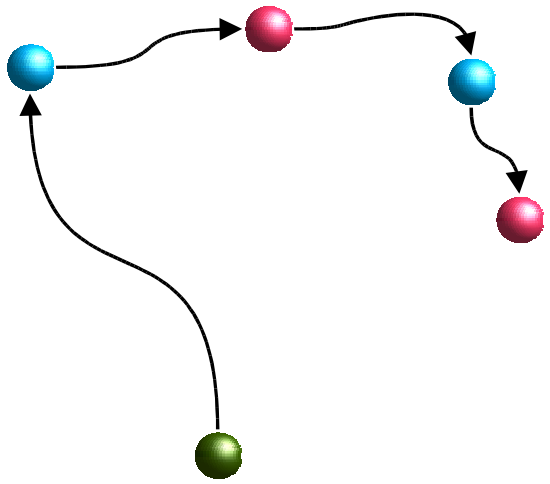


Abbildung 12 Offene Tour

In Abbildung 12 ist eine offene Tour eines Fahrzeugs zu sehen. Der Startknoten des Fahrzeugs ist hier grün dargestellt und bezeichnet gleichzeitig seinen Zielknoten oder Heimatstandort. Das Schließen kann die Funktion IloGenerateRoute der Dispatcher-Bibliothek übernehmen. Sie gibt ein Goal zurück, welches die letzte Visit einer Tour mit dem Endpunkt des Fahrzeugs verbindet. Auf jedes Fahrzeug ange-

wandt, kann es folgendermaßen implementiert werden:

```
for (IloIterator<IloVehicle> vhi(myEnv); vhi.ok(); ++vhi)
{
  IloVehicle vehicle = *vhi;
  if (!mySolver.solve(IloGenerateRoute(mySolver, vehicle)))
  {
    fail();// das gesamte Goal soll scheitern
  }
}
```

Sollte IloGenerateRoute fehlschlagen, kann das gesamte Goal MlcGeneratePlan als Fehlschlag betrachtet werden. Jedoch sollte dies nicht passieren, da das erfolgreiche Schließen einer Route nur noch davon abhängen kann, ob ein Fahrzeug noch vor dem Feierabend (`vehichle.getEndVisit().getCumulVar(time) <= closeTime`) am Depot sein kann. Aber genau das wurde von MlcLinkVisit und damit von MlcLinkPickup überprüft.

9.1.7 Achter Schritt

In diesem Schritt werden die Pausen verplant, wobei die darin investierte Berechnungszeit begrenzt wird. Es werden maximal 100 Aufrufe von IlcGoalI::fail() zugelassen.

```
mySolver.solve(IloLimitSearch(mySolver,
  IloInstantiateVehicleBreaks(mySolver, 0.0, IloFalse),
  IloFailLimit(mySolver, 100)));
```

9.1.8 Neunter Schritt

Zum Schluss müssen alle unverplanten Visits als solche markiert werden. Dazu prüft das Goal MlcUnperformVisits jede Visit, ob sie von einem Fahrzeug angefahren wird, und markiert sie gegebenenfalls als „unperformed“. Der Einfachheit halber wird das Goal als einziges Subgoal ausgeführt.

```
return MlcUnperformVisits(mySolver);
```

9.2 MlProduct2ProductConstraint

Die im Kapitel 6 auf der Seite 43 beschriebene Zuladung von Transportgütern ist nur dann möglich, wenn das geladene Produkt mit dem neuen Produkt übereinstimmt und das Fahrzeug noch genügend Kapazität besitzt. Bezogen auf die eingangs

erwähnte Fahrradflotte dürfte ein Kurier nur dann einen Sack Tomaten in seinen Rucksack packen, wenn dieser genügend Raum bietet und bereits Tomaten enthält. Es könnte etwas problematisch werden, wenn sich im Rucksack bereits Expressbriefe befinden. Völlig gegenstandslos wird diese Einschränkung aber, wenn der Rucksack leer ist. In diesem Fall darf er mit jedem kompatiblen Transportgut gefüllt werden. Die Produktverträglichkeit darf also nur bei einem teilbeladenen Stauraum beachtet werden, was die Deklaration eines entsprechenden Constraints erschwert. Aus diesem Grund ist ein vollkommen neues Constraint erforderlich, welches hier näher betrachtet wird.

Im vorangegangenen Kapitel wurde festgestellt, dass Constraints vor Goals propagiert werden. Genauer gesagt, werden sie bereits propagiert, nachdem sie deklariert wurden oder eine zugehörige Constraintvariable geändert wurde. Constraints können somit auch während der Ausführung eines Goals propagiert werden. Die Verbindung zwischen Constraintvariable und Constraint-Objekt stellt im ILOG-Solver die Methode IlcConstraintI::post her. Innerhalb dieser Methode wird eine Ereignisfunktion an das Constraint gebunden:

```
myDispatcher.getCumulVar(myVisit, myDim).whenRange(this);
```

Nach dem Aufruf von post wird sofort IlcConstraintI::propagate aufgerufen. Innerhalb von propagate werden die Domains der Variablen eingeschränkt.

Die Strategie, die in diesem Constraint-Objekt verfolgt wird, ist es, die unmittelbaren Nachfolger der gegebenen Visit einzuschränken, wenn das Fahrzeug eine Ladung enthält. Denn sollte ein Fahrzeug bereits beladen sein, darf es nur noch zu einem Beladeknoten fahren, welches das identische Transportgut führt, oder zu einem korrekten Entladeknoten. Ist das Fahrzeug leer, darf es überall hinfahren. Offenbar muss dieses Constraint nur die Beladestellen überwachen, denn nur dort kann ein Fahrzeug mit einem zweiten Transportgut beladen werden.

Zum posten des Constraints stehen sicher mehrere Möglichkeiten zur Auswahl. Man könnte z.B. das Constraint an die Next-Variable binden, die den Index der nachfolgenden Visit enthält. Wenn ein Nachfolger gefunden wurde, wird für diese Variable das „whenValue“-Event ausgelöst. Doch erscheint es einfacher, die Ladung selbst zu überwachen. Dazu wird das Constraint an das IloDimension1-Objekt gebunden, welches die Ladung (z.B. ldWeight oder ldVolume) repräsentiert. Immer wenn sich die Ladung vergrößert oder verkleinert, soll das Constraint propagiert werden.

Das Constraint soll die Next-Variable der aktuellen Visit manipulieren, um falsche Nachfolge-Visits auszuschließen. Die Next-Variable stellt durch ihre Domain alle möglichen Nachfolge-Visit-ID's dar. Um ein Label aus einer Domain zu entfernen, kann die Funktion removeValue verwendet werden. Sollten mehrere Labels entfernt werden, bietet sich removeRange mit den Intervallgrenzen min, und max an. Doch ist dies noch nicht sehr befriedigend, da sicher kein zusammenhängender Wertebereich erwartet werden kann. Die Verwendung von removeValue, für jedes unpassende Label, erzeugt jedesmal ein whenValue-Event, welches wiederum unnötige Propagierungen hervorruft. Besser er ist deshalb, die Domain der Next-Variable, mit einer vorbereiteten Domain gleichzusetzten. Genauer gesagt, werden alle Labels entfernt, die in der anderen Domain nicht enthalten sind. Die Domain einer Variable, die alle erlaubten Nachfolge-ID's enthält, lässt sich aus einem IloIntArray konstruieren, sodass die Modell-Klasse ein solches Array besitzen muss. Die entsprechende Visit und das IloDimension1-Objekt müssen ebenfalls von der Modellklasse MloProduct2ProductConstraintI gespeichert werden. Die Deklaration der Modellklasse sieht damit folgendermaßen aus:

```
class MloProduct2ProductConstraintI : public IloCPConstraintI
{
private:
  IloIntArray      _successors;
  IloVisit         _visit;
  IloDimension1    _dim;
public:
  MloProduct2ProductConstraintI(IloEnvI*, const IloIntArray &,
                                IloVisit &, IloDimension1 &);
  IloExtractableI* makeClone(IloEnvI*) const;//geerbte Funktion ...
  void display(ostream & out) const;         // ...
  IlcConstraint extract(const IloSolver) const; // ... überschreiben
};
```

Die abstrakten Methoden IloConstraintI::makeClone, IloConstraintI::display und IloConstraintI::extract haben die gleiche Bedeutung, wie die eines Goals und sollen hier nicht näher betrachtet werden. Wesentlich interessanter ist die Implementationsklasse MlcProduct2ProductConstraintI.

```
class MlcProduct2ProductConstraintI : public IlcConstraintI
```

```
{
private:
  IloSolver        mySolver;
  IloDispatcher    myDispatcher;
  IloVisit         myVisit;
  IlcIntVar        mySuccessors;
  IloDimension1    myDim;
public:
  MlcProduct2ProductConstraintI(IloSolver, IlcIntVar,
                                IloVisit &, IloDimension1 &);
  void post();                        //geerbte Memberfunktionen ...
  void propagate();                   // ...
  IlcBool isViolated() const;         // ... überschreiben
};
```

Neben den obligatorischen Objekten von IloSolver und IloDispatcher enthält diese Klasse die Constraintvariable „mySuccessors“. Sie wird innerhalb der Methode MloProduct2ProductConstraintI::extract aus dem Array _successors erstellt und hat in seiner Domain nur noch Indizes kompatibler Be- und Entladeknoten dieser Visit. Somit ist es möglich mit einem Befehl mehrere falsche Beladestationen aus einer Domain zu entfernen.

Standardmäßig wird sofort nachdem ein IlcConstraint-Objekt erzeugt wurde, die Methode IlcConstraintI::post ausgeführt. Für MlcProduct2ProductConstraint sieht diese Methode folgendermaßen aus:

```
void MlcProduct2ProductConstraintI::post()
{
  myDispatcher.getCumulVar(myVisit, myDim).whenRange(this);
}
```

Mit Hilfe des IloDispatcher-Objektes wird die Constraint-Variable aus der Constraintumgebung IloEnv beschafft. Die Funktion IloDispatcher::getCumulVar gibt die Referenz auf ein IloFloatVar-Objekt zurück, welches mit diesem Constraint verbunden wird. Statt „whenRange“ stehen noch „whenValue“ und „whenDomain“ zur Verfügung. Bis auf das „whenValue“-Event darf hier alles benutzt werden, denn „whenValue“ tritt nur dann auf, wenn das Fahrzeug die gesammte Kapazität ausgeschöpft hat. Das Event „whenRange“ wird nur ausgelöst, wenn ein Label vom Rand

des Domainintervalls verschwindet – „whenDomain“ dagegen immer. Natürlich werden sich bei einer Ladung nur die Grenzen ändern, sodass es egal ist, welche der beiden möglichen Events zum Posten verwendet wird.

Nachdem die Memberfunktion MlcProduct2ProductConstraintI::post ausgeführt wurde, wird automatisch MlcProduct2ProductConstraintI::propagate aufgerufen um eine erste Verringerung des Suchraumes durchzuführen.

```
void MlcProduct2ProductConstraintI::propagate()
{
  //wenn das Fahrzeug _beladen_ weiterfährt ...
  if ((myDispatcher.getCumulVar(myVisit, myDim).getMin() > 0))
  {  //...lege die Nachfolger fest
    myDispatcher.getIntVar(myVisit.getNextVar()).setDomain(mySuccessors);
  }
}
```

Durch das Setzen der Next-Variable werden einige Constraints propagiert. In einigen Fällen ist es nur nötig zu prüfen, ob ein Constraint ungültig geworden ist. Dazu muss die abstrakte Memberfunktion IlcConstraintI::isViolated überschrieben werden.

```
IlcBool MlcProduct2ProductConstraintI::isViolated() const
{
  IloVisit vHelp(myVisit);
  while (myDispatcher.getPrevVar(vHelp).isBound())
  {      //auf Vorgänger wechseln:
    vHelp = myDispatcher.getVisit(
                          myDispatcher.getPrevVar(vHelp).getValue());
         //wenn beladen -> korrekte Mischung ?
    if (!mySuccessors.isInDomain(myDispatcher.getIndex(vHelp)) &&
        ((vHelp.getCumulVar(myDim).getMin() +
          vHelp.getTransitVar(myDim).getMin()) >0))
      return IlcTrue;
         //Fahrzeug ist leer -> keine Abhängigkeiten mehr
    if (vHelp.getCumulVar(myDim).getMin()
                        + vHelp.getTransitVar(myDim).getMin() == 0)
       break;
  }
  return IlcFalse;          //Constraint nicht verletzt
}
```

In diesem Fall soll isViolated die letzten Beladungen überprüfen. Kann die Iteration der While-Schleife bis zu einem leeren Fahrzeug geführt werden, gilt das Constraint als erfüllt. Wird aber vorher festgestellt, dass ein falsches Produkt geladen wurde, muss das Constraint „nicht erfüllt“ zurückgeben.

Nach den beiden Abschnitten über Constraint- und Goal-Klassen-Programmierung, kann nun untersucht werden, wie von ILOG vordefinierte Constraints deklariert werden und wie ein Modell stückweise erweitert werden kann.

9.3 Constraints deklarieren

In der Memberfunktion MlManager::createAllObjects werden vorwiegend Constraints deklariert, weshalb sich eine genauere Betrachtung lohnt. Jedoch soll nicht jede einzelne Code-Zeile separat untersucht werden.

Entsprechend einer Goldenen Regel der Constraintprogrammierung sollen die Constraintvariablen an Anzahl so gering wie möglich gehalten werden. Das wird dadurch erreicht, indem alle Fahrzeuge und Aufträge einzeln betrachtet werden, ob sie in sich konsistent sind. Dazu wird für jedes Fahrzeug ein Modell erstellt und die fahrzeugbezogenen Constraints untersucht. Kann dabei ein Problem entdeckt werden, wird dieses Fahrzeug generell von weiteren Untersuchungen ausgeschlossen.

Die Constraints, die hierfür in Frage kommen, beziehen sich auf Einsatzzeit und Entfernung zwischen Start- und Zielknoten (sie müssen nicht übereinstimmen). Diese Knoten wurden bereits während des Einlesens durch die Klasse MlVehicles erzeugt. Das Modell eines Fahrzeuges wurde jeweils an das IloVehicle-Objekt gedockt. An dieser Stelle muss nur noch das Modell extrahiert und vom Solver gelöst werden.

```
mySolver.extract(vhModel);          //Fahrzeugmodell extrahieren

IloGoal goal = IloLimitSearch(myEnv,          //das Goal
  IloGenerateRoute(myEnv, iloVehicle),
  IloFailLimit(myEnv, 100));

       //das Modell lösen und Änderungen rückgängig machen (...IlcTrue)

if (mySolver.solve(goal, IloSynchronizeAndRestart, IloTrue))

{
```

Bei Erfolg wird das Fahrzeug samt seiner Constraints in einem Array gespeichert:

```
  modVehicles.add(vhModel);

}
```

Nachdem diese Prüfung eine einwandfreie Fahrzeugflotte hinterlassen hat, können

die Visits der Be- und Entladeknoten untersucht werden.
Dazu wird erneut ein Modell je Auftrag erzeugt und anschließend vom Solver gelöst. War dies möglich, kann das Modell des Auftrags in das Gesamtmodell eingefügt werden. Eventuelle Duplizitäten, wie doppelte Fahrzeuge werden automatisch aus dem Gesamtmodell eliminiert. Die Be- und Entladung stellt für den Solver ebenfalls ein Constraint dar. Mit dem Anfahren einer Visit soll das Fahrzeug eine Änderung des IloDimenstion1-Objektes durchführen.

```
thisModel.add(iloPickup.getTransitVar(ldWeight) == fWeight);
thisModel.add(iloDelivery.getTransitVar(ldWeight) == -fWeight);
```

Zur Einhaltung der Öffnungszeiten muss das Fahrzeug mindesten ein Zeitfenster einhalten. Dazu wird eine Disjunktion über alle möglichen Zeitfenster gebildet und als Constraint in das Modell eingefügt:

```
IloOr pickTime(myEnv);      //Disjunktions-Objekt erzeugen
myPickups.get(pID)->setOpenTimeToBegin();
while (myPickups.get(pID)->getNextOpenTime(from, to))//solange noch
{                                                    //TimeWindows da sind
  pickTime.add(from <= iloPickup.getCumulVar(time) <= to);
}
thisModel.add(pickTime);    //Öffnungszeiten hinzufügen
```

Mit den Zeiten des Entladeknotens geht man identisch vor, nur dass zusätzlich die Lieferzeiten eingehalten werden müssen. Aus diesem Grund müssen für einen Entladeknoten zwei IloOr-Objekte erzeugt werden. Beide in das Modell eingefügt, bilden eine Konjunktion. Jedes Constraint, welches durch IloModel::add in ein einziges Modell eingefügt wurde, muss eingehalten werden und wird somit als Und-Verknüpfung angesehen. Ebenso geschieht das mit dem Constraint zur Einhaltung der Reihenfolge IloVisit::isAfter oder IloVisit::isBefore:

```
thisModel.add(iloPickup.isBefore(iloDelivery));
```

In jedem Fall muss verlangt werden, dass Be- und Entladeknoten auf der selben Tour liegen und mit dem selben Fahrzeug gekoppelt sind.

```
thisModel.add(iloDelivery.getVehicleVar() == iloPickup.getVehicleVar());
```

Wenn das Fahrzeug laut der Eingangsdaten bereits beladen ist, steht bereits das Fahrzeug und der erste Beladeknoten fest und es kann der Beladeknoten an das Fahrzeug durch ein Constraint gebunden werden.

```
thisModel.add(iloVehicle.getFirstVisit().getNextVar() == iloPickup);
```

Etwas schwieriger ist die Deklaration der Constraints, die in Abhängigkeit vom gewählten Fahrzeug und dem noch zu wählenden Be- oder Entladeknoten erzeugt werden müssen. Man könnte für jede Eventualität ein Constraint einfügen. Vielleicht folgendermaßen:

```
IloIfThen(myEnv, iloDelivery.getVehicleVar() == iloVehicle,
   iloPickup.getDelayVar(time) == myVehicles.get(vID)->getCleaningTime());
```

In diesem Beispiel wird eine Reinigungszeit hinzugerechnet, wenn der Auftrag gerade mit diesem Fahrzeug befördert wird. Die Reinigungszeit müsste für jede Fahrzeug-Auftrag-Kombination deklariert werden. Bei zwanzig Aufträgen und ebenso vielen Fahrzeugen wären dies bereits vierhundert zusätzliche Constraints. Gewiss ist es sehr positiv, wenn möglichst viele Constraints deklariert werden (selbst redundante), doch sind Konditionelle Constraints problematisch. Sie beschränken erst dann eine Domain, wenn der Fall auch tatsächlich eintritt. Dazu ein kleiner Vergleich:

Wenn-Dann-Regel	**Ohne Wenn-Dann-Regel**

`IloIfThen(myEnv,`
`  iloPickup.performed(),`
`  iloPickup.isBefore(iloDelivery));`
	`iloPickup.isBefore(iloDelivery);`

`cout << iloPickup.getNextVar();`	`cout << iloPickup.getNextVar();`
`[0...568]`	`[3]`

Diese Zeilen könnten der Ausschnitt eines realen Programms sein. Die Linke Seite zeigt deutlich, dass die Next-Variable noch nicht beschränkt wurde und somit 569 Möglichkeiten erlaubt. Die Next-Variable der rechten Seite hat dagegen bereits ein Label mit dem Wert 3.

Deshalb und weil aus Programmierersicht ein anderer Weg vielversprechender ist, wird diese Idee verworfen. Statt dessen wird eine Bewertungsfunktion erstellt, die den Zusammenhang zwischen IloVisit und IloVehicle numerisch herstellen kann. Somit lassen sich Kosten, die aus einer bestimmten Fahrzeug-Auftragsverbindung resultieren darstellen. Bezogen auf die Fahrradflotte kann nun ausgedrückt werden, dass ein Tomatentransport mit einem Velotaxi besonders hohe Kosten verursacht, ein Tourist dagegen weniger. Wenn beispielsweise der Auftrag ein Fahrzeug des Typs 3 vorsieht und auch tatsächlich durch ein solches Fahrzeug ausgeführt wird, sind die zusätzlichen Kosten gleich null. Wenn statt dessen Typ 26 verwendet wird, erhöhen sich die Kosten durch einen Faktor.

Die Fahrzeugbewertungsfunktion sieht ein Fahrzeug als besonders billig an, wenn es genau der Forderung des Auftrages entspricht, also: *„verwendetes Fahrzeug = gefordertes Fahrzeug“*. Die Kosten werden etwas höher, wenn das Fahrzeug eine andere ID hat, als gefordert, aber zumindest einer der geforderten Typen entspricht. Die Kosten steigen wiederum ein wenig, wenn nur noch die Fahrzeugklasse übereinstimmt.

Eine solche Bewertungsfunktion setzt zwei Arrays voraus. Das eine muss mit den Fahrzeug-Objekten gefüllt sein und das andere muss die Werte enthalten, die bei diesem Fahrzeug Anwendung finden sollen. Außerdem müssen die Indizes übereinstimmen. Das Array der Fahrzeuge soll hier als gegeben betrachtet werden und heißt „arVehicles“.

```
IloNumArray visitCost(myEnv, arVehicles.getSize());//Werte-Array erzeugen
for (k = 0; k < arVehicles.getSize(); ++k)
{
```

Bei der Implementierung wird ein Faktor manipuliert, mit dem die statischen Kosten des Fahrzeugs multipliziert werden. Wenn das Fahrzeug der Vorgabe entspricht, bleibt der Faktor (costFactor) auf eins stehen, sonst wird er erhöht.

```
  short costFaktor = 1;
```

Stimmt die Klasse nicht überein, werden übermäßig viel Kosten produziert und der Faktor um 50 erhöht.

```
  costFactor += myVehicles.getClassIdOf(arVehicles[k].getName()) ==
                                    myOrder->getVhClassID()?0:50;
```

Die Fahrzeugtypen werden ebenfalls überprüft. Sollte ein passender Typ dabei sein, müssen die Kosten nicht weiter erhöht werden. Das Gleiche gilt für die Fahrzeug-ID.

```
  short tempFactor = 2; //der Kostenfaktorzuwachs für einen falschen Typ
  myOrder->setTypeToBegin();
  while (!myOrder->typesEnd())  //alle Typen durchsuchen
  {
    if (myVehicles.getTypeIdOf(arVehicles[k].getName()) ==
                                              myOrder->getVhTypeID())
    {
      tempFactor = 0; //den passenden Typ gefunden
      break;
    }
```

```
  }
  costFactor += tempFactor;
  tempFactor = 1;           //die Kosten für ein falsches Fahrzeug
  myOrder->setVehicleToBegin();
  while (!myOrder->vehiclesEnd()) //alle Fahrzeuge überprüfen
  {
    unsigned int vID = atoi(arVehicles[k].getName());
    if (vID == myOrder->getVehicleID())
    {
      tempFactor = 0; //das richtige Fahrzeug
      break;
    }
  }
  costFactor  += tempFactor;
```

Anschließend können für diese Fahrzeug-Auftrags-Kombination die Kosten bestimmt werden.

```
  visitCost[k] = arVehicles[k].getCost() * (IloNum)costFactor;
}
```

Nachdem die Kosten in beschriebener Weise errechnet wurden, kann die Kostenfunktion erzeugt werden.

```
IloVehicleToNumFunction vehicleCost(myEnv, arVehicles, visitCost);
```

Eine Constraintvariable übernimmt die Kosten und setzt sie in einem IloDimension1-Objekt ein:

```
IloNumVar vhPenalty(myEnv, 0, IloInfinity);
thisModel.add(vhPenalty >= vehicleCost(iloDelivery.getVehicleVar()));
thisModel.add(iloDelivery.getTransitVar(vhCost) == vhPenalty);
```

Wie die Konstruktion der Fahrzeug-Auftrags-Kostenfunktion (eine IloVehicleToNumFunction) impliziert, können weit mehr Probleme durch Kostenfunktionen dieser Art gelöst werden. So könnte dieser Mechanismus auch zum Beschränken der Fahrzeuge auf bestimmte Knoten anhand des Schallpegels eingesetzt werden.
Mit Hilfe einer Schleife wird ein Array mit den Werten der Fahrzeugschallpegel gefüllt.

```
arVhSoundLevel.add(myVehicles.get(iloVehicle.getName())->getSoundLevel());
```

Mit dem so erstellten Array und dem vorhandenen Fahrzeug-Array „arVehicles“ kann wiederum eine IloVehicleToNumFuction erstellt werden.

```
IloVehicleToNumFunction vehicleSoundLevel(myEnv,
                                    arVehicles, arVhSoundLevel);

thisModel.add(myPickups.get(pID)->getSoundLevel() >=
                     vehicleSoundLevel(iloPickup.getVehicleVar()));
```

Gewiss muss dieses Constraint für jede Visit (Be- und Entladeknoten) deklariert werden, doch ist der Programmieraufwand gering. Ebenso lassen sich die Constraints für Höhe, Breite und Länge deklarieren.

Für die Einbeziehung der zeitlichen Verzögerungen bei einer Reinigung oder einer Entladung, müssen mehrere Kostenfunktionen definiert werden. Mit der Summe dieser Zeitkosten kann dann das IloDimension1-Objekt „time“ belastet werden.

```
if (myOrder->needCleaning())
  thisModel.add(iloDelivery.getDelayVar(time) ==
```

Die Zeitkosten, die von dem jeweiligen Knoten abhängen (hier ein Entladeknoten), bilden den ersten Summand.

```
         myDeliveries.get(dID)->getDelay()
```

Die Kostenfunktion der fahrzeugtypischen allgemeinen Zeitkosten muss an jeder Position einbezogen werden:

```
         + vehicleTimeDelay(iloDelivery.getVehicleVar())
```

Die Reinigungszeit häng ebenfalls vom Fahrzeug ab und wird nur in Verbindung mit einem Entladeknoten betrachtet.

```
         + vehicleCleaningTime(iloDelivery.getVehicleVar())
```

Die notwendige Zeit, um eine Entladung vorzubereiten, wird durch folgende IloVehicleToNumFunction zum Ausdruck gebracht.

```
         + vehicleUnloadLag(iloDelivery.getVehicleVar())
```

Um die Zeitkosten je Wareneinheit zu berücksichtigen, muss noch mit der Ladungsmenge multipliziert werden, in diesem Falle mit „fWeight“. Um die tatsächlichen zeitlichen Entladekosten der gesamten Masse zu erhalten, muss der Zeitverbrauch je Liter ebenfalls berücksichtigt werden.

```
         + vehicleULPerLitre(iloDelivery.getVehicleVar())*fVolume
         + vehicleULPerGramm(iloDelivery.getVehicleVar())*fWeight);
```

Entsprechend müssen Aufträge ohne Reinigungsflag berücksichtigt werden.

```
else

  thisModel.add(iloDelivery.getDelayVar(time) ==
         myDeliveries.get(dID)->getDelay()
         + vehicleTimeDelay(iloDelivery.getVehicleVar())
         + vehicleUnloadLag(iloDelivery.getVehicleVar())
         + vehicleULPerLitre(iloDelivery.getVehicleVar())*fVolume
         + vehicleULPerGramm(iloDelivery.getVehicleVar())*fWeight);
```

Man hat bereits gesehen, dass eine IloVehicleToNumFunction mit einer anderen addiert werden kann. Die Multiplikation mit einem Faktor ist ebenso möglich und erlaubt die Umsetzung der im Kapitel 6 auf der Seite 45 hergeleiteten Funktionen für Kosten je Meter und je Stunde.

$$x\frac{(y_2-y_1)}{x_2}+y_1=y$$

Geht man von den Kosten je Meter, also dem Spritverbrauch aus, ergibt sich folgender Faktor:

$$\frac{(Kosten_{beladen}-Kosten_{unbeladen})}{maximale\,Fahrzeugkapazität}$$

Daraus folgen diese Code-Zeilen für das Füllen des Arrays mit dem Faktor:

```
arFakMa.add(
   (myVehicles.get(iloVehicle.getName())->getCostPerMeterLoaded()
    - myVehicles.get(iloVehicle.getName())->getCostPerMeterUnloaded())
  / iloVehicle.getCapacity(ldWeight));
```

Ein zweites Array muss mit $y_{1,}$ den Kosten für das unbeladene Fahrzeug, erzeugt werden. Aus beiden Arrays und dem Array der Fahrzeugobjekte können zwei IloVehicleToNumFunctions erzeugt werden. Diese werden an jede Visit gebunden um die Kosten zu berechnen.

Die so erstellte Faktor-Funktion (vhMeterFaktor) und die zugehörige Summand-Funktion (vhMeterN) können nun zur Berechnung der wirklichen Spritkosten herangezogen werden. Dazu wird der Faktor mit der wirklichen Masse der Ladung multipliziert. Das Ergebnis wird mit dem Summand aus der IloVehicleToNumFunction addiert und anschließend mit dem Weg zum nächsten Knoten multipliziert. Eine Constraintvaribale übernimmt auch hier den Wert, der dann an ein entsprechendes IloDimension1-Objekt gebunden werden kann.

```
thisModel.add(fvMileageP == iloPickup.getTravelVar(length)  //Weg
                  * (vhMeterFaktor(iloPickup.getVehicleVar()) //Faktor
                    * (iloPickup.getCumulVar(ldWeight) + fWeight) //Ladung
                    + vhMeterN(iloPickup.getVehicleVar()))); //N

thisModel.add(iloPickup.getTransitVar(vhMileage) == fvMileageP);
```

Die Funktion IloVisit::getTravelVar(IloDimension2) gibt die Kosten zum nächsten Knoten an, in diesem Fall also die Entfernung in Metern. Um die Ladungsmenge zu erhalten, die das Fahrzeug zum nächsten Knoten befördert, muss zur aktuellen Ladung (IloVehicle::getCumumlVar(IloDimension1)), die Ladungsänderung an der gegebenen Position eingerechnet werden. Denn IloVehicle::getCumulVar gibt nur den Wert zurück, der mit dem Eintreffen des Fahrzeugs an dieser Position Gültigkeit hat. Da aber die Kosten zum nächsten Knoten interessant sind, muss die Änderung einbezogen und fWeight aufaddiert werden.

Abschließend zu diesem Kapitel soll auf das Constraint zur Einhaltung der Beladeknotenkardinalitäten eingegangen werden, das bei einer Auswahl von mehreren Beladeknoten zum Einsatz kommt. Dieses Constraint sorgt dafür, dass je Entladeknoten nur ein Beladeknoten verplant wird. Zu diesem Zweck werden alle Beladeknoten eines Auftrages in ein Array gespeichert. Das Pickup-Array muss auf dem Heap des IloEnv-Objektes erzeugt werden:

```
IloVisitArray *arPickups = new (myEnv) IloVisitArray(myEnv, 0);
```

Die Verarbeitung aller Beladeknoten kann in einer While-Schleife stattfinden.

```
myOrder->setPickupToBegin();
while (!myOrder->pickupEnd())  //alle Beladeknoten überprüfen
{
```

Nachdem die ID des Beladeknotens in der Variable pID gespeichert wurde, kann ein IloNode-Objekt und damit ein IloVisit-Objekt erzeugt werden. Der Name der Visit wird mit einem Präfix gekennzeichnet, sodass sie als Beladeknoten identifiziert werden kann.

```
  unsigned int pID = myOrder->getNextPickupID();
  IloNode  iloNode(myNodes.get(myPickups.get(pID)->getNodeID()));
  IloVisit iloPickup(iloNode,
    std::string(PREPICKUP).append(myPickups.get(pID)->getName()).c_str());
```

Das Einfügen in das Array der eben erzeugten Visit ist trivial.

```
  arPickups->add(iloPickup);
}
```

Das Zählen der verwendeten Beladenknoten soll auf eins reduziert werden. Genauer gesagt soll die Anzahl der Beladeknoten gleich der Anzahl der Entladeknoten sein. Denn so ist es erlaubt, einen Auftrag nicht zu erfüllen, was durchaus vorkommt. Das

Zählen der durchgeführten Beladungen übernimmt ein Objekt der Klasse IloExpr.

```
IloExpr performedPickups(myEnv);
```

Die Deklaration zum Zählen der durchgeführten Beladeknoten bedient sich des Pickup-Arrays und der Funktion IloVisit::performed, die als Constraint- und Bool'sche Funktion implementiert wurde. Da in C der Zustand true durch eine eins dargestellt wird, ist diese Herangehensweise erfolgreich.

```
for (k = 0; k < arPickups->getSize(); ++k)
{
  performedPickups += (*arPickups)[k].performed() == IloTrue; //+ 1 ?
}
```

Entsprechend den Fortschritten der Suche durch IloSolver::solve, enthält „performedPickups“ die Anzahl der verwendeten Beladeknoten. Die Beschränkung kann somit vollständig deklariert werden:

```
thisModel.add(performedPickups == iloDelivery.performed());//0==0 || 1 ==1
```

Die hier beschriebenen Constraints entsprechen nicht der wirklichen Reihenfolge, doch decken sie einen Großteil des interessanten Programmkodes ab. Weitere Constraints müssen entweder in gleicher Weise implementiert werden oder lohnen keiner detaillierten Betrachtung. Nur der Gesamtablauf mag von Interesse sein:

1. Alle Fahrzeuge in ein Array speichern und dabei alle schlechten aussortieren.
2. Jeden Auftrag in ein eigenes Modell bringen und lösen.
   a) Bei Erfolg wird dieses Modell in das Gesamtmodell eingefügt.
3. Das Gesamtmodell wird extrahiert.

Die Extraktion ist an dieser Stelle für das Constraint MlProduct2ProductConstraint notwendig, da die ID's der extrahierten Objekte benötigt werden. Nach der Extraktion werden alle (Ilo-)ID's von produktgleichen Be- und Entladeknoten an das MlProduct2ProductConstraint übergeben.

9.4 Lösung berechnen

Im vorangegangen Kapitel wurden alle Constraints erstellt und in myModel eingefügt. Dieses Modell kann mit Hilfe von MlGeneratePlan in MlManager::computeSolution gelöst und anschließend verbessert werden.

Für die Optimierung kann eine Zielfunktion deklariert werden, sodass beispielsweise

die Anzahl der erfüllten Aufträge oder die Gesamtkilometerzahl verringert wird. In verschiedenen Tests zeigte sich aber, dass eine Zielfunktion der Form

```
IloObjective myObjective =
          IloMinimize(myEnv,myDispatcher.getNbOfUnperformedVisits());

myModel.add(myObjective);
```

kaum einen Einfluss auf das Ergebnis hat. Außerdem wurde das Modell bereits extrahiert, was nach einem IloModel::add erneut durchgeführt werden müsste. Zusammen mit dem Prototypenstatus ist das Weglassen einer Zielfunktion gerechtfertigt. Somit werden nicht spezielle Kosten verringert, sondern die Gesamtkosten optimiert.

Die übliche Vorgehensweise, um mit den ILOG-Bibliotheken ein gutes Ergebnis zu erhalten, wurde bereits im Kapitel 7 besprochen. Demnach wird eine erste Lösung erzeugt und anschließend verbessert. Um eine Lösung zu verbessern, muss sie in einem „Lösungsobjekt“ gespeichert werden. Das Entsprechende Objekt wurde bereits als privates Attribut von MlManager deklariert und ist vom Typ IloRoutingSolution.

Die Constraintvariable, die die Gesamtkosten darstellt, muss durch ein Goal ein Label erhalten. In früheren Versionen war dies nicht notwendig und es zeigte sich während der Implementierung, dass gerade das Finden des korrekten Labels für die Kostenvariable die meiste Zeit in Anspruch nimmt. Die Funktion, die ein entsprechendes Goal erstellt, heißt IloDichotomize und implementiert binäre Suche.

```
IloGoal instantiateCost =
        IloDichotomize(myEnv, myDispatcher.getCostVar(), IloTrue, 300.0);
```

Unter bestimmten Bedingungen ist sogar nur wegen der Instantiierung der Kostenvariable eine Lösung unmöglich. In der Praxis äußert sich dieser Umstand durch einen stetig steigenden Speicherbedarf. Abhilfe schafft hier, die optionale Genauigkeit zu verringern, wodurch schneller eine Lösung gefunden werden kann und weniger Backtrack-Schritte im Speicher gehalten werden müssen. Die hier angestrebte Genauigkeit soll eine Auflösung von 5 Minuten, also 300 Sekunden bieten. Eine Lösung ist somit nur dann besser, wenn sie mindesten 5 Minuten einspart.

```
IloGoal instantiateCost =
        IloDichotomize(myEnv, myDispatcher.getCostVar(), IloTrue, 300.0);
```

Für die binäre Suche von IloDichotomize, kann die jeweils obere Hälfte als erstes Testgebiet (IloTrue) festlegt werden. Gerade wenn große Zahlen für die Kostenvariable zu erwarten sind, macht es sich bezahlt.

Die Goals zur Instantiierung der Kostenvariable und zur Lösungsfindung können als Konjunktion deklariert und berechnet werden.

```
IloGoal goal = MloGeneratePlan(myEnv, wDeliveries) && instantiateCost;
if (!mySolver.solve(goal))
  return false;
```

Sollte eine erste Lösung möglich sein, muss sie gesichert werden, denn nur ein IloRoutingSoulution-Objekt kann zur Verbesserung einer Lösung verwendet werden.

```
mySolution.store(mySolver);
```

Die Grundlage zur Verbesserung einer Lösung schaffen mehrere Heuristiken, die sogenannten Nachbarschaften IloNeighborhood. Dabei ist es durchaus möglich, mehrere IloNeighborhood-Objekte in einem entsprechenden Array zu kombinieren.

```
IloNHoodArray arNeighborhood(myEnv, 5);
arNeighborhood[0] = IloTwoOpt(myEnv);
arNeighborhood[1] = IloOrOpt(myEnv);
arNeighborhood[2] = IloRelocate(myEnv);
arNeighborhood[3] = IloExchange(myEnv);
arNeighborhood[4] = IloCross(myEnv);
IloNHood neighborhood = IloConcatenate(myEnv, arNeighborhood);
```

Die Heuristik „TwoOpt“ entfernt zwei Verbindungen aus der Route und versucht sie gewinnerhöhend neu zu verknüpfen. Die zweite Heuristik „OrOpt“ arbeitet innerhalb einer Route. Sie betrachtet zu einem Zeitpunkt immer nur ein Fahrzeug. Auf dem gewählten Fahrzeug werden dabei Teile der Route an anderer Stelle positioniert. Kann diese Änderung als Verbesserung betrachtet werden, wird sie beibehalten.
Die Heuristik „Relocate“ versucht eine Visit auf einem anderen Fahrzeug unterzubringen. Unter bestimmten Bedingungen kann dies sehr wirkungsvoll die Route verkürzen. Bei einen „Pickup-and-Delivery“ Problem müssen aber immer zwei Visits betrachtet werden (Be- und Entladeknoten). Glücklicherweise wird dieser Umstand automatisch von der Heuristik berücksichtigt. Ähnlich arbeitet die Heuristik „Exchange“, wobei in diesem Fall Visits mit Visits anderer Fahrzeuge *ausgetauscht* werden. Auch hier werden Be- und Entladeknoten gemeinsam betrachtet.
Die Heuristik „Cross“ trennt zwei sich kreuzende Routen zweier Fahrzeuge auf und verbindet diese neu. Die Enden der Routen zweier Fahrzeuge wird somit ausge-

tauscht und wahrscheinlich verbessert.

Das Goal zur Verbesserung der ersten Lösung kann mit dem so definierten IloN-Hood-Objekt erzeugt werden.

```
IloGoal improve = IloSingleMove(myEnv,
                   mySolution,        //gespeicherte Lösung
                   neighborhood,      //Heuristiken
                   IloImprove(myEnv, 1000), //Gready-Search
                   IloFirstSolution(myEnv),
                   instantiateCost);  //Subgoal
```

Die Funktion „IloSingleMove“ implementiert eine lokale Suche. IloImprove stellt die Metaheuristik dar. In diesem Fall wird nur nur eine Greedy-Heuristik (Kapitel 2.4.4) verwendet, die aber gerade bei der Verbesserung der ersten Lösung sehr wirkungsvoll sein kann. Das Subgoal zur Initialisierung der Kostenvariable darf nicht fehlen, da anhand der Kostenvariable der Fortschritt gemessen wird.

Das Goal verbessert nur einmal die Lösung und gibt dann die Kontrolle zurück. Die Zeit der gesamten Verbesserung sollte in jedem Fall beschränkt werden. Eine Möglichkeit besteht darin, die Zeit für die Gesamtsuche zu beschränken. In diesem Fall wird nach jedem Verbesserungsschritt die abgelaufene Zeit verglichen und nach einer Minute abgebrochen.

```
while (mySolver.solve(improve) && (myTimer.getTime() < 60.0))
{}
```

Anschließend muss die beste Lösung im Solver wiederhergestellt werden.

```
mySolver.solve(IloRestoreSolution(myEnv, mySolution));
```

Die so gefundene Lösung stellt das endgültige Ergebnis dar und kann in eine Ergebnisstruktur geschrieben werden. Diese wird von dem Delphiprogramm weiterverarbeitet.

9.5 Erläuterungen zur Delphi-Implementierung

Die folgende Beispielimplementierung soll die Benutzbarkeit der entwickelten Dynamic Link Library nachweisen. Hierzu müssen sich DLL und Programm auf eine Aufrufkonvention für DLL-Funktionen festlegen. Ohne eine bestimmte zu bevorzugen, wurde der Einfachheit halber die C-Konvention gewählt, da sie in Delphi keinen Mehraufwand bedeutet. Dazu muss die Funktion auf C++-Seite einen C-Namen erhalten, was dadurch erreicht wird, dass sie als `extern „C“` deklariert wird:

C-Name	C++-Name
`_loadProblem`	`?loadProblem@@YA_NPAD@Z`

Die DLL-Export-Deklaration ist von Compiler zu Compiler unterschiedlich. In diesem Fall wurde MS Visual Studio C++ 6.0 verwendet, wodurch eine DLL-Funktion folgendermaßen deklariert werden kann:

```
extern "C" __declspec(dllexport) bool loadProblem(char *theDirectory)
```

Ein Delphiprogramm muss die gleichen vereinbarten Konventionen einhalten, um die Parameter zu übergeben. Hier muss die Import-Funktion des Delphiprogrammes den C-Aufrufkonventionen folgen:

```
function loadProblem(thisDirectory: string): boolean; cdecl;
```

Der Aufruf im Programm geschieht dann völlig transparent:

```
if not loadProblem(edDirectory.Text) then
begin
  Application.MessageBox('Konnte Problem nicht laden!', 'Fehler', MB_OK);
end
```

Die vollständige Beispielimplementation (mlSolverTest) findet sich auf der beiliegenden CD-ROM.

Wird der Startbutton des Beispielprogramms betätigt, werden die Eingangsdaten in Form der generierten XML-Files aus dem angegebenen Verzeichnis verarbeitet und berechnet. Nach der Berechnung des gegebenen Routingproblems werden im Ausgabefenster Eckdaten der Lösung, wie z.B. die Anzahl der benutzten Fahrzeuge, verplante Visits und die Anzahl der erfüllten Aufträge angezeigt. Der Ausbau dieser Schnittstelle wäre ein Ziel der nächsten Version.

Um eine Fülle von Eingangsdaten zur Verfügung zu haben, wurde ein Datengenerator (DataGen auf der CD-ROM) implementiert. Dieser erzeugt zufallsgesteuert eine Probleminstanz des „Pickup and Delivery“ Problems. Da das Datenformat in XML und somit in Textform vorliegt, ist es sehr einfach, die Ergebnisse zu speichern.

Ein konkreter Vergleich der Berechnungen kann kaum durchgeführt werden, da die Daten zufällig generiert sind und man nicht immer von konsistenten Werten ausgehen kann. Beispielsweise ist es möglich, dass alle Beladeknoten eines Auftrags durch Zufall deaktiviert sind. Die Berechnung von 20 Aufträgen und über 500 Visits (weil jeder Auftrag fast 30 Beladeknoten zur Auswahl hat) benötigt auf einem durch-

schnittlichen PC (mit AMD K6-2 500MHz und 384 MB RAM) 47 Sekunden und kann 18 Aufträge der zufallsgenerierten Probleminstanz erfüllen. Eine andere mit 50 Aufträgen besitzt etwas mehr als 80 Visits und benötigt auf dem gleichen PC 26 Sekunden, kann aber nur 19 Aufträge verplanen, also nicht einmal 50%. Für einen besseren Vergleich müssen also Eingangsdaten eingelesen werden, die möglichst aus einem bestehenden System stammen, da über deren Lösbarkeit konkretere Aussagen getroffen werden können. Dazu sollte man in diesem Fall einen Konverter implementieren, der diese realen Daten in die benötigte Form überführt.

10 Zusammenfassung und Fazit

Man könnte das gegebene Problem noch beliebig erweitern und komplexer gestalten. So wäre es durchaus möglich, für jedes Fahrzeug einen festen Reinigungsplatz zu verlangen, denn die Säuberung kann unter Umständen nur mit bestimmten Auflagen für Umwelt und Natur stattfinden.
Die Dauer einer Fahrt könnte mit einer Maximalzeit beschränkt werden, wenn das Fahrzeug beladen ist. Denn bestimmte Ladungen wie Beton oder Lebensmittel dürfen nur über eine gewisse Zeit befördert werden. Das dazu notwendige Constraint müsste die Zeit zwischen dem Beladeknoten und dem zugehörigen Entladeknoten beschränken.
In der Praxis kann nicht davon ausgegangen werden, dass ein Fahrzeug immer einen bestimmten Anhänger mitführt, stattdessen können Fahrzeuge und Anhänger fast beliebig kombiniert werden. Aus diesem Grund kann auch die Ladekapazität schwanken. In bestimmten Fällen wird ein großer Anhänger vonnöten sein, in anderen ein kleinerer. Die verschiedenen Kombinationen aus Zugmaschinen und Anhängern können eine große Herausforderung für ein Planungssystem bilden.

Noch viele andere Erweiterungen und Verfeinerungen wären möglich, würden den Rahmen dieser Arbeit jedoch sprengen. Im Mittelpunkt sollte der Einsatz von modernen Constrainttechniken für Routingprobleme stehen.
Wie aufgezeigt ergeben sich einige Vorteile, die sich aus den Eigenschaften von Constraints ableiten lassen. So trägt ihr deklarativer Charakter dazu bei, dass sich Randbedingungen eines Problems schnell und einfach implementieren lassen, indem passende Constraints deklariert werden. Die Tatsache, dass sie Teilinformationen repräsentieren, erlaubt ein stückweises Vereinfachen der Problemstellung,

indem Constraints gelöscht werden können, ohne den Algorithmus zu ändern. Auch kann ein System dadurch leichter erweitert werden. Wird eine neue Randbedingung benötigt, muss nur ein weiteres Constraint hinzugefügt werden. Die Deklaration von Constraints ist eine bequeme Art, die Problemgröße einzugrenzen und den Algorithmus schlank zu halten.

Die Auswahl eines zweckmäßigen Constraintsolvers machte deutlich, dass der Markt im professionellen Bereich überschaubar ist. Die meisten Solver besaßen dabei leider nur eine geringe Ausrichtung auf das Fahrzeugrouten, wodurch die Problemrepräsentation eines „Pickup and Delivery“ Problems äußerst umfangreich gewesen wäre. Hier zeigte sich auch der deutliche Vorsprung von ILOG gegenüber den anderen Constraintsystemen. Zudem bot diese Klassenbibliothek die besten Voraussetzungen für eine Systementwicklung, nicht zuletzt aufgrund der C++-Schnittstelle.

Im Ergebnis wurde deshalb eine DLL auf Basis der ILOG-Bibliotheken entwickelt, auf die von einem Delphiprogramm zugegriffen wird. Sie ist in der Lage ein komplexes „Pickup and Delivery“ Problem zu lösen und dabei eine einfache Schnittstelle zu bieten. Die entwickelte Dynamic Link Library liest dazu die Probleminstanz ein, erzeugt die Beschränkungen mittels Constraints der ILOG-Bibliothek und berechnet eine Lösung.

Die große Mächtigkeit der Problemrepräsentation von ILOG Dispatcher stützte die Entscheidung, sich auf diese Klassenbibliothek festzulegen. Das Design eines Programmes ohne ILOG-Bibliotheken wäre ungleich schwieriger und die Implementierung wesentlich zeitintensiver und damit teurer gewesen, denn Programmiersprachen wie Object Pascal oder C++ bieten keine Möglichkeiten, Constraints zu deklarieren. Hier müssen Beschränkungen definiert und somit implementiert werden. In vielen Fällen wird man eine enge Verknüpfung zwischen Algorithmus und den verschiedenen Randbedingungen nicht vermeiden können. Dadurch werden die Programme komplexer und schwieriger zu warten oder zu erweitern. Eine Änderung an der einen Stelle könnte an anderer Stelle einen Fehler provozieren.

Jedoch dürfen die ILOG-Bibliotheken nicht überschätzt werden. Neben den immensen Kosten der Anschaffung und einigen Bugs im Handbuch (besonders die Erläuterungen zum Macro ILOCPCONSTRAINTWRAPPERDECL sind widersprüchlich) ist die Behandlung von Zahlenbereichsüberläufen problematisch. Das IloDimension2-Objekt für den Zeitverbrauch, enthält keinen Wert, wenn ein interner

Überlauf stattfindet. Eine Fehlerdiagnose ist nur anhand von Vermutungen möglich, eine Ausnahme (exception) wird nicht ausgelöst. Somit bleibt der Programmierer immer im Dunkeln, wenn eine Variable einen Zahlenbereich überschreitet. Aus diesem Grund wird in der Implementierung eine Normalisierung der Zeitwerte durchgeführt. Dazu werden alle Zeitwerte, wie '15.01.2002' mit einer anderen Zeit (hier 01.01.2000) subtrahiert. Das Ergebnis ist dann etwas wie '15.01.1972'. Die Darstellung des utc-Formats (Seite 43; utc) wird somit auf eine genügend kleine Zahl geführt.

Die Programmierer müssen bereit sein, ein Stück Verantwortung abzugeben und sich auf die Constraint-Bibliothek zu verlassen. Leider ist dies nicht immer möglich. Denn wie die Implementierung gezeigt hat, gibt es mit der ILOG-Constraint-Bibliothek streckenweise größere Probleme. Das Handbuch [SolvManual51] gibt wohl deshalb die Empfehlung, ein Programm, das ein Problem implementiert, sukzessiv zu formulieren und nach jedem Schritt die Konsistenz zwischen theoretischem und realem Modell zu prüfen. Ergänzend kann man hinzufügen, dass nach jedem Schritt die einwandfreie Funktion des Programms kontrolliert werden sollte. Hin und wieder kam es vor, dass beim Setzen bestimmter Constraints keine Lösung mehr möglich war. Syntaktisch und semantisch waren diese korrekt, mögen aber Interferenzen mit anderen Constraints hervorgerufen und die Lösung damit unbrauchbar gemacht haben. Es bleibt also nur die Möglichkeit, ein Programm im Wechsel mit Implementierung und Test zu erstellen. Die erhoffte Geschwindigkeitszunahme der Programmierung ist dadurch eher enttäuschend.

Ein weiterer negativer Punkt der ILOG-Bibliotheken ist die Instantiierung der Kostenvariable. Das Label der Variable ergibt sich aus der Routinglösung, ein einfaches Addieren der verschiedenen Größen sollte eigentlich genügen, stattdessen wird die Domain der Variable binär durchsucht. Um zu zeigen, dass eine effektivere Lösung möglich ist, als im Handbuch beschrieben, wurde testweise ein Goal MlBindCost implementiert. Dieses Goal addiert alle Kosten und bindet diese anschließend an die Kostenvariable. Es verursacht dadurch kein Backtracking und findet dementsprechend schneller zu einer Lösung.

Grundsätzlich entlastet die ILOG-Constraint-Bibliothek den Programmierer bei den meisten Problemen, die sich für ein Programm zur kombinatorischen Problemlösung ergeben. Doch muss auch beachtet werden, dass einige neue Probleme, wie oben beschrieben, entstehen können. So bleibt zu hoffen, dass die Bibliotheken weiter-

entwickelt werden und Probleme beseitigt werden. Der Vergleich der ILOG-Dispatcher-Bibliothek 2.1 und der Version 3.1 zeigt in jedem Fall einen deutlichen Fortschritt hinsichtlich der Performanz und den Debuggingmöglichkeiten.
Das Backtracking ist jetzt schon hervorragend integriert und wird von jeder Klasse implizit genutzt. So ist der Goalstack, die Status der Constraints und deren Variablen vom Backtracking des Systems abhängig.

Die Arbeit hat gezeigt, das Constraints ein sehr guter Ansatz zur Lösung von „Pickup and Delivery“ Problemen sind. Allerdings spielt die Zeitkomponente immer noch eine wichtige Rolle bei der Berechnung. In der Zukunft könnten DNA-Computer solche NP vollständigen Probleme in einer Zeit linear zu ihrer Problemgröße lösen.

Anhang A - Eingangsdaten

Für das Verständnis des vorliegenden Anhangs werden grundlegende Kenntnisse von XML erwartet. Die abgebildeten Schema-Dateien geben die Struktur der verschiedenen Eingangsdaten wieder. Das so festgelegte Format wird in den zugehörigen XML-Dateien benutzt, welche dann von der Bibliothek eingelesen werden.

TimeWindow

Der Typ *timeWindow* kann auf die vordefinierten Datums- und Zeitformate zurückgreifen.

```
<xsd:complexType name="timeWindow">
  <xsd:sequence>
    <xsd:element name="from" type="xsd:dateTime"/>
    <xsd:element name="to" type="xsd:dateTime"/>
  </xsd:sequence>
</xsd:complexType>
```

Diese Definition verlangt, dass die zwei Subelemente von *timeWindow from* und *to* vom Typ *dateTime* sind und sie in der angegebenen Reihenfolge erscheinen.

Nodes

Ein Längen- und Breitengrad darf genau einmal je Datensatz angegeben werden. Die ID des Knotens muss vorhanden sein.

```
<xsd:element name="node">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="longitude" type="xsd:integer"/>
      <xsd:element name="latitude" type="xsd:integer"/>
    </xsd:sequence>
    <xsd:attribute name="uID" type="xsd:positiveInteger" use="required"/>
  </xsd:complexType>
</xsd:element>
```

Zusätzlich muss die Mehrzahl eines Knotens beschrieben werden:

```
<xsd:element name="nodes">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="node" minOccurs="1" maxOccurs="unbounded"/>
```

```
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Breaks

Die Identifikationsnummer *nodeID* eines geographischen Knotens wird von einer positiven Zahl repräsentiert. Sie soll größer als Null sein, was durch *positiveInteger* erreicht wird. Da eine Pause an einem bestimmten Ort stattfinden kann aber nicht muss, soll das Tag der Knotenidentifikationsnummer *nodeID* auch weglassen werden können, weshalb auch die *minOccurs* und *maxOccurs* entsprechend gesetzt sind.

Die Fahrzeugidentifikationsnummer *vehicleID* muss genau einmal vorhanden sein. Das kann entweder dadurch erreicht werden, dass *minOccurs* und *maxOccurs* auf eins gesetzt wird oder beide einfach weggelassen werden. Der Anfang der Pause *beginning* soll in einem Zeitinterval liegen und ist deshalb vom Typ *timeWindow,* die Dauer *duration* dagegen von einem entsprechend vordefinierten Typus. Ebenso die maximale Verzögerung einer Pause *timeDelay*.

Die entsprechenden Größen für die geladenen Liter und Gramm (*loadedLiters* und *loadedGramms*) müssen auch die Zahl 0 annehmen können. Deshalb sind sie vom Typ *nonNegativeInteger*. Die Identifikationsnummer der Pause *uID* ist eine Zahl größer 0, wodurch sich folgende Definition ergibt:

```
<xsd:element name="break">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="nodeID" type="xsd:positiveInteger" minOccurs="0"
                                                            maxOccurs="1"/>
      <xsd:element name="vehicleID" type="xsd:positiveInteger"/>
      <xsd:sequence>
        <xsd:element name="beginning" type="timeWindow"/>
      </xsd:sequence>
      <xsd:element name="duration" type="xsd:duration"/>
      <xsd:element name="timeDelay" type="xsd:duration"/>
      <xsd:element name="loadedLiters" type="xsd:nonNegativeInteger"/>
      <xsd:element name="loadedGramms" type="xsd:nonNegativeInteger"/>
    </xsd:sequence>
```

```
    <xsd:attribute name="uID" type="xsd:positiveInteger" use="required"/>
  </xsd:complexType>
</xsd:element>
```

Die Mehrzahl wird wie im Abschnitt zuvor erreicht:

```
<xsd:element name="breaks">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="break" minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Beladestation

Beladestationen oder Beladeknoten haben die gleichen Eigenschaften wie Entladeknoten, doch soll zur besseren Trennung jedes XML-Schema-File für sich definiert werden. Eine Knotenidentifikationsnummer wird genau einmal benötigt. Auch bedarf es der Aussage, ob die Station geöffnet ist (*isActive*) nur einmal. Die Öffnungszeiten *openHours* sind vom Typ *timeWindow*. Die Maximalgrößen des Fahrzeugs und dessen maximale Lautstärke werden einmal benötigt. Ebenso die Zeitverzögerung *timeDelay* des Aufnahmepunktes.

```
<xsd:element name="pickup">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="nodeID" type="xsd:positiveInteger"/>
      <xsd:element name="isActive" type="xsd:boolean"/>
      <xsd:sequence>
        <xsd:element name="openHours"
                    type="timeWindow" minOccurs="1" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:element name="length" type="xsd:nonNegativeInteger"/>
      <xsd:element name="width" type="xsd:nonNegativeInteger"/>
      <xsd:element name="height" type="xsd:nonNegativeInteger"/>
      <xsd:element name="mass" type="xsd:nonNegativeInteger"/>
      <xsd:element name="soundLevel" type="xsd:nonNegativeInteger"/>
      <xsd:element name="timeDelay" type="xsd:duration"/>
```

```
    </xsd:sequence>
    <xsd:attribute name="uID" type="xsd:positiveInteger" use="required"/>
  </xsd:complexType>
</xsd:element>
```

Entsprechend den vorherigen Abschnitten (Anhang A - Eingangsdaten) kann auch hier die Mehrzahl eines *pickup* - Elementes beschrieben werden.

Entladestation

Das Analogon zu den Beladestationen bilden die Entladestationen. Und wie in Kapitel 8.1 bereits festgestellt, besteht eine enge formale Verwandtschaft zwischen diesen beiden. Auch in den Eingangsdaten spiegelt sich das wieder, wodurch die Änderung nur im Elementnamen *pickup* zu *delivery* besteht.

Fahrzeuge

Der Typ der *classID* und der *typeID* sind ganzzahlige Integerwerte. Der Heimatstandort *nodeHome* und der aktuelle Standort *nodeTopical* sind Referenzen auf die Kennungen geographischer Knoten (Kapitel 8.2.2 und Anhang A - Eingangsdaten - Nodes). Der Arbeitstag wird durch *workTimeWindow* eingegrenzt und die maximale Arbeitszeit je Tag wird durch *workingHours* angegeben. Die Geschwindigkeit *speed* wird in km/h aufgefasst. Die Ladung für volumenabhängige Transportgüter *loadVolume* wird in Litern angegeben, die für massenabhängige Güter in Gramm. Die allgemeine Zeitverzögerung *timeDelay*, die durch dieses Fahrzeug verursacht wird und an jeder Position einzurechnen ist, wird als *duration* angegeben. Die Verzögerungen für die Inbetriebnahme (*timeDelayStart*) und das endgültige Abstellen des Fahrzeugs (*timeDelayEnd*) werden ebenfalls als *duration* übergeben. Ebenso die Verzögerungen, die bei einer Beladung (*loadLag*) oder bei einer Entladung (*unloadLag*) auftreten. Auch die Zeitverzögerungen je Maßeinheit werden als *duration* ausgewertet.

```
<xsd:element name="vehicle">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="classID" type="xsd:positiveInteger"/>
      <xsd:element name="typeID" type="xsd:positiveInteger"/>
      <xsd:element name="nodeHome" type="xsd:nonNegativeInteger"/>
      <xsd:element name="nodeTopical" type="xsd:nonNegativeInteger"/>
      <xsd:element name="workTimeWindow" type="timeWindow"/>
```

```
      <xsd:element name="workingHours" type="xsd:duration"/>
      <xsd:element name="length" type="xsd:nonNegativeInteger"/>
      <xsd:element name="width" type="xsd:nonNegativeInteger"/>
      <xsd:element name="height" type="xsd:nonNegativeInteger"/>
      <xsd:element name="mass" type="xsd:nonNegativeInteger"/>
      <xsd:element name="soundLevel" type="xsd:nonNegativeInteger"/>
      <xsd:element name="speed" type="xsd:positiveInteger"/>
      <xsd:element name="loadVolume" type="xsd:nonNegativeInteger"/>
      <xsd:element name="loadMass" type="xsd:nonNegativeInteger"/>
      <xsd:element name="timeDelay" type="xsd:duration"/>
      <xsd:element name="timeDelayStart" type="xsd:duration"/>
      <xsd:element name="timeDelayEnd" type="xsd:duration"/>
      <xsd:element name="loadLag" type="xsd:duration"/>
      <xsd:element name="unloadLag" type="xsd:duration"/>
      <xsd:element name="cleaningTime" type="xsd:duration"/>
      <xsd:element name="loadLagPerCubicmetre" type="xsd:duration"/>
      <xsd:element name="unloadLagPerCubicmetre" type="xsd:duration"/>
      <xsd:element name="loadLagPerTon" type="xsd:duration"/>
      <xsd:element name="unloadLagPerTon" type="xsd:duration"/>
      <xsd:element name="cost" type="xsd:nonNegativeInteger"/>
      <xsd:element name="costPerHourUnloaded" type="xsd:float"/>
      <xsd:element name="costPerHourLoaded" type="xsd:float"/>
      <xsd:element name="costPerKmUnloaded" type="xsd:float"/>
      <xsd:element name="costPerKmLoaded" type="xsd:float"/>
    </xsd:sequence>
    <xsd:attribute name="uID" type="xsd:positiveInteger" use="required"/>
  </xsd:complexType>
</xsd:element>
```

Die notwendige Mehrzahl wird wie oben beschrieben erreicht.

Aufträge

Die Transportgutnummer *productID* und die Klassen-ID des Fahrzeugs *vehicle-ClassID* sind obligatorisch. Die Typnummer des Fahrzeugs *vehicleTypeID* kann dagegen auch weggelassen, bzw. mehrfach genannt werden. Das Gleiche gilt für die Fahrzeug-ID (*vehicleID*). Die Beladestationen *pickupID* müssen mindestens einmal

vorhanden sein, und die Entladestationen genau einmal. Die Lieferzeit *deliveryTime* ist vom Typ *timeWindow* und die Volumen- und Massenmengen werden nur einmal erwartet. Die maximal zulässige Verzögerung *maximumTardiness* ist vom Typ *duration* und die Werte, ob der Auftrag bereits auf ein Fahrzeug geladen wurde (*preloading*) und ob dieses Material eine Reinigung des Fahrzeugs verlangt, sind vom Typ Bool.

```
<xsd:element name=“order“>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:positiveInteger"/>
      <xsd:element name="vehicleClassID" type="xsd:positiveInteger"/>
      <xsd:element name="vehicleTypeID" type="xsd:positiveInteger"
                                  minOccurs="0" maxOccours="unbounded"/>
      <xsd:element name="vehicleID" type="xsd:positiveInteger"
                               minOccurs="0" maxOccours="unbounded"/>
      <xsd:element name="pickupNode" type="xsd:nonNegativeInteger"
                                minOccurs="1" maxOccours="unbounded"/>
      <xsd:element name="deliveryNode" type="xsd:nonNegativeInteger"/>
      <xsd:element name="supplyPeriod" type="timeWindow"/>
      <xsd:element name="quantityVolume" type="xsd:nonNegativeInteger"/>
      <xsd:element name="quantityMass" type="xsd:nonNegativeInteger"/>
      <xsd:element name="maximumTardiness" type="xsd:duration"/>
      <xsd:element name="preloaded" type="xsd:boolean"/>
      <xsd:element name="needCleaning" type="xsd:boolean"/>
    </xsd:sequence>
    <xsd:attribute name="uID" type="xsd:positiveInteger" use="required"/>
  </xsd:complexType>
</xsd:element>
```

Wieder kann die Mehrzahl eines Auftrages (*order*) entsprechend Anhang B festgelegt werden.

Anhang B – Klassendeklarationen

Die Klassendeklarationen in diesem Anhang geben den Kern wider. Sie stehen in engem Zusammenhang zu dem Klassendiagramm und können mit gewissen Freiheiten ineinander überführt werden.

MlTimeWindow

```
class _TimeWindow
{
         friend class MlTimeWindow;
private:
         unsigned int from;
         unsigned int to;
public:
         _TimeWindow(void);
         ~_TimeWindow() {};
};

typedef std::list<_TimeWindow*> LISTWINDOWS;

class MlTimeWindow
{
private:
         LISTWINDOWS                wndList;
         LISTWINDOWS::iterator      wndIter;
public:
         MlTimeWindow(void);
         ~MlTimeWindow();
         bool add(const unsigned int, const unsigned int);
         bool get(unsigned int &, unsigned int &);
         bool end(void);
         bool begin(void);
         void setToBegin(void);
         void setToEnd(void);
};
```

MlNodes

```
class MlNodes
{
private:
         LISTNODES                   nodeList;
         LISTNODES::iterator         nodeIter;
         IloEnv                      myEnv;
         IloNodeI*                   locate(std::string);
public:
         MlNodes(IloEnv);
         ~MlNodes(void);
         bool     add(const DOM_Node &);
         bool     del(DOM_Node);
         bool     del(std::string);
         IloNodeI* get(unsigned int);
};
```

MlBreaks

```
class MlBreak
{
private:
         std::string        name;
         unsigned int       nodeID;
         unsigned int       vehicleID;
         MlTimeWindow       beginning;
         unsigned int       timeDelay;
         unsigned int       duration;
         unsigned int       loadedLiters;
         unsigned int       loadedGramms;
public:
         MlBreak(void);
         ~MlBreak();
         std::string  getName(void);
         unsigned int getNodeID(void);
         unsigned int getVehicleID(void);
```

```
        bool     getBeginning(unsigned int &, unsigned int &);
        unsigned int getTimeDelay(void);
        unsigned int getDuration(void);
        unsigned int getLoadedLiters(void);
        unsigned int getLoadedGramms(void);
        void setName(const std::string);
        void setNodeID(const unsigned int);
        void setVehicleID(const unsigned int);
        bool setBeginning(const unsigned int, const unsigned int);
        void setTimeDelay(const unsigned int);
        void setDuration(const unsigned int);
        void setLoadedLiters(const unsigned int);
        void setLoadedGramms(const unsigned int);
};

typedef std::list<MlBreak*> LISTBREAKS;

class MlBreaks
{
private:
        LISTBREAKS                  breakList;
        LISTBREAKS::iterator        breakIter;
        IloSolver                   mySolver;
        MlNodes                     *myNodes;
        MlBreak*  locateNextOf(const unsigned int vehicleID);
public:
        MlBreaks(IloSolver, MlNodes *);
        ~MlBreaks(void);
        bool add(const DOM_Node &);
        bool insertBreaks(const IloDimension2 dimTime, IloModelI *);
        bool del(DOM_Node);
        bool del(std::string);
};
```

MlVisits

```
class MlVisit
{
private:
         MlTimeWindow       openHours;
         std::string        name;
         unsigned int       nodeID;
         double             length,
                            width,
                            height,
                            mass,
                            soundLevel;
         IloBool            active;
         unsigned int       timeDelay;
public:
         MlVisit(void);
         ~MlVisit();
         std::string  getName(void);
         unsigned int getNodeID(void);
         double getLength(void);
         double getWidth(void);
         double getHeight(void);
         double getMass(void);
         double getSoundLevel(void);
         int    getDelay(void);
         bool   getOpenTime(unsigned int &, unsigned int &);
         bool   openTimeEnd(void);
         bool   openTimeBegin(void);
         IloBool isActive(void);
         void setName(const std::string);
         void setNodeID(const unsigned int);
         void setLength(const double);
         void setWidth(const double);
         void setHeight(const double);
         void setMass(const double);
```

```
        void setSoundLevel(const double);
        void setDelay(const int);
        void addOpenTime(const unsigned int, const unsigned int);
        void setOpenTimeToBegin(void);
        void setActive(const IloBool thisBool = true);
};

typedef std::list<MlVisit*> LISTVISITS;

class MlVisits
{
private:
        LISTVISITS        visitList;
        LISTVISITS::iterator visitIter;
        MlVisit* locate(const unsigned int);
public:
        MlVisits(void);
        ~MlVisits();
        bool add(const DOM_Node &);
        MlVisit* get(const unsigned int);
};
```

MlVehicles

```
class MlVehicle;
typedef std::list<MlVehicle*>        ML_LISTVEHICLES;
typedef std::list<IloVehicleI*>      ILO_LISTVEHICLES;
typedef std::list<IloIntArray*>      LISTCLASSES;
typedef std::list<IloIntArray*>      LISTTYPES;

class MlVehicle
{
private:
        unsigned int      classID,
                          typeID,
                          nodeHome,
```

```
                            nodeTopical,
                            length,
                            width,
                            height,
                            mass,
                            soundLevel,
                            timeDelay,
                            timeDelayStart,
                            timeDelayEnd,
                            loadLag,
                            unloadLag,
                            cleaningTime;
        double              loadLagPerLitre,
                            unloadLagPerLitre,
                            loadLagPerGramm,
                            unloadLagPerGramm,
                            costPerHourUnloaded,
                            costPerHourLoaded,
                            costPerMeterUnloaded,
                            costPerMeterLoaded;
        std::string         name;
public:
        MlVehicle(std::string thisName);
        std::string  getName(void);
        unsigned int getClassID(void);
        unsigned int getTypeID(void);
        unsigned int getNodeHomeID(void);
        unsigned int getNodeTopicalID(void);
        unsigned int getLength(void);
        unsigned int getWidth(void);
        unsigned int getHeight(void);
        unsigned int getMass(void);
        unsigned int getSoundLevel(void);
        unsigned int getTimeDelay(void);
        unsigned int getTimeDelayStart(void);
```

```
        unsigned int getTimeDelayEnd(void);
        unsigned int getLoadLag(void);
        unsigned int getUnloadLag(void);
        unsigned int getCleaningTime(void);
        double   getLoadLagPerLitre(void);
        double   getUnloadLagPerLitre(void);
        double   getLoadLagPerGramm(void);
        double   getUnloadLagPerGramm(void);
        double   getCostPerHourUnloaded(void);
        double   getCostPerHourLoaded(void);
        double   getCostPerMeterUnloaded(void);
        double   getCostPerMeterLoaded(void);
        void setName(const unsigned int);
        void setName(const std::string);
        void setClassID(const unsigned int);
        void setTypeID(const unsigned int);
        void setNodeHomeID(const unsigned int);
        void setNodeTopicalID(const unsigned int);
        void setLength(const unsigned int);
        void setWidth(const unsigned int);
        void setHeight(const unsigned int);
        void setMass(const unsigned int);
        void setSoundLevel(const unsigned int);
        void setTimeDelay(const unsigned int);
        void setTimeDelayStart(const unsigned int);
        void setTimeDelayEnd(const unsigned int);
        void setLoadLag(const unsigned int);
        void setUnloadLag(const unsigned int);
        void setCleaningTime(const unsigned int);
        void setLoadLagPerLitre(const double);
        void setUnloadLagPerLitre(const double);
        void setLoadLagPerGramm(const double);
        void setUnloadLagPerGramm(const double);
        void setCostPerHourUnloaded(const double);
        void setCostPerHourLoaded(const double);
```

```
        void setCostPerMeterUnloaded(const double);
        void setCostPerMeterLoaded(const double);
};

class MlVehicles
{
private:
        IloEnv                      myEnv;
        MlNodes                     *myNodes;
        ILO_LISTVEHICLES            vehicleList;
        ILO_LISTVEHICLES::iterator  vehicleIter;
        ML_LISTVEHICLES             mlVehicleList;
        ML_LISTVEHICLES::iterator   mlVehicleIter;
        LISTCLASSES                 classList;
        LISTCLASSES::iterator       classIter;
        LISTTYPES                   typeList;
        LISTTYPES::iterator         typeIter;

        IloIntArray* getClass(unsigned int);
        IloIntArray* getType(unsigned int);
        bool isVehicleInClass(const unsigned int thisVehicle,
                              const unsigned int thisClass);
        bool isVehicleInType(const unsigned int thisVehicle,
                             const unsigned int thisType);
        MlVehicle*  locate(const std::string &);
public:
        MlVehicles(IloEnv, MlNodes*);
        ~MlVehicles();
        bool del(DOM_Node);
        bool add(const DOM_Node &);
        void setVehiclesToBegin(void);
        void setVehiclesToEnd(void);
        void addVehicleToClass(const std::string, const unsigned int);
        void addVehicleToType(const std::string, const unsigned int);
        MlVehicle*   get(const std::string);
        MlVehicle*   get(const unsigned int);
```

```
        IloVehicleI* getIloVehicleAt(const unsigned int);
        IloVehicleI* getIloVehicle(const unsigned int);
        bool vehiclesEnd(void);
        bool vehiclesBegin(void);
        unsigned int getClassIdOf(const unsigned int &);
        unsigned int getTypeIdOf(const unsigned int &);
        unsigned int getClassIdOf(const std::string);
        unsigned int getTypeIdOf(const std::string);
        unsigned int getNumber(void);
};
```

MlOrders

```
typedef std::list<unsigned int*>     _LISTTYPES;
typedef std::list<unsigned int*>     _LISTPICKUPS;
typedef std::list<unsigned int*>     _LISTVEHICLES;

class MlOrder
{
private:
        std::string                  name;
        unsigned int                 productID;
        unsigned int                 vhClassID;
        _LISTTYPES                   typeList;
        _LISTTYPES::iterator         typeIter;
        _LISTVEHICLES                vehicleList;
        _LISTVEHICLES::iterator      vehicleIter;
        _LISTPICKUPS                 pickupList;
        _LISTPICKUPS::iterator       pickupIter;
        unsigned int                 delivery;
        MlTimeWindow                 supplyPeriod;
        unsigned int                 massQuantity;
        unsigned int                 volumeQuantity;
        unsigned int                 maximumTardiness;
        IloBool                      preloaded;
        IloBool                      _needCleaning;
```

```
public:
        MlOrder(void);
        ~MlOrder();
        std::string  getName(void);
        unsigned int getProductID(void);
        unsigned int getVhClassID(void);
        unsigned int getVhTypeID(void);
        unsigned int getVehicleID(void);
        unsigned int getPickupID(void);
        unsigned int getDeliveryID(void);
        bool getSupplyPeriod(unsigned int &, unsigned int &);
        bool supplyPeriodBegin(void);
        bool supplyPeriodEnd(void);
        unsigned int getQuantityMass(void);
        unsigned int getQuantityVolume(void);
        unsigned int getMaximumTardiness(void);
        IloBool isPreloaded(void);
        IloBool needCleaning(void);
        bool pickupEnd(void);
        bool pickupBegin(void);
        bool typesBegin(void);
        bool typesEnd(void);
        bool vehiclesBegin(void);
        bool vehiclesEnd(void);
        void setPickupToBegin(void);
        void setVehicleToBegin(void);
        void setTypeToBegin(void);
        void setName(const std::string);
        void setProductID(const unsigned int);
        void setVhClassID(const unsigned int);
        bool addVhTypeID(const unsigned int);
        bool addVehicleID(const unsigned int);
        bool addPickupID(const unsigned int);
        void setDeliveryID(const unsigned int);
        bool addSupplyPeriod(const unsigned int, const unsigned int);
```

```
        void setSupplyPeriodToBegin(void);
        void setSupplyPeriodToEnd(void);
        void setQuantityMass(const unsigned int);
        void setQuantityVolume(const unsigned int);
        void setMaximumTardiness(const unsigned int);
        void setPreloaded(const IloBool yes=true);
        void setCleaning(const IloBool yes=true);
};

typedef std::list<MlOrder*> LISTORDERS;

class MlOrders
{
private:
        LISTORDERS                  orderList;
        LISTORDERS::iterator        orderIter;
        MlOrders*                   locate(std::string);
public:
        MlOrders(void);
        ~MlOrders(void);
        bool add(const DOM_Node &);
        bool del(DOM_Node);
        bool del(std::string);
        MlOrder* get(unsigned int);
        MlOrder* getNext(void);
        unsigned int getNumber(void);
        void     setToBegin(void);
        bool     begin(void);
        bool     end(void);
};
```

MlManager

```
class MlManager
{
private:
```

```
        IloEnv              myEnv;
        IloDispatcher       myDispatcher;
        IloSolver           mySolver;
        IloRoutingSolution  mySolution;
        IloModel            myDimModel;
        IloModel            myModel;
        MlNodes             myNodes;
        MlVehicles          myVehicles;
        MlVisits            myPickups;
        MlVisits            myDeliveries;
        MlOrders            myOrders;
        MlBreaks            myBreaks;
        //Dimensions
        IloDimension1       ldWeight; //load weight
        IloDimension1       ldVolume; //load Volume
        IloDimension2       time;
        IloDimension2       length;
        IloDimension1       vhMileage;
        IloDimension1       vhCost;
        IloDimension1       tardiness;
public:
        MlManager(void);
        ~MlManager();
        bool insertNode(const DOM_Node &);
        bool insertVehicle(const DOM_Node &);
        bool insertPickup(const DOM_Node &);
        bool insertDelivery(const DOM_Node &);
        bool insertOrder(const DOM_Node &);
        bool insertBreak(const DOM_Node &);
        bool createAllObjects(void);
        bool computeSolution(void);
};
```

MlInterface

```
#ifdef _USRDLL
```

```
  #define _IDLL __declspec(dllexport)
#else
  #define _IDLL
#endif
extern "C"
{
bool _IDLL loadProblem(char *theDirectory);
bool _IDLL computeProblem(struct TResultOverview &theResult);
}
bool openManager(void);     // diese Funktionen erhalten C++ Namen
bool closeManager(void);    //sie werden intern von der DLL genutzt
#endif
```

MlGeneratePlan

```
// Modellklasse
class MloGeneratePlanI : public IloGoalI
{
private:
  IloInt delivCount;
public:
  MloGeneratePlanI(IloEnvI*, IloInt);
  IlcGoal   extract(const IloSolver) const;
  IloGoalI* makeClone(IloEnvI*) const;
  void      display(ILOSTD(ostream&)) const;
};
IloGoal MloGeneratePlan(const IloEnv thisEnv, IloInt delivCount);

// Algorithmenklasse
class MlcGeneratePlanI : public IlcGoalI
{
private:
  IlcIntSet        myDeliveries;
  IlcIntArray      myVehicles;
  IlcBool          createVisitSet(void);
  IlcBool          createVehicleSet(void);
```

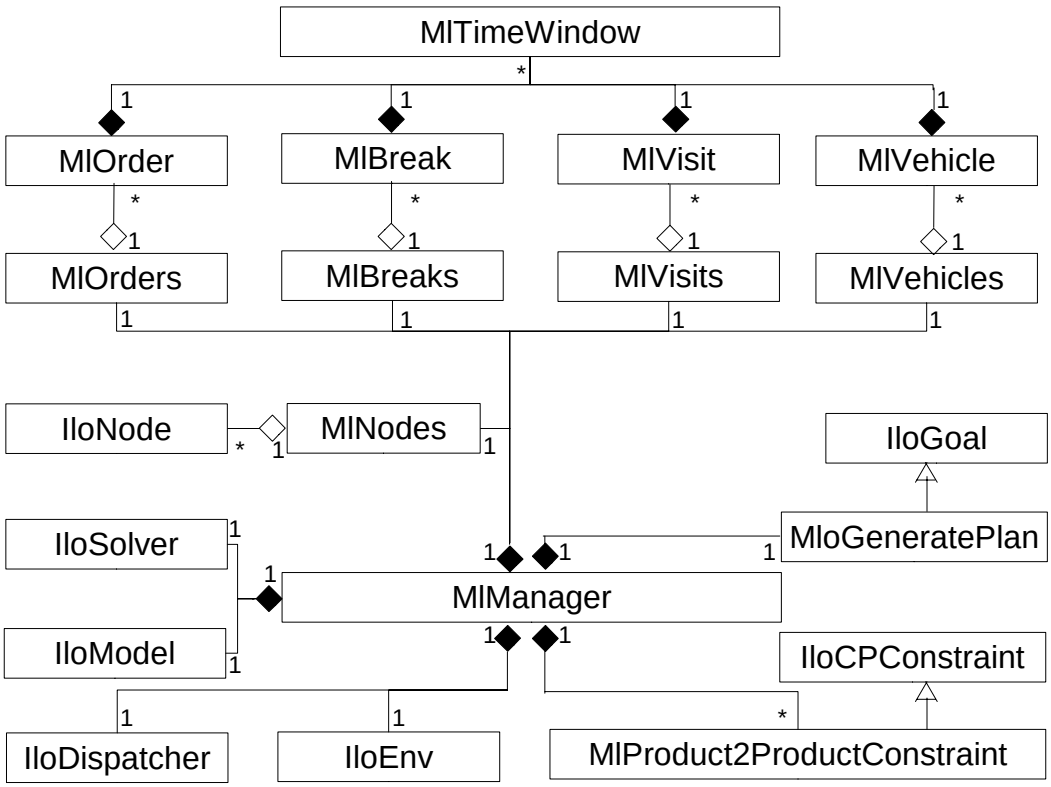
```
  IlcBool         createDeliveryListOn(IloVehicle, IlcIntArray &);
  IloVisit        getNearestDelivery(IloVisit, IloVehicle, IlcIntSet);
public:
  MlcGeneratePlanI(IloSolver, IlcInt);
  IlcGoal execute();
};
IlcGoal MlcGeneratePlan(const IloSolver thisSolver, IlcInt delivCount);
```

MlProduct2ProductConstraint

```
//Modellklasse
class MloProduct2ProductConstraintI : public IloCPConstraintI
{
  ILOCPCONSTRAINTWRAPPERDECL
private:
  IloIntArray     _successors;
  IloVisit        _visit;
  IloDimension1   _dim;
public:
  MloProduct2ProductConstraintI(IloEnvI*, const IloIntArray &,
                                IloVisit &, IloDimension1 &);
  IloExtractableI* makeClone(IloEnvI*) const;
  void display(ostream & out) const;
  IlcConstraint extract(const IloSolver) const;
};
IloConstraint MloProduct2ProductConstraint(IloEnv, IloIntArray,
                                           IloVisit &, IloDimension1 &);


//Algorithmenklasse
class MlcProduct2ProductConstraintI : public IlcConstraintI
{
private:
  IloSolver       mySolver;
  IloVisit        myVisit;
  IloDispatcher   myDispatcher;
  IlcIntVar       mySuccessors;
  IloDimension1   myDim;
```

```
public:

  MlcProduct2ProductConstraintI(IloSolver, IlcIntVar,
                                          IloVisit &, IloDimension1 &);

  void post();

  void propagate();

  IlcBool isViolated() const;

};

IlcConstraint MlcProduct2ProductConstraint(IloSolver, IlcIntVar,
                                              IloVisit, IloDimension1);
```

Anhang C - UML-Klassendiagramm mit Kardinalitäten



Literaturverzeichnis

[Ascheuer u.a. 1999] Ascheuer, Fischetti, Grötschel (1999) : *Solving the Asymmetric TSP with Time Windows by Branch and Cut*.

[Baget & Tognetti 2001] Baget, Tognetti (2001) : *Backtracking Through Biconnected* Components of a Constraint Graph.

[Barták 1998] Roman Barták, *On-Line Guide to Constraint Program ming* (http://kti.ms.mff.cuni.cz/~bartak/constraints/), 1998

[Barták 1999] Roman Barták (1999) : *Constraint programming: In Pursuit of the Holy Grail*.

[Baumgärtel 1997] Hartwig Baumgärtel : *Dezentrale Constraintverarbeitung für die Produktionslogistik*. Aus: KI, 1/1997

[Chew u.a. 2000] Tee Young Chew, Martin Henz, Ng Ka Boon, *A Toolkit for Constraint-based Inference Engines*, 2000

[CHIPsys1997] o.V. **(**1997) : *The CHIP System*

[c't 1996] http://www.heise.de/ct/96/05/087/default.shtml

[c't 2001] c't 7/2001

[DispatcherMan21] o.V. (1999) : *ILOG Dispatcher 2.1 Users Manual.*

[FH-Flensburg] http://www.iti.fh-flensburg.de/lang/algorithmen/grund lagen/graph.htm

[Freuder & Wallace 1992] Freuder, Wallace (1992) : *Partial Constraint Satisfaction*.

[Frühwirth & Abdennadher 1997] Frühwirth, Abdennadher (1997) : *Constraint-Program mierung*

[Garey & Johnson 1979] Garey, M. R., D. S. Johnson, *A Guide to the Theory of NP-Completeness,* 1979

[Guesgen 1997] Hans Werner Guesgen, *KI:Partielle Constraint-Erfüllung*, 1997

[Harvey & Ginsberg 1995] William D. Harvey, Matthew L. Ginsberg (1995) : *Limited Discrepancy Search*.

[Heinsohn & Socher 1999] Heinsohn, Socher-Ambrosius (1999) : *Wissensverarbei tung Eine Einführung.*

[Hillier & Lieberman 1997] Hillier, Lieberman (1997) : *Operations Research Einfüh rung. 5. Auflage*

[IVU Web] http://www.ivu.de/cms/cms_seite_suchen.pl?seiten_id=2_ 4de

[Kirkpatrick u.a. 1983] S. Kirkpatrick, C.D. Gelatt, Jr. & M.P. Vecchi : *Optimiza tion by Simulated Annealing*. Aus: Science, 220/1983

[Mackworth & Freuder 1993] Alan K. Mackworth, Eugene C. Freuder (1993) : *The f Constraint Satisfaction Revisited*.

[OZ] http://www.mozart-oz.org

[Parc] http://www-icparc.doc.ic.ac.uk/eclipse/features.html

[Revesz 1998] Peter Z. Revesz (1998) : *Constraint Database: A Survey*.

[Rodosek & Wallace 1998] Rodosek, Wallace (1998) : *A generic model and hybrid algorithm for hoist scheduling*.

[sciam 1998] http://www.sciam.com/1998/0698issue/0698gershenfeld. html

[Singh 1995] Moninder Singh (1995) : *Path Consistency Revisited*.

[SolverRef44] o.V. (1999) : *ILOG Solver 4.4 Reference Manual.*

[SolvManual51] o.V. (2001) : *User's Manual 5.1*

[Stützle 1996] Thomas Stützle : *ECAI'96-Workshop Non-Standard Constraint* Processing. Aus: KI, 1/1997

[Tsang u.a. 1999] Edward Tsang, Christos Voudouris, John Ford, Patrick Mills (1999) : *Operations Research Meets Constraint Programming: Some* Achievements So Far.

[Voudouris & Tsang 1998] Christos Voudouris, Edward Tsang, *Guided Local Search Joins the Elite in Discrete Optimisation, 1998*

[Walsh 2001] Toby Walsh (2001) : *Search on High Degree Graphs*.

[Wegener 1997] I. Wegener (1997) : *Komplexität*. Aus Informatikhandbuch

Erklärung

Ich erkläre hiermit, dass die vorliegende Arbeit von mir selbst und ohne fremde Hilfe verfasst wurde.
Alle benutzten Quellen sind im Literaturverzeichnis angegeben. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mathias Lühr

Potsdam, den *21.04.2002*