Ruprecht-Karls-Universität Heidelberg

Institut für Informatik

Sommersemester 2014

SEMINARARBEIT

Constraint-Programmierung für Scheduling

bei Prof. Dr. Gerhard Reinelt
Dipl.-Math. Stefan Wiesberg
Dipl.-Math. Achim Hildenbrandt

Ekaterina Tikhoncheva

Matrikelnummer 3237882

Geburtsdatum: 27.03.1990
tikhoncheva@stud.uni-heidelberg.de

29. Juni 2014

Inhaltsverzeichnis

Abbildungsverzeichnis

Einleitung

Die Constraint Logische Programmierung (engl. Constraint Logic Programming) oder einfach Constraint Programmierung (kurz *CP*) ist ein Programmierparadigma, das unbekannte Variablen und Beziehungen zwischen ihnen durch Nebenbedingungen (Constraints) beschreibt.
Es kam ursprünglich aus dem Bereich der künstlichen Intelligenz und gilt als eine Erweiterung der Ideen der logischen Programmierung.
Der Constraint-Programmierung liegen zwei Hauptprinzipien zugrunde [vgl Baptiste et al., 2001] : Deduktion der zusätzlichen Nebenbedingungen aus den vorhandenen durch logische Folgerungen und Anwendung der Suchalgorithmen zum Untersuchen des Lösungsraums.
Das Betrachten von Problemen bezüglich der logischen Beziehungen zwischen ihren Objekten erlaubt oft eine einfachere und natürliche Formulierung des Modells für diese Probleme. Kombiniert mit der Benutzung effektiver Suchalgorithmen macht dies die Anwendung von Constraint-Programmierung sehr attraktiv für viele kombinatorische Probleme.
In dieser Arbeit betrachten wir den Ansatz der Constraint-Programmierung in der Scheduling Theorie.
Die Arbeit besteht aus 3 Kapiteln. Im ersten machen wir den Leser mit den Grundideen und Algorithmen der Constraint-Programmierung bekannt. Wir vergleichen auch die Unterschiede zwischen der Benutzung von Mixed-Integer-Programmierung und Constraint Programmierung zur Lösung von Scheduling-Problemen. Im zweiten Kapitel betrachten wir die Anwendungsbeispiele von CP auf konkrete Scheduling-Probleme.
Diese Ausarbeitung orientiert sich hauptsächlich an den Definitionen und der Notation aus Kanet et al. [2004]. Die zusätzliche Quellen werden an jeder Stelle explizit kenntlich gemacht.

Kapitel 1

Constraint Programming

Kanet et al. [2004] definieren **Constraint Programming** (deutsch: Bedingungsprogrammierung, Constraintsprogrammierung) als eine Methode zur Formulierung und Lösung eines diskreten Bedingungserfüllungs- oder Bedingungsoptimierungsproblems durch systematische Anwendung deduktiver Folgerungen, um den Suchraum zu minimieren. CP ist eine Erweiterung der logischen Programmierung, da sie erlaubt, verschiedene mächtige Suchalgorithmen zu benutzen und sie für jedes spezifische Problem entsprechend anzupassen. Das macht dieses Programmierparadigmas sehr flexibel, fordert aber eine bestimmte Erfahrung in der deklarativen logischen Programmierung und in der Entwicklung von Suchalgorithmen [vgl Kanet et al., 2004].
In diesem Kapitel definieren wir die Hauptbegriffe aus der Theorie der CP, stellen das allgemeine CP Programmierungsgerüst vor und am Ende machen wir einen Vergleich zwischen CP und gemischter ganzzahliger Optimierung (Mixed-Integer Programming, **MIP**).

1.1 Constraint Satisfication Problem

Constraint Satisfication Problem (deutsch: Bedingungserfüllungsproblem, **CSP**) ist ein Problem der Zuweisung der gegebenen Variablen die Werte aus der gegebenen Definitionsbereichen, so dass alle Bedingungen an die Variable erfüllt sind [vgl Brailsford et al., 1999]. Formeller:
Ein CSP ist gegeben durch die Menge $X = \{x_1, x_2, \ldots, x_n\}$ diskrete Variablen zusammen mit ihren endlichen Definitionsbereichen $\{D_1, D_2, , \ldots D_n\}$ und die Menge der Bedingungen $C_{ijk\ldots}$ zwischen den Variablen $x_i, x_j, x_k, \ldots$, die die möglichen Werte der Variablen zusätzlich einschränken.
Im allgemeinen können die Variablen von verschiedenen Typen sein (ganzzahlig, boolean, symbolisch, Mengen). Dafür gibt es verschiedene Typen der Bedingungen:

- mathematische (Fertigstellungszeit = Startzeit + Bearbeitungszeit),
- disjunktive (Jobs J_1 und J_2 müssen an verschiedenen Maschinen abgearbeitet werden),
- relational (die Maschine I kann höchstens vier Jobs abarbeiten),

- explizite (die Arbeiten J_1, J_2 und J_5 können nur auf der Maschine 1 abgearbeitet werden).

Ein CSP besitzt eine **(zulässige) Lösung**, wenn eine Zuweisung der Werten aus den Definitionsbereichen zu jeder Variable existiert, sodass alle Bedingungen erfüllt sind. Abhängig von der Aufgabe kann man sich für eine oder für alle zulässige Lösungen interessieren.

1.1.1 Allgemeiner Algorithmus zur Lösung des Constraint Satisfaction Problems

Auf der Abbildung 1.1 (Kanet et al. [2004]) ist der allgemeine Algorithmus zur Lösung eines CSP dargestellt.
Man beginnt mit der Definition der Variablen, ihren Definitionsbereichen und Bedingungen (siehe Block 1).
Block 2 repräsentiert den Prozess der **Bedingungsfortpflanzung** (engl. **constraint propagation**) und der **Reduktion von Definitionsbereichen** (engl. **domain reduction**). Das bedeutet, dass aus den vorhandenen Bedingungen durch die Anwendung der auf der Logik basierten Filteralgorithmen neue Bedingungen abgeleitet werden, die systematisch die Definitionsbereiche von Variablen reduzieren oder zu einem Widerspruch führen.
Nach der Reduktion von Definitionsbereichen bestehen zwei Möglichkeiten (Block 3): Entweder ist eine Lösung des Problems gefunden oder nicht. Im ersten Fall terminiert der Algorithmus. Im zweiten wird geprüft, ob das Problem widersprüchlich ist (Block 4).
Wenn das Problem nicht widersprüchlich ist (siehe Pfeil 6), wendet man ein Suchverfahren für **die Verzweigung** (engl. **branching**) an. Das Ziel der Verzweigung ist die Aufteilung des ursprünglichen Problems in eine Menge von sich gegenseitig ausschließenden Teilproblemen, die aber zusammen die Lösung des Problems vollständig beschreiben. Es wird ein so genannten Entscheidungsbaum aufgebaut.
Jeder Zweig repräsentiert das Hinzufügen der zusätzlichen Bedingungen, die das Problem einschränken. Bei der Verzweigung wird ein Zweig gewählt und man beginnt erneut mit der Bedingungsfortpflanzung (Block 2), um das Teilproblem zu lösen.
Wenn das Problem sich in Block 4 als widersprüchlich erwiesen hat, wird in Block 5 geprüft, ob alle Teilprobleme untersucht wurden. Falls nicht (Block 7), geht der Algorithmus im Entscheidungsbaum eine Stufe höher, wählt ein anderes Teilproblem aus und versucht es zu lösen. Sonst ist die Widersprüchlichkeit des Problems nachgewiesen und der Algorithmus terminiert.

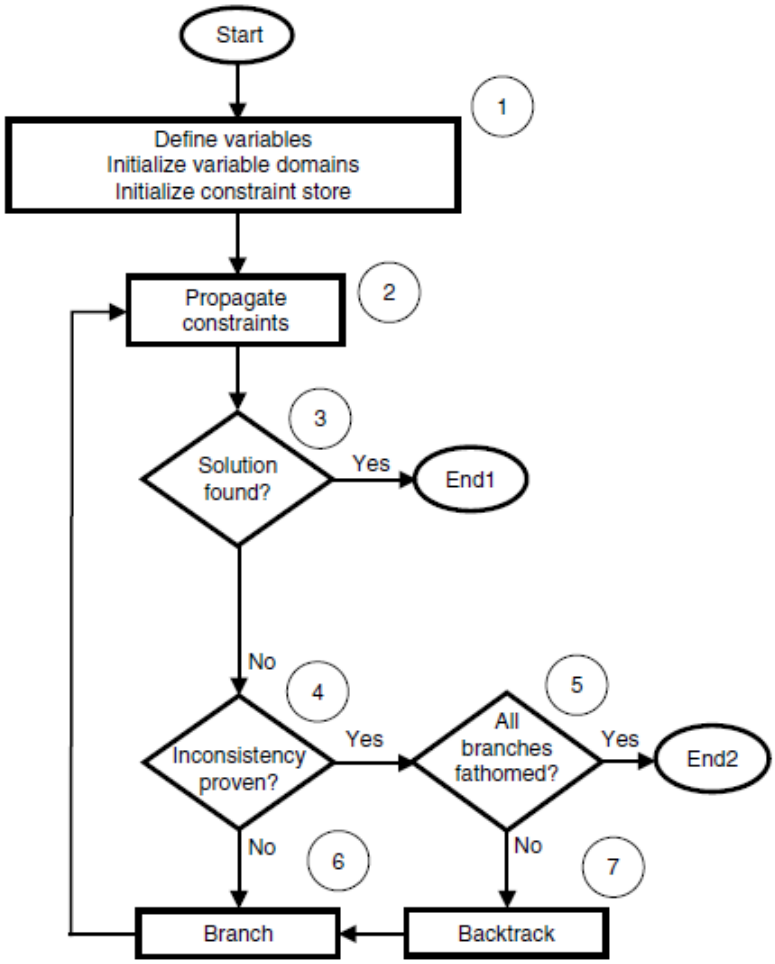

Abbildung 1.1: Allgemeiner Algorithmus für das Constraint Satisfaction Problem [aus Kanet et al., 2004]

1.1.2 Bedingungsfortpflanzung (constraint propagation)

Das Verfahren der Bedingungsfortpflanzung prüft die Widerspruchsfreiheit der im Problem vorhandenen Bedingungen, um die neuen Bedingungen ableiten zu können (eng. **arc consistency checking**).

Hier geht man davon aus, dass die Bedingungen zwischen der Variablen eines CSP immer nur zwei Variablen beeinflussen. In diesem Fall kann CSP als ein Bedingungsgraph dargestellt werden. Variablen des CSP bilden die Menge der Knoten des Graphen. Die Knoten sind adjazent, wenn es eine Bedingung gibt, die die entsprechende Variable verbindet [vgl. Brailsford et al., 1999].

Sei eine Bedingung C_{ij} zwischen den Variablen x_i und x_j gegeben. Der Bogen (x_i, x_j) heißt widerspruchsfrei (eng. **consistent**), wenn für jeden Wert $a \in D_i$ ein Wert $b \in D_j$ existiert, sodass die Zuweisungen $x_i = a$ und $x_j = b$ die Bedingung C_{ij} erfüllen (Brailsford et al. [1999]).

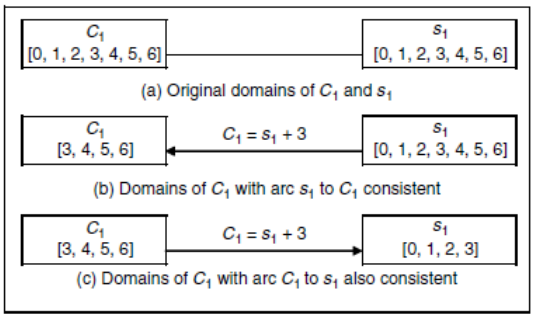

Abbildung 1.2: Reduktion des Suchraums mithilfe der Widerspruchsfreiheit eines Bogens [aus Kanet et al., 2004]

Alle Werte $a \in D_i$, für die dies nicht gilt, können aus dem Definitionsbereich D_i der Variable x_i gelöscht werden, da sie keine zulässige Lösung bilden können. Das Löschen von solchen Werten macht den Bogen widerspruchsfrei. Ein einfaches Beispiel der Bogenwiderspruchsfreiheit ist auf der Abbildung 1.2 (Kanet et al. [2004]) dargestellt.

Offensichtlich ist, wenn alle Bogen widerspruchsfrei gemacht wurden, ist der Suchraum des Problems kleiner geworden (engl. **domain reduction**). Nach der Reduktion des Suchraums soll das Suchen nach der Lösung einfacher werden.

Zu bemerken ist, dass die Bedingungsfortpflanzung die Information der Definitionsbereiche der Variablen nicht nur innerhalb einer Bedingung benutzen kann, sondern auch zwischen mehreren Bedingungen. Dazu das Beispiel auf der Abbildung 1.3 (Kanet et al. [2004]). Das führt zu der so genannten Reduktion des Suchraums zwischen den Bedingungen (eng. **between-constraint domain reduction**).

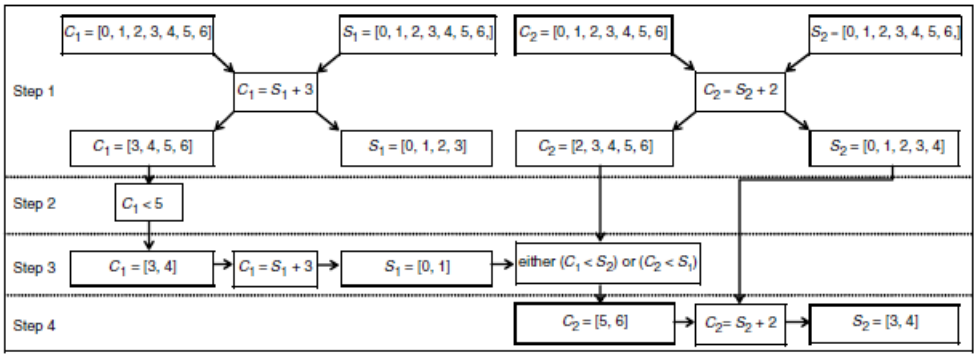

Abbildung 1.3: Reduktion des Suchraums mithilfe der Widerspruchsfreiheit mehreren Bogen [aus Kanet et al., 2004]

1.1.3 Branching

Typischerweise wird das Backtracking-Verfahren fürs Branching (siehe Block 6 auf der Abbildung 1.1) verwendet, welches aber oft ineffizient ist. Deswegen werden auch andere Verfahren wie Forward checking oder MAC (von engl. maintaining arc consistency) verwendet [vgl. Brailsford et al., 1999]. Ihre Vorteile sind, dass sie so genannte lookahead Verfahren sind. D.h. jeder Zweig beeinflusst die Definitionsbereichen aller Variablen und nicht nur derjenigen, die im Suchbaum früher betrachtet wurden.

Auf jeden Fall bleibt die Frage offen, wo als erstes verzweigt werden muss. An dieser Stelle werden verschieden Heuristiken verwendet, wie z.B. es wird als erstes in die Variable mit dem kleinsten Definitionsbereich verzweigt.

Nachdem die Wahl der Variable getroffen wurde, wird ein neuer Zweig im Suchbaum gebildet, indem man dieser Variable einen Wert aus ihrem Definitionsbereich zuweist. Hier wird wiederum heuristisch entschieden, welchen Wert diese Variable annehmen soll. Häufige Vorgehensweise ist den kleinsten Wert aus dem Definitionsbereich der Variable zu nehmen.
In allgemeinen kann die Verzweigung auf verschiedene Weise verlaufen: es kann einen Wert zu einer Variable, mehrere Werte zu einer Variable oder mehrere Werte zu mehreren Variablen zugewiesen werden.

1.2 Constraint Optimierungsproblem

Das klassische CSP kann einfach zur **Constraint Optimierungsproblem** (engl. Constraint Optimization Problem, **COP**) erweitert werden.
Angenommen unser COP hat eine Zielfunktion Z, die minimiert werden soll. Wenn wir das Problem ohne Zielfunktion betrachten, ist es ein CSP und seine Lösung können wir mit Hilfe des Algorithmus von der Abbildung 1.1 finden. Sobald eine Lösung von CSP gefunden wurde, berechnen wir den Wert der Zielfunktion (Z') und fügen die Bedingung $Z < Z'$ zum Problem hinzu. Als Ergebnis bekommen wir ein neues CSP, suchen nach seiner Lösung und so weiter, bis es zu einem Widerspruch gekommen ist und alle Zweige im Suchbaum untersucht wurden. Die zuletzt gefundene zulässige Lösung ist somit die Lösung des COP.

1.3 Vorteile von Constraint Programmierung

In diesem Abschnitt möchten wir die Vorteile der Constraint Programmierung vorstellen, die dieses Programmierparadigmas attraktiv zum Lösen von Scheduling-Aufgaben machen.

1.3.1 Indexierung der Variable

Analog zu den Programmiersprachen ermöglicht CP die Benutzung einer Variablen als Indizes von anderen Variablen, was normalerweise die Anzahl der Entscheidungsvariablen reduziert.
Die CP-Formulierungen sind deswegen kompakt und intuitiv verständlich.
Als Beispiel betrachten wir ein Einzelmaschine-Batch-Sequenzierung-Problem mit den Reihenfolgenabhängigen Rüstzeiten aus Jordan and Drexl [1995]. Die Zielfunktion ist die Minimierung von Rüstkosten und Verfrühungstrafen bei den gegeben Rüstzeiten der adjazenten Jobs in der Reihenfolge.
In IP[1] werden binäre Variablen $y[i, j]$ benutzt, um zu zeigen, ob der Job i sofort nach

[1] engl. Integer Programming, de. Ganzzahlige Programmierung

dem Job j abgearbeitet wird. Falls es insgesamt n Jobs gibt, dann braucht man $n(n-1)$ binäre Variablen y, um alle möglichen Reihenfolge der Jobbearbeitung zu definieren. Sei $cost[i,j]$ der Preis der Bearbeitung des Jobs i sofort nach dem Job j. Dann ist der gesamte Preis der Jobsreihenfolge definiert durch $\sum_{i,j,i\neq j} cost[i,j]y[i,j]$.
Um dieses Problem in CP formulieren zu können, definiert man die Entscheidungsvariablen $job[k]$, deren Wert dem Index desjenigen Jobs gleich ist, der als k-ter in der Reihenfolge abgearbeitet wird. Es ist klar, dass es insgesamt n solcher Variablen gibt. Mit diesen Variablen is der gesamte Preis der Jobsreihenfolge gleich $\sum_{k>1} cost[job[k-1], job[k]]$.
Somit ist die Anzahl der Variablen des Problems von $n(n-1)$ auf n reduziert.

1.3.2 Nebenbedingungen

Die CP verfügt über eine Menge von Operationen, die es ermöglichen, viele Bedingungen einfach zu formulieren. Wir beschreiben hier einige Bedingungen aus der Scheduling-Theorie, die sich sehr einfach in CP, aber nicht in IP formulieren lassen (siehe Tabelle 1.1).
Angenommen wir haben ein Problem mit zwei Maschinen. Die Variablen *MaschineA* und *MaschineB* können im ersten Fall den Wert 0 haben, falls es keinen Job gibt, der auf der Maschine abgearbeitet werden muss, oder 1 ansonsten. Wir suchen nach einer solchen Zuordnung der Jobs zu diesen Maschinen, so dass nur eine Maschine in Betrieb sein kann.

CP	IP
Fall 1: Ungleichungen mit binären Variablen	
$MaschineA \neq MaschineB$	$MaschineA + MaschineB = 1$
Fall 2: Ungleichungen mit ganzzahligen Variablen	
$MaschineA \neq MaschineB$	$(MaschineA - MaschineB - \varepsilon + \sigma BigM \geq 0)$ $(MaschineB - MaschineA - \varepsilon + (1-\sigma)BigM \geq 0)$ $\sigma \in 0,1$
Fall 3: Logische Bedingungen	
$(A.start > B.end) Xor (B.start > A.end)$	$(A.start - B.end - \varepsilon + \sigma BigM \geq 0)$ $(B.start - A.end - \varepsilon + (1-\sigma)BigM \geq 0)$ $\sigma \in 0,1$

Tabelle 1.1: Unterschiede zwischen der Formulierung von Bedingungen in CP und IP

Im zweiten Fall seien die Variablen *MaschineA* und *MaschineB* ganzzahlig und ihre Werte sind die Nummern der Jobs, die auf diesen Maschinen abgearbeitet werden. Die Bedingung, die wir in diesem Fall betrachten, lautet „ein Job kann nicht auf zwei Maschinen abgearbeitet werden“. Die entsprechende Formulierung in CP und IP findet man in der Tabelle 1.1. Das Problem, das wir hier haben, ist das man in *IP* für die Ungleichheit zwei Nebenbedingungen mit $<$ und $>$ entsprechend braucht. Da es in IP aber nur Operationen $\leq$, $\geq$ gibt, braucht man eine zusätzliche Variable ε. Die „gross M“–Notation wird benutzt, um logische Operation Xor realisieren zu können. $\sigma = 1$ bedeutet

$MaschineB > MaschineA$ und $\sigma = 0$ implizit $MaschineA > MaschineB$.
Endlich im dritten Fall (Fall 3 in der Tabelle 1.1) betrachten wir die Fertigstellungszeit-Bedingung: falls Job *A* und Job *B* auf einer Maschine abgearbeitet werden, dann muss die Bearbeitung von Job *A* früher fertig werden, als die Bearbeitung vom Job *B* beginnt, oder umgekehrt.
Ein anderes gutes Beispiel sind Bedingungen, die für alle Variablen gelten. In CP sind viele solche Bedingungen schon vordefiniert. Z.B. die Nebenbedingung $alldifferent(Machine)$ stellt sicher, dass kein Job auf zwei Maschinen abgearbeitet wird. Die Formulierung dieser Bedingung in IP benötigt $m(m-1)$ Nebenbedingungen (m ist die Maschinenanzahl) und $m(m-1)/2$ zusätzliche binäre Variablen.
Aus diesen Beispielen ist es offensichtlich, dass sich viele Bedingungen im CP direkt und selbst erklärend formulieren lassen.

1.4 Constraint und ganzzahlige Programmierung

Die beiden Methoden Constraint und ganzzahlige Programmierung sind ganz unterschiedlich und lassen sich nicht miteinander vergleichen. So macht IP sich der mathematischen Struktur des Problems voll zunutze, während CP dem Forscher großen Spielraum im Entwerfen von Nebenbedingungen und Suchalgorithmen lässt.
Es gilt generell, dass die Zielfunktion eine zentrale Rolle in IP spielt. Dabei wird auch versucht, die kleinste Menge der Nebenbedingungen zu finden, die ausreichend ist, den Kernpunkt des Problems zu beschreiben. CP konzentriert sich dagegen auf die Constraints. Alles zusätzliche Wissen des Problem, die als Constraints formuliert und ins Modell hinzugefügt wurden, können die Performanz des Suchalgorithmus verbessern. Die Suchverfahren stützen sich seinerseits weniger auf die mathematische Struktur der Zielfunktion oder des Constraints, aber mehr auf die spezifischen Aspekte des Problems.
Die Wahl, welche von beiden Paradigmen besser zu benutzen ist, hängt deswegen in der ersten Linie von dem Problem, den Problemgrößen, den Daten, dem Model, aber auch von den Forschern und der Software, die er benutzt, ab. Außerdem es ist auch möglich beide Methoden zu kombinieren.

Kapitel 2

Anwendung von CP in der Scheduling Theorie

In diesem Kapitel betrachten wir das Formulieren und Lösen einiger Probleme der Scheduling Theorie mit Constraint Programmierung.
Bevor wir tiefer einsteigen, möchten wir die wichtigste Eingenschaften von CP auflisten, die den Ansatz von CP zum Lösen der Scheduling-Probleme besonders attraktiv machen:

- CP ist besonders für ganzzahlige Variablen geeignet. Die Reduktionsmethoden funktionieren auch am besten für endliche Definitionsbereiche der Variablen.
- CP passt am besten zum Lösen von Problemen mit vielen Nebenbedingungen.
- CP ist mehr effektiv in Fällen vieler Nebenbedingungen mit wenigen Variablen in jedem einzelnen Constraint.
- Der Ansatz von CP lohnt sich besonders für Probleme, die sich als eine optimale Abbildung einer geordneten Menge in eine andere Menge darstellen lassen.

2.1 Minimieren der gewichteten Gesamtverspätung

Wir betrachten den Fall eines Schedule mit einer Maschine und der Aufgabe die gewichtete Gesamtverspätung zu minimieren. In der $\alpha|\beta|\gamma$ Notation aus Pinedo [2012] wird dieses Problem als $1||\sum w_j T_j$ bezeichnet.
Obwohl sich die Problemstellung einfach formulieren lässt, ist es ein bekanntes *NP*-hartes Problem. Es gibt einen Lösungsansatz mittels Dynamischer Programmierung, der einen pseudopolynomialen Algorithmus liefert [vgl. Pinedo, 2012].
Die einfachste Formulierung des Problems in CP benötigt n Variablen s_j mit dem Definitionsbereich $[0 \cdots \sum_{j=1..n} p_j]$ für Startzeiten und n Variablen C_j mit dem Definitionsbereich $[0 \cdots \sum_{j=1..n} p_j]$ für die Fertigstellungszeiten von Jobs. Die Zielfunktion zusammen mit den Nebenbedingungen lassen sich wie folgt ausdrücken:

$$\begin{aligned} & \min \sum_j w_j T_j & \\ s.t. \quad & C_j = p_j + s_j & \forall j = 1..n \\ & C_j \leq s_k \vee C_k \leq s_j & \forall j,k = 1..n,\ k > j \end{aligned}$$

In dieser Formulierung sind insgesamt $2n$ Variablen und $\frac{n(n+1)}{2}$ Nebenbedingungen definiert.
Eine andere Formulierung ist möglich, indem man n Variablen s durch n Variablen pos ersetzt, wobei $pos[j]$ die Position des Jobs j in der Bearbeitungsreihenfolge ist. Der Definitionsbereich von pos ist offensichtlich $[1..n]$. Die Nebenbedingungen für diese Formulierung lauten:

$$
\begin{aligned}
&pos_j \neq pos_k && \forall j,k = 1..n,\ k > j \\
&pos_j > pos_k \Leftrightarrow C_j \geq C_k + p_j && \forall j,k = 1..n,\ k \neq j \\
&pos_j = 1 \Rightarrow C_j = p_j && \forall j = 1..n
\end{aligned}
$$

Diese Formulierung enthält $\frac{3n^2-n}{2}$ Nebenbedingungen, aber ihre Optimierung ist viel langsamer, als beim ersten Modell. Eine mögliche Verbesserung wäre das Hinzufügen von zusätzlichen Nebenbedingungen, die aus der folgenden Überlegung entstehen. Wird ein Job auf der Position j platziert, ist die untere Schranke seiner Fertigstellungszeit gleich der Summe seiner Bearbeitungszeit und der Summe p_i von $j-1$ Jobs mit den kleinsten Bearbeitungszeiten. Das bedeutet[1]:

$$
\begin{aligned}
pos_j = k \wedge j <= k \Rightarrow C_j \geq \sum_{l<=k} p_l \\
pos_j = k \wedge j > k \Rightarrow C_j \geq p_j + \sum_{l<k} p_l
\end{aligned}
$$

Um die Lösung eines Problems zusätzlich zu beschleunigen, können verschiedene heuristische Verfahren an den Suchverfahren angewendet werden. Für das betrachtete Problem kann man z.B Heuristik WMDD („*weighted modified due date* “) [siehe Kanet and Li, 2004] benutzen.

2.2 Job Shop Scheduling

Eine häufig vorkommende Aufgabe in der Scheduling-Theorie ist die Zuordnung von n Jobs zu m Maschinen. Jeder Job besteht dabei aus einer Menge von Operationen O_j, die in einer bestimmten Reihenfolge abgearbeitet werden müssen. Typische Bedingungen sind, dass jede Maschine zu jedem Zeitpunkt nur eine Operation bearbeiten kann und umgekehrt dass jede Operation zu jedem Zeitpunkt nur auf einer Maschine abgearbeitet wird. Das Problem mit der Zielfunktion C_{max} (Minimierung des Makespan) wird in der Notation aus Pinedo [2012] als $Jm||C_{max}$ bezeichnet.
Zur Modellierung dieses Problems verwendet man so genannte *disjunktive Graphen* und zur Lösung das *Branch & Bound* Verfahren. Wir betrachten den Ansatz von CP zur Lösung des beschriebenen Job Shop Problem [vgl. Brailsford et al., 1999].

[1] Jobs sind in aufsteigender Reihenfolge der Bearbeitungszeiten geordnet

Wir definieren eine Variable s_o für die Startzeit der Operation $o \in O_j$. Offensichtlich gilt $s_o \in \{0,1,\ldots C\}$, wobei C eine obere Schranke für C_{max} ist. Man kann aber den Definitionsreich von diesen Variablen ein bisschen einschränken, wenn man den Vorgänger $pred(o)$ und den Nachfolger $succ(o)$ von einer Operation o betrachtet. Dann liegen s_o in $[\sum_{o' \in pred(o)} p_{o'} \ldots\ C - p_o - \sum_{o' \in succ(o)} p_{o'}]$.
Die Nebenbedingungen lassen sich wie folgt definieren:

$$\begin{aligned} s_o + p_o \leq s_{o'} \qquad & o, o' \in O_j,\ o' \in succ(o),\ j = 1 \ldots n \\ s_o + p_o \leq C \qquad & o \in O \\ s_o + p_o \leq s_{o'} \ \vee\ s_{o'} + p_{o'} \leq s_o \qquad & o, o' \in O, o \neq o', m_o = m_{o'} \end{aligned}$$

wobei O die Menge aller Operationen bezeichnet. Die erste Nebenbedingung garantiert, dass die Operationen auf einen Job sich nicht unterbrechen lassen. Mit der zweiten Nebenbedingung wird sicher gestellt, dass C eine obere Schranke an den Makespan ist. Und die dritte Nebenbedingung besagt, dass unter zwei Operationen o und o', die auf der gleichen Maschine abgearbeitet werden, eine vor der anderen abgearbeitet wird oder umgekehrt.
Wie kann man die Bedingungsfortpflanzung für dieses Modell effektiv durchführen? Eine Möglichkeit wurde in der Arbeit von Nuijten and Aarts [1996] vorgeschlagen. Für jede Operation o definiert man die frühst möglichste Startzeit $est(o)$ und die spät möglichste Startzeit $lst(o)$. Die Werte von $est(o)$ und $lst(o)$ sind entsprechend der kleinste und der größte Wert aus dem Definitionsbereich von s_o.
Aus der ersten Nebenbedingung folgt, dass $s_o + p_0 \leq lst(o')$ und $est(o) + p_o \leq p_{o'}$ gilt. Deswegen werden die Werte größer als $lst(o') - p_o$ und kleiner als $est(o) + p_o$ aus den Definitionsbereichen von s_o und $s_{o'}$ entsprechend entfernt.
Aus der dritten Nebenbedingung folgt $s_o \leq lst(o') - p_o$, falls die Operation o vor der Operation o' auf einer Maschine abgearbeitet wird, oder $s_o \geq est(o') + p_{o'}$, falls umgekehrt. Wenn $lst(o') - p_o + 1 \leq est(o') + p_{o'} - 1$ gilt, dann können alle Werte aus $[lst(o') - p_o + 1 \ldots est(o') + p_{o'} - 1]$ aus dem Definitionsbereich von s_o gelöscht werden.

2.3 Timetabling

In diesem Abschnitt betrachten wir ein Beispiel der Erstellung eines Zeitplanes (engl. Timetable) mit Hilfe von Constraint Programmierung. Das Beispiel stammt aus dem Artikel von Henz [2002] und beschreibt Scheduling eines Rundenturniers für Basketball-Spiele.
Bevor Henz [2002] wurde die Aufgabe mittels ganzzahliger Programmierung und erschöpfender Suche [siehe Nemhauser and Trick, 1998] gelöst und die Berechnung hat 24 Stunden gedauert. Im Gegensatz dazu hat die Lösung mit Hilfe von CP weniger als eine Minute gebraucht.
Es wurden $n = 9$ Teams betrachtet, für die $2n = 18$ Termine festgelegt werden sollten, sodass an jedem Termin $n-1$ Teams spielen und ein Team nicht spielt (d.h. hat „bye“). Alle

Termine sollten in neun Wochen (von 31.12.1997 bis 1.03.1998) mit einem Wochenende und einen Arbeitstag pro Woche geplant werden.
Das Problem wird in drei Phasen gelöst:

- Generierung von einem Muster (engl. pattern) für jedes Team, das für jeden Termin feststellt, ob Team ein Heim-, Auswärtsspiel oder „bye“ spielen kann. Die Generierung von Mustern hängt von allen formulierten Bedingungen an die Spiele ab.
- Suche nach einer Menge von 9 Mustern, die die Erstellung eines Zeitplanes ermöglichen.
- Generierung eines Zeitplans, indem man anhand der Menge von Mustern die Zuordnung von Spielen zu den Teams und Paare von Gegner bestimmt.

2.3.1 Berechnung von Mustern

Man definiert drei Gruppen von binären Variablen $h_j, a_j, b_j, j = 1 \dots 18$, für Heimspiele, Auswärtsspiele und byes und löst das Problem mit Hilfe von Constraint Programmierung. Einige der betrachteten Bedingungen sind:

- jedes Team hat nur ein Spiel an jedem Termin: $h_j + a_j + b_j = 1$
- s.g. Mirroring: Termine sind paarweise in die vorgegebene Menge m gruppiert, jedes Paar bezeichnet Spiele der gleichen Teams: $h_j = a_{j'},\ a_j = h_{j'},\ b_j = b_{j'}\ (j, j') \in m$
- die beiden letzten Spielen können nicht beide Auswärtsspiele sein : $a_{17} + a_{18} = 2$
- Es sind nicht mehr als zwei Auswärtsspiele nacheinander erlaubt: $a_j + a_{j+1} + a_{j+2} < 3$
- Es sind nicht mehr als zwei Heimspiele nacheinander erlaubt: $h_j + h_{j+1} + h_{j+2} < 3$
- Es sind nicht mehr als drei Auswärtsspiele oder byes nacheinander erlaubt: $a_j + b_j + a_{j+1} + b_{j+1} + a_{j+2} + b_{j+2} + a_{j+3} + b_{j+3} < 4$
- Es sind nicht mehr als vier Heimspiele oder byes nacheinander erlaubt: $h_j + b_j + h_{j+1} + b_{j+1} + h_{j+2} + b_{j+2} + h_{j+3} + +b_{j+3} + h_{j+4} + b_{j+4} < 5$
- An allen Wochenenden spielt jedes Team vier Heim-, vier Auswärtsspiele und ein bye: $\sum_{j\in\{2,4,\dots 18\}} h_j = 4$, $\sum_{j\in\{2,4,\dots 18\}} a_j = 4$, $\sum_{j\in\{2,4,\dots 18\}} b_j = 1$
- Jedes Team spielt ein Heimspiel oder bye mindestens an zwei der ersten fünf Wochenenden $\sum_{j\in\{2,4,6,8,10\}} (h_j + b_j) \geq 2$

Die effektivste Suchstrategie in diesem Fall ist das Durchzählen der möglichen Werte vom h, a, b in der Reihenfolge $h_1, a_1, b_1, h_2, a_2 \dots$.

2.3.2 Menge von Mustern

Es wurden insgesamt 38 Mustern für das betrachtete Problem gefunden, die in drei 38×9 Matrizen h, a und b gespeichert sind. Das Ziel ist eine Menge aus 9 Mustern zu finden, die es erlaubt einen zulässigen Zeitplan zu bilden.
Um diese Aufgabe lösen zu können definieren wir binäre Variablen x_i, $i = 1 \ldots 38$. $x_i = 1$, falls das Muster i in die gesuchte Menge kommt, sonst 0. Das Modell formuliert als CSP lautet:

$$\sum_i x_i = 9$$

$$\sum_i h_{ij} x_i = 4,\ \sum_i s_{ij} x_i = 4,\ \sum_i b_{ij} x_i = 1 \qquad j = 1 \ldots 18$$

$$x_i + x_{i'} < 1 \qquad j = 1 \ldots 18,\ i, i' = 1 \ldots 9,\ i \neq i',$$

$$(h_{ij} = 0 \vee a_{i'j} = 0) \wedge (a_{ij} = 0 \vee h_{i'j} = 0)$$

wobei in der dritten Bedingung die Muster, die keinen zulässigen Spielplan bilden können, extra ausgeschlossen werden.

2.3.3 Erstellung von einem zulässigen Zeitplan

Es gibt mehrere Verfahren, um einen zulässigen Zeitplan aus der gegebenen Menge von Mustern zu erstellen. Das Verfahren, das in beschriebenen Artikel von Henz [2002] verwendet wurde, ordnet erst jedes Team zu einem von 9 vorhandenen Mustern zu und danach sucht es für jedes Team nach den möglichen Gegnern an jedem Termin.
Die Aufgabe der Erstellung eines zulässigen Zeitplanes wird wiederum als ein CP Modell formuliert, das jeweils für jedes Team und jeden Termin einen Gegner und eine Spielart (Heim-/Auswärtsspiel oder bye) berechnet. Die Nebenbedingungen werden von allgemeinen Bedingungen an ein Rundenturnier und Bedingungen aus den Abschnitt 2.3.1 bestimmt.

Kapitel 3

Zusammenfassung

Der Schwerpunkt dieser Arbeit war Constraint Programmierung und ihre Anwendung in der Scheduling-Theorie.
Im ersten Teil wurden wichtige Begriffe und Algorithmen dieses Programmierparadigmas erklärt. Außerdem wurde auf die Vorteile des Constraint Programmierung eingegangen und ein kleiner Vergleich zwischen ihr und ganzzahliger Optimierung gemacht. Dies macht diese Arbeit zu einer kurzen Einführung in Constraint Programmierung für Anfänger.
Im zweiten Teil wurden einige Beispiele aus der Scheduling Theorie betrachtet und wie sie mit Hilfe von Constraint Programmierung formuliert und gelöst werden können. Es wurde mit einem einfacheren Beispiel der Minimierung der Gesamtverspätung angefangen, um zu zeigen wie einfach ein Problem in CP formuliert werden kann. Und das Beispiel mit Timetabling am Ende zeigt noch mal, das Benutzung von Contraint Programmierung anstatt ganzzahliger Optimierung zu signifikanter Verbesserung der Laufzeit führen kann.
Es existieren aber weitere interessante Ansätze der CP im Bereich der Scheduling-Theorie, die in dieser Arbeit nicht betrachtet wurden. In dieser Richtung kann die Arbeit erweitert und vervollständigt werden.

Literaturverzeichnis

Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. International Series in Operations Research & Management Science. Springer US, 2001.

S. C. Brailsford, C. N. Potts, and B. M. Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operation Research*, 119:557 – 581, 1999.

M. Henz. Scheduling a major college basketball conference. *Operations Research*, 49: 163–168, 2002.

C. Jordan and A. Drexl. A comparison of constraint and mixed-integer programming solvers for batch sequencing with sequence-dependent setups. *ORSA Journal of Computing*, 7:160 – 165, 1995.

J. J. Kanet and X. Li. A weighted modified due date rule for sequencing to minimize weighted tardiness. *Journal of Scheduling*, 7:261 – 276, 2004.

J. J. Kanet, S. L. Ahire, and M. F. Groman. Constraint programming for scheduling. In J. Y. T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Taylor & Francis, 2004.

G. L. Nemhauser and M. A. Trick. Scheduling a major college basketball conference. *Operations Research*, 46:1–8, 1998.

W. P. N. Nuijten and E. H. L. Aarts. A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *European Journal of Operation Research*, 90:269–284, 1996.

Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 4th edition, 2012.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe.
Heidelberg, im Juli 2014

.....

Ekaterina Tikhoncheva