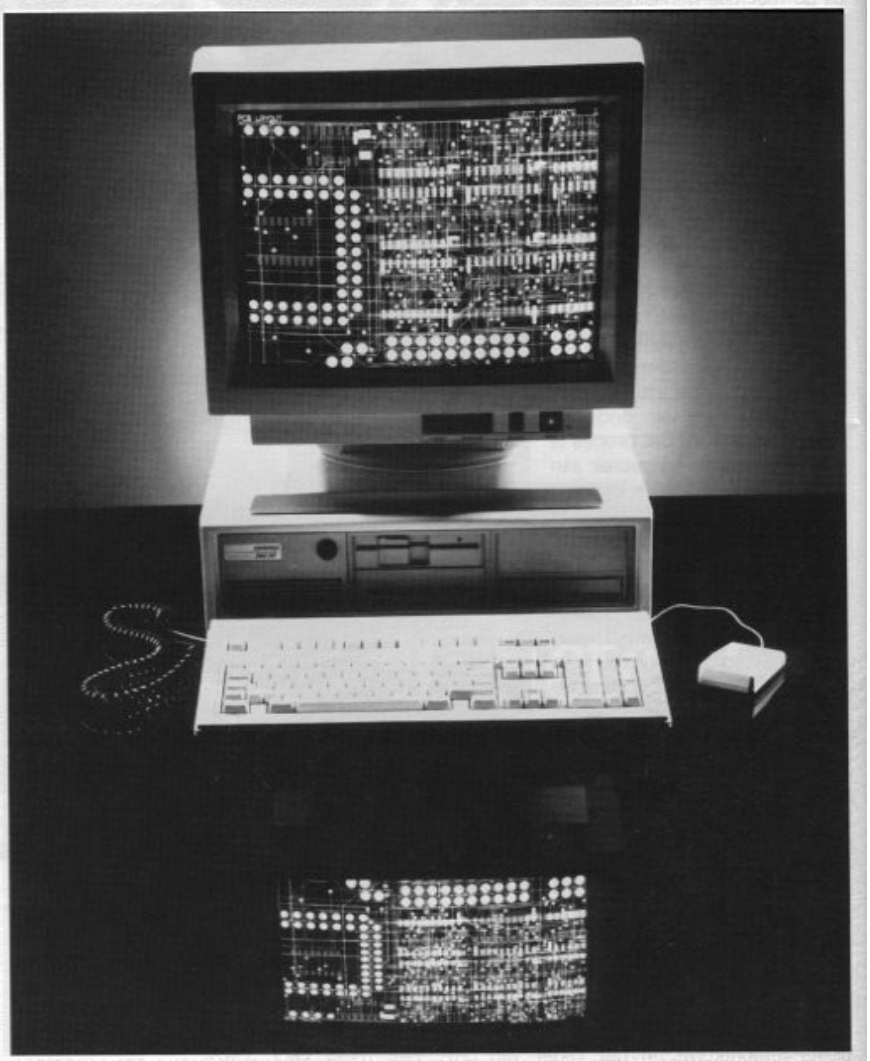
Beperkingen verkleinen de oplossingsruimte

Hoe kleiner de hooiberg des te groter de speld

Er is een klasse van problemen waarvoor een oplossing letterlijk gevonden moet worden, namelijk door een mogelijke oplossingsruimte te doorzoeken. Een dergelijke ruimte is vaak bepaald door gegeven beperkingen en randvoorwaarden (constraints) en er zijn strategieën ontwikkeld om de inherente combinatorische complexiteit te beteugelen. Programmeertalen waarin het probleem door middel van beperkingen beschreven kan worden, en die genoemde strategieën als executeerbare functie bevatten, heten talen voor Constraint Logic Programming (CLP) of, algemener, Constraint Programming Languages (CPL). Zowel in de wetenschap als in de praktijk staan deze declaratieve talen in de belangstelling. Een produkt dat op dit moment al bedrijfsmatig wordt toegepast is Charme van het Franse Bull.



Constraint Logic Programming benadert problemen door de oplossingsruimte te doorzoeken aan de hand van gegeven randvoorwaarden. Een kenmerkend voorbeeld van zo'n probleem is de routering van printplaten.

Procedurele programmeertalen als Pascal of C zijn bij uitstek geschikt om algoritmen in uit te drukken en iedereen die wel eens een standaardbibliotheek heeft gebruikt, weet hoe belangrijk het is om algemene oplossingsmethoden direct op specifieke data te kunnen uitvoeren. Bij objectgeoriënteerde talen worden de mogelijke operaties en de datatypen waarop zij kunnen werken zelfs als één geheel genomen.
Nog prettiger zou het wezen, in principe althans, als de programmeur alleen nog

ir. J.M. VAN THIEL

maar het probleem hoefde te formuleren omdat de oplossingsmethode als zodanig al in een redeneermechanisme zou zijn geïmplementeerd. Een dergelijke programmeertaal omvat dan eigenlijk alleen maar een datasyntaxis, een formalisme om gegevens en gegevensstructuren in te beschrijven.
Deze wens is, tenminste gedeeltelijk, gerealiseerd in logisch programmeren. Hierbij bestaat een systeem vrijwel alleen nog maar uit feiten (gegevens) en logische verbanden in de vorm van predicaten en proposities. Het grootste deel van bijvoorbeeld een programma in Prolog, de bekendste logische taal, bestaat uit de beschrijving van voorkomende logische verbanden. Interne operaties en stroomsturing worden als een taak van het inferentiemechanisme beschouwd en de explicitering daarvan blijft dan ook bij voorkeur tot een minimum beperkt.

Beperkingen

Onder meer vanwege dit declaratieve karakter is logisch programmeren het uitgangspunt geworden bij de ontwikkeling van talen voor Constraint Logic Programming (CLP). In feite gaat het hierbij om een uitbreiding om het oorspronkelijke concept van logisch programmeren. Dat er veel overeenkomsten zijn, blijkt duidelijk uit de structuur van CLP-programma's, die sterk lijkt op die van Prolog en consorten.
Het belangrijkste verschil tussen de twee categorieën zit in het feit dat een CLP-gebaseerd inferentiemechanisme zijn oplossing niet alleen vindt door te redeneren met de gedefinieerde gegevens en proposities. Hij komt ook achter potentiële oplossingen door middel van toetsing aan de gegeven beperkingen en randvoorwaarden, de *constraints*.
Van Hentenryck, Simonis en Dincbas (ref. 1) schrijven dat het ontwerp en de implementatie van de eerste CLP-talen als Chip, CLP, Prolog II en Prolog III pogingen waren om de voordelen van logisch programmeren te behouden en de nadelen te vermijden. Het Franse bedrijf Bull is met Charme een andere weg gegaan, in elk geval wat de syntaxis en semantiek betreft. Dit produkt is eigenlijk ook geen CLP meer maar, algemener, een *Constraint Programming Language* (CPL).

Puzzels

De verzameling van problemen waarvoor de oplossing moet worden bepaald uit gegeven randvoorwaarden is omvangrijk en zeer divers. Kenmerkende voorbeelden in de elektronicahoek zijn zaken als de verificatie van schakelingen en het routeren van printplaatontwerpen. Meer in het algemeen gaat het om optimalisatievraagstukken die moeilijk hanteerbaar zijn vanwege hun combinatorische complexiteit.
Een praktische manier om de grondgedachte van CLP aanschouwelijk te maken is aan de hand van geschikte puzzels. Veel van dergelijke vraagstukken vallen namelijk in dezelfde klasse problemen maar hebben als voordeel dat het aantal mogelijkheden overzichtelijk blijft. Een goed voorbeeld daarvan is de puzzel van de drie kabouters die elk of een rode of een witte muts krijgen opgezet zonder dat zij die zelf kunnen zien. Wel weten ze dat er in totaal vijf mutsen beschikbaar waren: drie witte en twee rode. Bovendien zijn ze zeer schrander en zullen ze alle beschikbare informatie ten volle benutten.
De kabouters zitten achter elkaar en mogen niet omkijken zodat ze alleen hun eventuele voorgangers kunnen zien. Aan elk van hen, te beginnen bij de achterste, wordt nu gevraagd wat de kleur van zijn eigen muts is. De achterste zegt dat hij het niet weet. Vervolgens zegt de middelste dat hij het ook niet weet. „Wel", zegt de voorste, die alleen maar het bos voor zich kan zien: „dan weet ik het zeker." De gevraagde oplossing is van welke kleur muts hij zo zeker is.

combinatie	A	B	C
1	w	w	w
2	w	w	r
3	w	r	w
4	w	r	r
5	r	w	w
6	r	w	r
7	r	r	w
8	r	r	r

Tabel 1. De oplossingsruimte van het mutsenprobleem bevat maar acht mogelijkheden.

combinatie	A	B	C
1	w	w	w
2	w	w	r
3	w	r	w
5	r	w	w
6	r	w	r
7	r	r	w

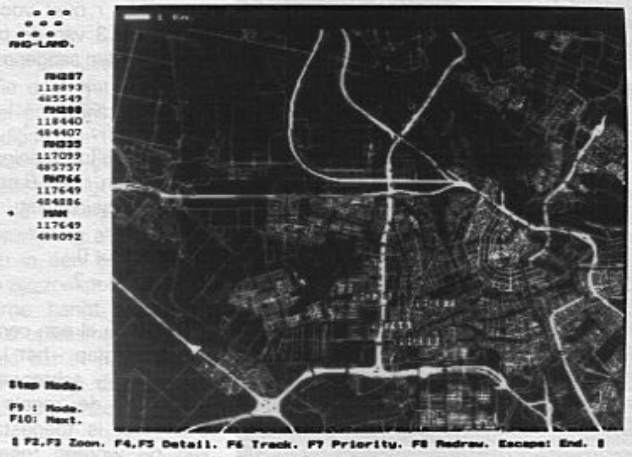
Tabel 2. Omdat combinatie 8 meteen afvalt en, gegeven de aanname voor B en C, ook mogelijkheid 4 van tabel 1 kan worden geëlimineerd, blijven na de eerste stap nog maar zes opties over.

combinatie	A	B	C
1	w	w	w
3	w	r	w
5	r	w	w
7	r	r	w

Tabel 3. Wil voor kolom B elke keuzemogelijkheid open blijven dan moet tabel 2 worden gereduceerd tot onderstaand lijstje. Hieruit volgt onvermijdelijk dat C = w.

Randvoorwaarden

Omdat het aantal combinaties in dit geval zo klein is, zijn zij uit te schrijven zoals in tabel 1 is gedaan. Per kabouter zijn er maar twee mogelijkheden – rood of wit – waarmee het totaal op acht komt. Bovendien sluit de eerste randvoorwaarde, namelijk dat het aantal rode mutsen maximaal twee kan zijn, mogelijkheid 8 direct al uit.
Nadat het eerste antwoord is gegeven, ontstaat de situatie in tabel 2. Hierin is ook de vierde regel weggevallen omdat in deze situatie – en alleen in deze – kabouter A juist wel had geweten welke kleur muts hij op heeft. Omdat de tweede kabouter evenmin zekerheid heeft over zijn hoofddeksel, vallen mogelijkheden 2 en 6 ook af. In dat geval zou hij immers kunnen concluderen dat zijn eigen muts wit is en dat kan hij juist niet. ▷



Optimalisatievraagstukken, bijvoorbeeld het zoeken naar de beste route, lenen zich uitstekend voor CLP.

▷ De overblijvende mogelijkheden staan in tabel 3 en die laten voor de derde kabouter slechts één conclusie toe: zijn muts is wit. Overigens kan hij eenzelfde redenering toepassen bij elk van de andere mogelijke antwoorden van zijn voorgangers. Daar zal, althans voor kabouter C, ook steeds een eenduidig resultaat uitrollen.

Terugredeneren

De lezer die de gedachtengang aan de hand van de tabellen controleert, zal merken dat hij of zij in feite gebruik maakt van terugredeneren (*backtracking*). Aan de hand van de inhoud van kolom C wordt nagegaan wat dit voor consequenties heeft voor de inhoud van kolom B. Daarna hetzelfde gebeurt met betrekking tot kolom A, die als uitgangspunt was genomen.
Een eenvoudig CLP-programma zou hetzelfde doen. Het is hierbij niet nodig om de oplossingsruimte volledig uit te schrijven. Dat zou, gezien de omvang ervan, voor reële problemen ook onmogelijk zijn. Met behulp van regels kan echter hetzelfde effect worden bereikt.

Listing 1 toont in een CPL-achtige pseudotaal een programma voor het mutsenprobleem. Daarbij definieert de eerste regel het domein. De twee daarop volgende geven de beperkingen van de aantallen rode en witte mutsen. Tenslotte drukken regel 4 en 5 uit dat *A* zowel als *B* beide mogelijkheden moeten behouden. Aan het einde wordt met de functie *ask* opdracht gegeven om naar een waarde te zoeken voor de variabele *C*.

```
1 enum A, B, C = {r, w};
2 Num(w) ≤ 3;
3 Num(r) ≤ 2;
4 Card(A) = 2;
5 Card(B) = 2;
6 ask(C);
```

Listing 1. Pseudolisting van een CLP-programma voor het mutsenprobleem.

Zoekproces

De meest voor de hand liggende manier om naar een oplossing te zoeken is door de boom van mogelijkheden die in de regels is vastgelegd systematisch te doorzoeken. Het inferentiemechanisme heeft daarvoor een beginwaarde nodig. Hij zou bijvoorbeeld aan *C* de waarde *r* kunnen toekennen, om vervolgens te controleren of dit in strijd is met een van de gegeven randvoorwaarden van programmaregel 2 tot en met 5.
Dit is niet het geval, zodat de volgende variabele aan de beurt komt. Als voor *B* ook *r* wordt gekozen, valt er opnieuw geen strijdigheid te constateren en kan *A* de waarde *r* krijgen. Dit leidt echter tot een overschrijding van de begrenzing in regel 3.
A kan dus niet *r* zijn, maar als dat zo is dan volgt hieruit direct een tegenspraak met regel 4, die aangeeft dat beide mogelijkheden moeten kunnen gelden. Dat zou wel kunnen als B = w maar dat klopt niet met regel 5 waarin staat dat ook variabele *B* beide mogelijk waarden moet kunnen aannemen. De enig mogelijke conclusie is dat de beginaanname, C = r, moet worden verworpen.

Oplossingen

Als ervan wordt uitgegaan dat het programma correct is en dat er een geldige oplossing voor *C* bestaat dan is op het dit punt al mogelijk om te concluderen dat de variabele de enig overgebleven mogelijke waarde moet hebben: C = w. Zonder deze veronderstelling is het nodig om op dezelfde manier nog eens alle voorwaarden langs te lopen en hier de stelling C = w aan te toetsen. Dit zou uiteraard ook moeten gebeuren als met die aanname was begonnen in plaats van met het alternatief *r*.
Het is in principe natuurlijk denkbaar dat het probleem geen oplossing heeft dan wel foutief geformuleerd is. Evenzo zou het kunnen zijn dat er meerdere alternatieven voor *C* aan de gegeven voorwaarden voldoen. Het feit dat de puzzel precies één oplossing heeft is niet evident. Zo'n gegeven kan echter worden afgedwongen door een extra beperking aan het programma toe te voegen:

Card(C) = 1

In de situatie dat de andere regels meerdere waarden voor *C* toelaten zou aan deze extra voorwaarde niet voldaan zijn en zou het totaal zou dus niet tot een oplossing leiden.

Aanpassing

De hierboven geschetste methode voor het toetsen aan gegeven regels is niet bijster efficiënt. Beter is het om na elke uitgevoerde stap de voorwaarden aan te passen aan de nieuw ontstane toestand. Als immers voor *C* de waarde *r* is gekozen, blijft er nog maar één *r* over voor de andere variabelen. Regel 3 van de pseudolisting kan derhalve worden aangepast naar:

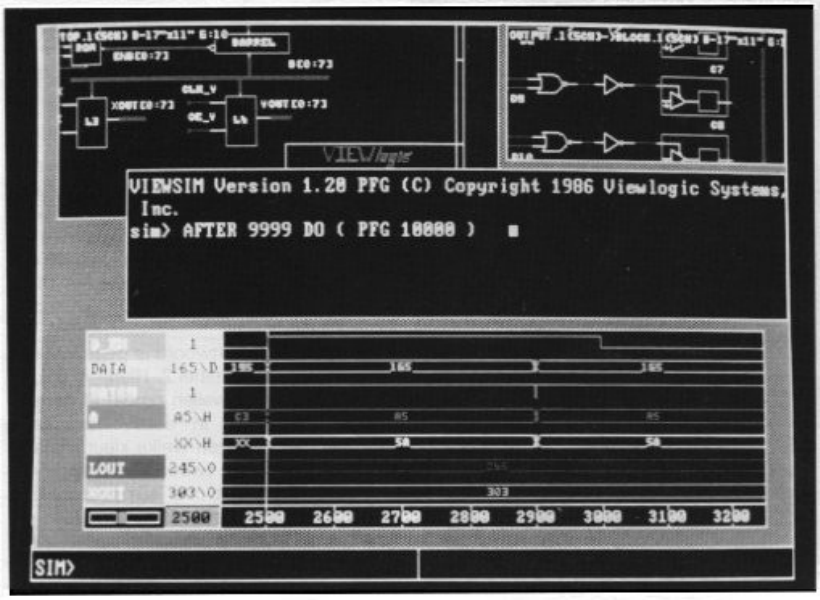
Num(r) ≤ 1

En op dezelfde manier ontstaat na het toekennen van *r* aan *B* een situatie die een tegenstrijdigheid bevat:

Num(r) = 0;
Card(A) = 2

Doordat nu al een conflict optreedt, kan de volgende stap – het kiezen van een waarde voor *A* – achterwege blijven. Het blijkt direct uit de regels dat de toekenning B = r niet is toegestaan. Deze beperking kan nu worden toegevoegd aan de be-

Ook het genereren van testpatronen en -programma's is een combinatie van een nagenoeg oneindige zoekruimte met veel randvoorwaarden die zich met CLP goed laat oplossen.



staande voorwaarden. Daaruit komt dan ook meteen weer een tegenstrijdigheid naar voren in combinatie met regel 5:

```
B not = r;
Card(B) = 2
```

Hieruit volgt dat de keuze C = r kan worden verworpen. En daarmee zijn we weer terug bij de beginsituatie, zij het met als toevoeging aan de voorwaarden dat *C* niet de waarde *r* mag hebben.

Optimalisatie

Uiteraard moet op die manier bij iedere volgende stap, dus ook bij terugredeneren, het geheel aan regels worden aangepast. Dit proces is nog verder te optimaliseren door de voorwaarden niet alleen te testen op strijdigheid maar ook nog op overbodigheid. Regel 1 van de pseudolisting is bijvoorbeeld redundant omdat het hierin aangegeven dit maximum al volgt uit het aantal variabelen. Een indicatie hiervoor is ook al te vinden in het redeneerproces, waar regel 1 geen enkele keer wordt gebruikt.
Dit snoeien in het programma kan bij elke verwerkingsstap gebeuren. Als bijvoorbeeld de eerste keuze van *C* de waarde *w* was geweest, bleven er voor *B* en *C* slechts twee waarden over. In die situatie is controle op het maximale aantal r-waarden niet meer nodig (regel 3).
In principe is het zelfs mogelijk om de puzzel nog sneller op te lossen. Als immers uit regels 4 en 5 direct wordt afgeleid dat het aantal beschikbare rode mutsen niet onder de twee mag komen zolang de variabelen *A* en *B* nog geen (voorlopige) waarde hebben gekregen dan zou daarmee de oplossing in één stap zijn bereikt. Het proces van toevoegen en weglaten van beperkingen na iedere berekeningstoestand wordt aangeduid met de term *constraint propagation* en vormt het eigenlijke grondprincipe van CPL's.

Zoekstrategie

Het is kenmerkend voor de klasse van oplossingsmethoden, die door het voorbeeld van de mutsen-puzzel enigszins wordt getypeerd, dat de benodigde zoektijd sterk afhankelijk kan zijn van het gekozen uitgangspunt. Een redeneeralgoritme dat was begonnen met de toekenning C = w – en dat is net zo willekeurig als de andere mogelijkheid – zou in één keer doorrekenen zijn geslaagd in zijn zoekpoging.
Van de beginaanname hangt bovendien ook vaak van af welke oplossing wordt gevonden, vooropgesteld dat er daarvan verschillende zijn. Elke uitslag die aan de voorwaarden voldoet, is immers correct.

Rekening houdend met deze kenmerken, hanteert het inferentiemechanisme doorgaans een zoekstrategie die het meest efficiënt tot een oplossing leidt. Een van de eenvoudigste principes daarbij is dat hij bij de kleinste domeinen begint. Mocht het probleem onoplosbaar zijn dan wordt dat op die manier eerder ontdekt dan wanneer de uitgangspunten uit grotere domeinen zijn geselecteerd.

De NS hebben in Charme een programma geïmplementeerd dat het probleem van de materieelomloop met succes aanpakt.

Talen

Het bijzondere aan deze vorm van programmeren is dat een oplossing kan worden gegenereerd zonder dat het nodig is om het probleem volledig te doorgronden. Dit is een algemeen kenmerk van declaratieve talen. Het is dan ook hierom dat CLP in regel wordt gerekend tot het terrein van de artificiële intelligentie.
Uiteraard is een relativering wel op zijn plaats; het formuleren van de verzameling randvoorwaarden in een CLP-programma vergt natuurlijk wel degelijk een diepgaande probleemanalyse. Vaak blijft het vraagstuk zelf echter enigszins ondoorzichtig. Zodra een probleem echt helemaal is ontrafeld, kan ook een gespecialiseerd algoritme worden geconstrueerd dat dan wellicht de voorkeur verdient boven enigszins omslachte redeneermethoden. Maar ook in zo'n situatie biedt het programmeren met randvoorwaarden de voordelen van kleine programma's en, daarmee samenhangend, korte ontwikkeltijden en goede onderhoudbaarheid.

Het wetenschappelijk onderzoek richt zich behalve op de inferentiemechanismen en het optimaal aanpassen van de regelverzameling aan de hand van de verkregen resultaten ook op het construeren en testen van programmeertalen. Een van de bekendste daarvan is Chip, wat staat voor *Constraint Handling in Prolog* (ref. 2). De taal cc(FD) is te beschouwen als een opvolger hiervan (ref. 1). Hiermee is ook een vraagstuk uit de digitale elektronica uitgewerkt, namelijk het genereren van testpatronen voor combinatorische schakelingen. Het verschil tussen Chip en cc(FD) is onder andere dat de laatste een expressie bevat voor voorwaardelijke beperkingen (*constraint implication*). Verder kent cc(FD) een operatie om aan te geven dat niet aan alle beperkingen uit een groep hoeft te worden voldaan maar slechts aan een bepaald aantal daarvan in een vrij te kiezen combinatie. Het principe achter deze zogeheten *cardinality combinator* is ook gehanteerd in het pseudoprogramma in listing 1.

Charme

Chip is ontwikkeld aan het ECRC (European Computer Industry Research Centre) in München. Het algemene AI-onderzoek bij deze instelling en Chip in het bijzonder vormen ook de basis geweest van Charme, de 'Constraint Programming Language' van ECRC-participant Bull. De oorspronkelijke band met logisch programmeren is in Charme voor wat betreft de syntaxis en se- ▷

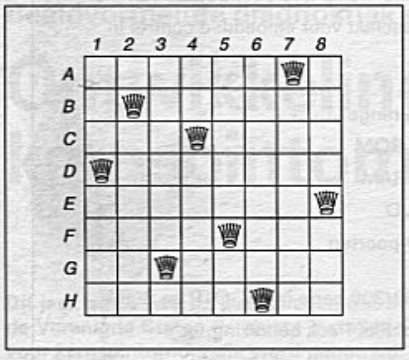


Fig. 1. Een van de oplossingen hoe N dames op een schaakbord met NxN velden te plaatsen zonder dat zij elkaar kunnen slaan.

```
{
/* one queen per line */
array Queens :: [1..8] of [1..8];
/* one queen per column */
all–diff(Queens);
/* not more than one queen on the same
diagonal */
for I in 1..7 do
for J in (I+1)..8 do
{
Queens[I]-Queens[J] != I-J;
Queens[I]-Queens[J] != J-I;
};
/* list solutions */
generate(Queens);
/* and display */
print(Queens);
}
```

Listing 2. Programma in Charme voor de schaakpuzzel uit figuur 1 (N = 8).

▷ mantiek grotendeels verlaten. Het is wel een declaratieve taal (ref. 3) maar het is zelf geschreven in C. Daardoor is het ook mogelijk om gewone C-functies te gebruiken in Charme en, andersom, om de taal aan te roepen vanuit een C-programma.

Het verschil tussen de syntaxis van Chip en die van Charme is duidelijk te zien in listing 2 en 3. Beide beschrijven het probleem hoe N dames op een schaakbord van N velden kunnen worden geplaatst zonder dat zij elkaar kunnen slaan. Een dame in het schaakspel kan, zoals bekend, horizontaal, verticaal en diagonaal over het hele bord bewegen. De acht stukken mogen derhalve in geen van deze richtingen op dezelfde lijn staan.

Schaakprobleem

Een van de mogelijke oplossingen is weergegeven in figuur 1. Het bijbehorende Charme-programma, zoals dat is uitgewerkt in *Introduction to Charme* (ref. 3), staat in listing 2. Hierin wordt eerst het domein gedefinieerd en vervolgens de verzameling beperkingen. De functie *generate* probeert het vraagstuk op te lossen maar dat proces blijft voor de programmeur verder verborgen.

Wat in feite gebeurt is hetzelfde als bij de mutsen-puzzel: het programma kiest posities en past na elke keuze de beperkingen aan. Als de gemaakte keuzen tot een onmogelijkheid leiden, vervangt het inferentiemechanisme via terugredeneren eerder gemaakte keuzen door alternatieven die niet eerder aan bod kwamen. Overigens kan, zo stellen de makers van Charme, de ontwikkelaar gebruik maken van een foutzoekprogramma met een stapfunctie omdat de onderliggende softwarelaag is opgebouwd met gewone C-routines.

In listing 3 staat de tegenhanger in Chip-code (ref. 5). Deze routine lost hetzelfde probleem op, zij het voor slechts vijf dames op een bord van vijf bij vijf velden. Wat hier in vergelijking met listing 2 vooral opvalt is de typische structuur van logisch programmeren met *goals, bodies* en recursieve operaties op lijsten.

Treinen

Charme is, in tegenstelling tot cc(FD), een commercieel produkt voor bedrijfsmatige toepassingen. Een van de gerealiseerde applicaties draait bij de Nederlandse Spoorwegen. Hier is de taal gebruikt om het logistieke probleem van de dagelijkse materieelinzet en de prognotisering van de materieelbehoefte op te lossen (ref. 5).

De moeilijkheid bij het inzetten van treinstellen is dat het passagiersaanbod gedurende de dag sterk varieert. Zo is er 's ochtends een piek richting grote steden. Dat betekent een behoefte aan veel treinstellen van de vertrekpunten naar de stedelijke centra maar niet andersom. In de andere richting komt de piek pas in de middagspits. Het probleem is meerledig, in die zin dat in de piekuren voldoende treinstellen op de juiste plaats moeten zijn, dat ze daarnaast ook de rest van de dag dienen te worden ingezet en dat er ondertussen zo min mogelijk lege treinstellen zouden moeten meerijden.

Deze planning van de materieelomloop bleek met conventionele technieken als operations research en procedurele benaderingen nauwelijks op te lossen. Dat kon alleen door het probleem sterk te reduceren, onder andere door het te beperken tot één treintype en wisselingen van samenstelling alleen op eindpunten toe te staan. In samenwerking met CVI en Bull is in relatief korte tijd een CPL-programma ontwikkeld dat die beperkingen niet heeft. Daarbij is de software voldoende flexibel om veranderingen en uitzonderingen eenvoudig op te vangen.

Listing 3. Geprogrammeerd in Chip (voor N = 5) ontstaat een geheel andere routine voor hetzelfde probleem.

```
domain five-queens(1..5).
five-queens([X1,X2,X3,X4,X5])
← safe([X1,X2,X3,X4,X5]),
   labeling([X1,X2,X3,X4,X5]).

safe([ ]).
safe(F | T)
← noattack(F,T),
   safe(T).

noattack(X,Xs)
← noattack(X,Xs,1).

noattack(X,[],Nb).
noattack(X,[Y | Ys],Nb)
← X ≠ Y,
   X ≠ Y – Nb,
   X ≠ Y + Nb,
   Nb1 is Nb + 1,
   noattack( X, Ys, Nb1).

labeling([ ]).
labeling([X | Y])
← indomain(X),
   labeling(Y).
```

[1.] P. van Hentenryck, H. Simonis, M. Dincbas: *Constraint Satisfaction Using Constraint Logic Programming*, Technical Report CS-91-62, Brown University, Providence, Verenigde Staten, november 1991.
[2.] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, F. Berthier: *The Constraint Logic Programming Language CHIP*, Proceedings of the International Conference on Fifth Generation Computer Systems ICOT, 1988.
[3.] *Artificial Intelligence, Introduction to Charme*, Bull, Parijs, 1990.
[4.] P. van Hentenryck: *A logic Language for Combinatorial Optimization*, Annals of Operations Research, nr. 21, 1989, pag. 247-274.
[5.] H. van Evert, E. Blaas, M. Blasband, R. Mooij: *Het omloopprobleem: plannen van de materieelinzet bij de Nederlandse Spoorwegen*, Proceedings van Kennistechnologie '92.