Artificiële Intelligentie
Project 2 - Constraint Processing

Philippe Dellaert
3^e Bachelor Computerwetenschappen - Elektrotechniek

13 mei 2007

1 Een logisch circuit

1.1 Variabelen en constraints

```
Name: In1, domain: (0..1)
Name: In2, domain: (0..1)
Name: In3, domain: (0..1)
Name: Tussen1, domain: (0..1)
Name: Tussen2, domain: (0..1)
Name: Uit, domain: (0..1)
Constraint: In1#=1
Constraint: In2#=0
Constraint: In3#=1
Constraint: min(In1,In2) #= Tussen1
Constraint: In3 #\= Tussen2
Constraint: max(Tussen1,Tussen2) #= Uit
```

1.2 Uitvoer

```
Backtracks: 0
Runtime: 0.0s
[In1,In2,In3,Tussen1,Tussen2,Uit] = [1,0,1,0,0,0]
```

1.3 Besluit

De bekomen uitvoer komt overeen met de verwachte waarde.

2 Geïndexeerde variabelen

2.1 Variabelen en constraints

Aangezien R2 niet gebruikt wordt, wordt er ook geen variabele voor aangemaakt. Ook kunnen J2 en K2 weggelaten worden aangezien deze toch altijd gelijk zijn aan Q1 en R1.

```
Name: In(i), i=1..3, domain: 0..1
Name: J1(i), i=1..3, domain: 0..1
Name: K1(i), i=1..3, domain: 0..1
Name: Q1(i), i=1..3, domain: 0..1
Name: R1(i), i=1..3, domain: 0..1
Name: Q2(i), i=1..3, domain: 0..1
Constraint: In(i)=1, range: i>0/\i<4
Constraint: J1(i)=In(i), range: i>0/\i<4
Constraint: K1(i)\=In(i), range: i>0/\i<4
Constraint: R1(i)\=Q1(i), range: i>0/\i<4
Constraint: J1(i)=0 /\ K1(i)=0 => Q1(i+1)=Q1(i), range: i>0/\i<3
Constraint: J1(i)\=K1(i) => Q1(i+1)=J1(i), range: i>0/\i<3
Constraint: J1(i)=1 /\ K1(i)=1 => Q1(i+1)\=Q1(i), range: i>0/\i<3
Constraint: Q1(i)=0 /\ R1(i)=0 => Q2(i+1)=Q2(i), range: i>0/\i<3
Constraint: Q1(i)\=R1(i) => Q2(i+1)=Q1(i), range: i>0/\i<3
Constraint: Q1(i)=1 /\ R1(i)=1 => Q2(i+1)\=Q2(i), range: i>0/\i<3
Constraint: Q1(i)=0, range: i=1
Constraint: Q2(i)=0, range: i=1
```

2.2 Uitvoer

```
Backtracks: 0
Runtime: 0.01s
[In,J1,K1,Q1,R1,Q2] = [[1,1,1],[1,1,1],[0,0,0],[0,1,1],[1,0,0],[0,0,1]]
```

3 Sudoku

3.1 Variabelen

We gebruiken maar één variabele, namelijk die die alle elementen voorsteld.

```
Name: EL(i), i=0..80, domain: 1..9
```

3.2 Constraints

Er zijn drie beperkingen bij het oplossen van een sudoku. Deze worden hieronder verder besproken en er wordt een constraint gekozen die kan gebruikt worden in het programma.

3.2.1 Rij beperking

Op een rij mogen geen twee dezelfde waarden voorkomen. Aangezien de deling door 9 aangeeft op welke rij het element zich bevindt, kunnen we dit makkelijk oplossen door de onderstaande constraint op te leggen.

```
Constraint: EL(i)\=EL(j), range: i/9=j/9 /\ i\=j
```

3.2.2 Kolom beperking

Op een kolom mogen geen twee dezelfde waarden voorkomen. Aangezien de modulo 9 bewerking aangeeft op welke kolom het element zich bevindt, kunnen we dit makkelijk oplossen door de onderstaande constraint op te leggen.

```
Constraint: EL(i)\=EL(j), range: i mod 9 = j mod 9 /\ i \= j
```

3.2.3 3x3 vierkant beperking

	Kolom 0			**Kolom 1**			**Kolom 2**		
Rij 0	0	1	2	3	4	5	6	7	8
	9	10	11	12	13	14	15	16	17
	18	19	20	21	22	23	24	25	26
Rij 1	27	28	29	30	31	32	33	34	35
	36	37	38	39	40	41	42	43	44
	45	46	47	48	49	50	51	52	53
Rij 2	54	55	56	57	58	59	60	61	62
	63	64	65	66	67	68	69	70	71
	72	73	74	75	76	77	78	79	80

Tabel 1: Voorstelling grote rijen en grote kolommen.

Het 9x9 veld van de sudoku wordt verdeeld in negen aparte 3x3 vierkanten. In een vierkant mogen geen twee dezelfde waarden voorkomen. Om hiervoor een constraint te vinden moeten we het speelveld op delen in drie grote rijen en drie grote kolommen, zoals aangegeven in tabel 1. Als we dit doen kunnen we makkelijk bepalen tot welke grote rij een element behoort door te delen door 27. Om te bepalen wat de grote kolom is waartoe een element behoort, voeren we eerst de modulo 9 bewerking uit en delen dan door 3. Deze twee samen geven dan aan tot welk vierkant het element behoort. Zo bekomen we de onderstaande constraint.

```
Constraint: EL(i)\=EL(j), range: i/27=j/27 /\ (i mod 9)/3=(j mod 9)/3 /\ i \= j
```

3.3 Oplossen via de `leftmost` methode

3.3.1 Resultaten

Eerst bekijken we het oplossen van de verschillende sudoku's waar we gebruik maken van de `leftmost` methode en de bovenstaande constraints. In tabel 2 staan vier waarden voor elke sudoku, namelijk de totale tijd die er nodig was om het op te lossen, het totaal aantal backtracks, het aantal oplossingen en de maximaal aantal backtracks voor één oplossing. Deze laatste waarde stelt dus het grootste aantal backtracks die is waargenomen om één oplossing te vinden voor het probleem.

	Totale tijd (s)	Totaal # backtracks	# Oplossingen	Maximaal # backtracks
Easy 1	0.11	0	1	0
Easy 2	0.11	0	1	0
Easy 3	0.11	0	1	0
Hard 1	0.14	88	37	33
Hard 2	0.10	35	1	35
Hard 3	0.10	8	1	8

Tabel 2: Resultaten bij het verwerken met de `leftmost` methode.

3.3.2 Besluit

Het valt direct op dat bij de makkelijke sudoku's er veel minder backtracks moeten gebeuren dan bij de moeilijke sudoku's. Dit heeft te maken met de complexiteit van het probleem. Hoe meer waarden al vast staan, hoe minder vrijheidsgraden er moeten opgelost worden en hoe minder foute veronderstellingen er gemaakt kunnen worden. Dit zorgt dus voor minder backtracking.

3.4 Oplossen via de `first-fail+constraint count` methode

3.4.1 Resultaten

	Totale tijd (s)	Totaal # backtracks	# Oplossingen	Maximaal # backtracks
Easy 1	0.10	0	1	0
Easy 2	0.10	0	1	0
Easy 3	0.11	0	1	0
Hard 1	0.13	37	37	10
Hard 2	0.09	11	1	11
Hard 3	0.11	3	1	3

Tabel 3: Resultaten bij het verwerken met de `first-fail+constraint count` methode.

3.4.2 Vergelijking met de `leftmost` methode

In tabel 3 kunnen we de resultaten van de `first-fail+constraint count` methode bekijken voor de verschillende sudoku's. Als we deze tabel vergelijken met die van de `leftmost` methode (tabel 2), valt direct op dat het aantal backtracks sterk gedaald is voor de `first-fail+constraint count` methode. Dit heeft te maken met de manier waarop het probleem wordt aangepakt. Bij de `leftmost` methode wordt de volgorde van oplossen van variabelen bepaald door de volgorde waarin de variabelen gedefinïeerd zijn. Dit heeft als gevolg dat niet altijd de beste keuze gemaakt wordt. Bij de `first-fail+constraint count` wordt de volgende variabele gekozen aan de hand van het domein. De variabele met het kleinste domein wordt eerst opgelost, waardoor de domeinen van de overige

variabelen ook kan verkleinen als ze zicn bevoorbeeld in dezelde rij, dezelfde kolom of hetzelfde vierkant bevinden.

3.5 *"Two out of three"*-regel

3.5.1 Extra constraints

Bij het toepassen van de *"Two out of three"*-regel gaat men kijken of een getal al in twee rijen van een grote rij (cfr. tabel 1) aanwezig is. Dit betekent ook dat dit getal al in twee verschillende 3x3 vierkanten voorkomt in die grote rij. Daaruit kan men besluiten dat dit getal in het overgebleven 3x3 vierkant enkel op drie plaatsen kan staan, namelijk de drie elementen in het 3x3 vierkant die zich bevinden op de kleine rij die het getal nog niet bevat.

.	a	.	.	.	.	.	.	.
.	.	.	a?	a?	a?	.	.	.
.	.	.	.	.	.	.	.	a

Tabel 4: Voorstelling van de *"Two out of three"*-regel in één grote rij.

In tabel 4 wordt een grafische voorstelling gegeven van de *"Two out of three"*-regel. Het tweede element op de eerste rij is gelijk aan het getal a, evenals het negende element op de derde rij. Dit betekent dus dat in de bovenste en de onderste rij het getal a niet meer mag voorkomen. Daarnaast mag het getal a ook niet meer voorkomen in het linkse en het rechtse 3x3 vierkant. Daardoor wordt de positie van het getal a in het middelste vierkant beperkt tot de drie blauw gekleurde vakjes.

Om deze strategie in te bouwen in ons programma, hebben we drie constraints nodig. Voor elke van de drie mogelijke posities van het getal. De range die we nodig hebben is voor elke constraint hetzelfde op één term na die de positie van het het gezochte element bepaald. Deze range bestaat zelf uit drie delen:

- Als eerste moeten we bepalen dat de drie elementen in één grote rij zitten. Dit kunnen we doen door gebruik te maken van de eerder bepaalde regel dat delen door 27 bepaald in welke grote rij een element zich bevindt.

  ```
  i/27 = j/27 /\ j/27 = k/27
  ```

- Daarnaast moeten we ook bepalen dat de drie elementen in verschillende grote kolommen zitten. Dit geeft, in combinatie met het vorige deel, aan dat de drie elementen zin in drie verschillende 3x3 vierkanten bevinden. Ook hier kunnen we gebruik maken van de eerder bepaalde regel dat door de modulo 9 bewerking op het element en dan te delen door 3 we de grote kolom krijgen van het element.

  ```
  (i mod 9)/3 \= (j mod 9)/3
  /\ (j mod 9)/3 \= (k mod 9)/3
  /\ (i mod 9)/3 \= (k mod 9)/3
  ```

- Als laatste moeten we ook bepalen dat de drie elementen zich op drie verschillende rijen bevinden. Ook dit hebben we al eens geïmplementeerd.

  ```
  i/9 \= j/9 /\ j/9 \= k/9 /\ i/9 \= k/9
  ```

Als we deze drie delen samen nemen krijgen we de volgende basis range voor onze constraints.

```
basis range: i/27 = j/27 /\ j/27 = k/27
/\ (i mod 9)/3 \= (j mod 9)/3
/\ (j mod 9)/3 \= (k mod 9)/3
/\ (i mod 9)/3 \= (k mod 9)/3
/\ i/9 \= j/9 /\ j/9 \= k/9 /\ i/9 \= k/9
```

Om de constraints zelf te bepalen moeten we drie verschillende gevallen bekijken. Deze drie verschillende gevallen worden bepaald door welk element van de drie juist gezocht wordt. Dit zal er ook voor zorgen dat we uiteindelijk drie verschillende constraints zullen moeten toevoegen.

3.5.2 Het linkse element

.	$EL(i) = a$	.	.	.	.	.	.	.
.	.	.	$EL(k) = a$	$EL(k+1)$	$EL(k+2)$	.	.	.
.	.	.	.	.	.	.	.	$EL(j) = a$

Tabel 5: Situatie waarin het gezochte element ($EL(k)$) het linkse element is.

Uit tabel 5 blijkt duidelijk dat we moeten bepalen of het gezochte element zich bevindt in het linkse vakje. Hiervoor moeten we de volgende term toevoegen aan onze range.

```
range: basis /\ k mod 3 = 0
```

Om dan te bepalen dat element k effectief gelijk is aan het getal a, moeten we eerst bepalen dat element k+1 en element k+2 niet gelijk zijn aan a. Hiervoor kunnen we de volgende constraint gebruiken.

```
Constraint: EL(i) = EL(j) /\ EL(k+1) \= EL(i) /\ EL(k+2) \= EL(i)
    => EL(i) = EL(k)
```

3.5.3 Het middelste element

.	$EL(i) = a$	.	.	.	.	.	.	.
.	.	.	$EL(k-1)$	$EL(k) = a$	$EL(k+1)$	.	.	.
.	.	.	.	.	.	.	.	$EL(j) = a$

Tabel 6: Situatie waarin het gezochte element ($EL(k)$) het middelste element is.

Een soortgelijke redenering als bij het linkse element en tabel 6 levert de volgende aanpassing op de basis range en de tweede constraint

```
range: basis /\ k mod 3 = 1
Constraint: EL(i) = EL(j) /\ EL(k-1) \= EL(i) /\ EL(k+1) \= EL(i)
    => EL(i) = EL(k)
```

3.5.4 Het rechtse element

.	$EL(i) = a$	.	.	.	.	.	.	.
.	.	.	$EL(k-2)$	$EL(k-1)$	$EL(k) = a$	.	.	.
.	.	.	.	.	.	.	.	$EL(j) = a$

Tabel 7: Situatie waarin het gezochte element ($EL(k)$) het rechtse element is.

Een soortgelijke redenering als bij het linkse en het middelste element en tabel 7 levert de volgende aanpassing op de basis range en de tweede constraint

```
range: basis /\ k mod 3 = 2
Constraint: EL(i) = EL(j) /\ EL(k-2) \= EL(i) /\ EL(k-1) \= EL(i)
    => EL(i) = EL(k)
```

3.5.5 Uiteindelijk toegevoegde constraints

```
Constraint: EL(i) = EL(j) /\ EL(k+1) \= EL(i) /\ EL(k+2) \= EL(i)
    => EL(i) = EL(k)
range: i/27 = j/27 /\ j/27 = k/27
    /\ (i mod 9)/3 \= (j mod 9)/3
    /\ (j mod 9)/3 \= (k mod 9)/3
    /\ (i mod 9)/3 \= (k mod 9)/3
    /\ i/9 \= j/9 /\ j/9 \= k/9 /\ i/9 \= k/9
    /\ k mod 3 = 0

Constraint: EL(i) = EL(j) /\ EL(k-1) \= EL(i) /\ EL(k+1) \= EL(i)
    => EL(i) = EL(k)
range: i/27 = j/27 /\ j/27 = k/27
    /\ (i mod 9)/3 \= (j mod 9)/3
    /\ (j mod 9)/3 \= (k mod 9)/3
    /\ (i mod 9)/3 \= (k mod 9)/3
    /\ i/9 \= j/9 /\ j/9 \= k/9 /\ i/9 \= k/9
    /\ k mod 3 = 1

Constraint: EL(i) = EL(j) /\ EL(k-2) \= EL(i) /\ EL(k-1) \= EL(i)
    => EL(i) = EL(k)
range: i/27 = j/27 /\ j/27 = k/27
    /\ (i mod 9)/3 \= (j mod 9)/3
    /\ (j mod 9)/3 \= (k mod 9)/3
    /\ (i mod 9)/3 \= (k mod 9)/3
    /\ i/9 \= j/9 /\ j/9 \= k/9 /\ i/9 \= k/9
    /\ k mod 3 = 2
```

3.6 Oplossen via de `leftmost` methode

3.6.1 Resultaten

	Totale tijd (s)	Totaal # backtracks	# Oplossingen	Maximaal # backtracks
Easy 1	16.62	0	1	0
Easy 2	17.10	0	1	0
Easy 3	16.91	0	1	0
Hard 1	17.20	27	37	11
Hard 2	16.85	3	1	3
Hard 3	16.96	3	1	3

Tabel 8: Resultaten bij het verwerken met de `leftmost` methode met de *"Two out of three"*-regel.

3.6.2 Waarneming

Wat direct opvalt uit tabel 8 is dat de tijd die nodig is om het probleem op te lossen veel groter is dan wanneer we niet gebruik maken van de *"Two out of three"*-regel. Daarnaast valt het wel op dat het aantal backtracks gedaalt is. Dit is zeer duidelijk voor de eerste moeilijke sudoku. Daar is het aantal backtracks gedaalt van 88 tot 27.

3.7 Oplossen via de `first-fail+constraint count` methode

3.7.1 Resultaten

	Totale tijd (s)	Totaal # backtracks	# Oplossingen	Maximaal # backtracks
Easy 1	16.68	0	1	0
Easy 2	16.64	0	1	0
Easy 3	16.64	0	1	0
Hard 1	17.22	23	37	12
Hard 2	17.00	7	1	7
Hard 3	17.17	6	1	6

Tabel 9: Resultaten bij het verwerken met de `first-fail+constraint count` methode met de *"Two out of three"*-regel.

3.7.2 Waarneming

Ook hier valt het duidelijk op uit tabel 9 dat de tijden spectaculair zijn toegenomen. En ook hier kunnen we zien dat het aantal backtracks gedaald is, al geldt dit niet voor elke sudoku.

3.8 Besluit

Het eerste dat duidelijk opvalt uit de resultaten, is de spectaculaire stijging van de benodigde verwerkingstijd. Dit heeft alles te maken met de ingewikkelde toegevoegde constraints. Hierdoor moeten veel meer controle stappen doorlopen worden, die het proces zwaar vertragen. Tussen de `first-fail+constraint count` methode en de `leftmost` methode zelf is er weinig verschil te merken in de verwerkingstijd.

Bij het aantal benodigde backtracks is voor beide methodes een zelfde daling waar te nemen. Deze daling is ook te verklaren door de extra constraints. Doordat er meer constraints zijn, zullen de mogelijke waarden in elk element ook mee beperkt worden, waardoor er minder fouten voorkomen.

Uit het gevolg op de verwerkingstijd blijkt dat het niet interessant is om de *"Two out of three"*-regel te implementeren voor het oplossen van sudoku's op de pc. Dit komt voornamelijk door de drie eerste constraints die het probleem al zodanig definieren dat het programma er al genoeg aan heeft. Als er nog meer ingewikkelde constraints worden toegevoegd, zal dit zorgen voor vertragingen aangezien die ook nog eens, onnodig, verwerkt moeten worden.