Practicumverslag Computationele Intelligentie

Cornelis Bouter (studentnummer) en Roel Brouwer (3976866)

27 juni 2014

Chapter 1

Implementatie algoritmen

1.1 Hill-climbing

1.1.1 Algemeen

Representatie en bepalen begintoestand

Sudoku's zijn gerepresenteerd als een dubbel array ($n^2 \times n^2$) van cijfers (1 t/m n^2). In dit hele verslag wordt met n de wortel van de breedte (of hoogte) van de puzzel bedoeld. Voor een puzzel van 9 bij 9 geldt dus, bijvoorbeeld $n = 3$. We beschouwen dus alleen vierkante sudoku's. Probleemtoestanden zijn volledig ingevulde sudoku's, wat betekent dat er nog iets met de input moet gebeuren om een begintoestand te verkrijgen.

De lege vakjes in de sudoku worden gevuld door een willekeurig getal dat nog niet in het $n \times n$ blok voorkomt. Welk getal in het vakje terecht komt wordt bepaald door de volgende functie:

```
ZoekVulling(AlGevuld)
    // AlGevuld is een boolean array, met voor elk getal dat
    // in een n bij n blok moet voorkomen een positie.
    // AantalOngevuld is een variabele waarin het aantal nog
    // ongevulde vakjes is opgeslagen.
    TeVullen = (int)(randomFloat(0, 1) * AantalOngevuld);
    count = 0;
    // Tot is exclusief.
    for i = 0 tot n * n
        if (!AlGevuld[i])
            if(count == TeVullen)
                return i + 1;
            else
                count++;
```

Het resultaat is een volledig gevulde begintoestand.

Evaluatie

Het is nodig om toestanden te beoordelen, zodat ze vergeleken kunnen worden en we daadwerkelijk kunnen klimmen.

De onderstaande evaluatiefunctie doet dat door het aantal missende cijfers in alle rijen en kolommen bij elkaar op te tellen:

```
Evalueer (Sudoku)
    TeReturnen = 0;
    contains = boolean[n * n];
    // Telt alle ontbrekende cijfers in de rijen op.
    for i = 0 tot n * n
        for k = 0 tot n * n
            contains[k] = false;
        for j = 0 tot n * n
            contains[Sudoku[i][j] - 1] = true;
        for k = 0 tot n * n
            if(!contains[k])
                TeReturnen++;
    // Telt alle ontbrekende cijfers in de kolommen op.
    for i = 0 tot n * n
        for k = 0 tot n * n
            contains[k] = false;
        for j = 0 tot n * n
            contains[Sudoku[j][i] - 1] = true;
        for k = 0 tot n * n
            if(!contains[k])
                TeReturnen++;
    return TeReturnen;
```

Zoekoperator

Het bepalen van de volgende toestand in het *hill-climbing* gebeurt aan de hand van de zoekoperator (hieronder in pseudo-code uitgewerkt). Deze geeft een nieuwe sudoku terug, die van de oude verschilt op twee vakjes: de eerste verwisseling die een verbetering opleverde in de evaluatiewaarde van de sudoku.

```
ZoekOperator (Sudoku)
    // eval bevat de waarde die de evaluatiefunctie aan de sudoku geeft.
    eval = Sudoku.evalueer();
    // Verwisselen gaat per nxn-blok. g en h representeren het blok. Het zijn constantes die bij
    // de variabelen die de positie van de vakjes binnen het blok representeren (i, j, k en l)
    // worden opgeteld om het juiste vakje te kunnen identificeren. Tot is exclusief. In de
    // volgende geneste loops worden alle wissel-mogelijkheden afgelopen tot er een gevonden is
    // die de huidige toestand verbetert.
    for g = 0 tot n*n (ophogen per 3)
        for h = 0 tot n*n (ophogen per 3)
            for i = 0 tot n
                for j = 0 tot n
                    for k = 0 tot n
                        for l = 0 tot n
                            if ((i,j) en (k,l) representeren niet hetzelfde vakje EN
                                (i,j) en (k,l) zijn variabele vakjes)
                                // Verkrijg de nieuwe sudoku door de vakjes te verwisselen.
                                Sudoku.verwissel(g + i, h + j, g + k, h + l);
                                // Als de nieuwe beter is zijn we klaar.
                                if (Sudoku.evalueer() > eval)
                                    return Sudoku;
                                // Anders maken we de verwisseling ongedaan.
                                else
                                    Sudoku.verwissel(g + i, h + j, g + k, h + l);
    // Als er niks verwisseld kan worden, zitten we in een lokaal
    // optimum. We geven niets terug.
    return null;
```

Een lokaal optimum (of plateau) is gevonden op het moment dat de zoekoperator niets teruggeeft: alle

successor-toestanden zijn dan immers slechter dan of even goed als de huidige toestand.

Het is noodzakelijk dat deze zoekoperator-functie systematisch alle mogelijk successors langsgaat. Op die manier kunnen we er zeker van zijn dat geen van de successor-toestanden meer een verbetering is, en weten we dat we in een lokaal optimum (of plateau) zitten.

1.1.2 Random Restart

Met het klaarstaan van de zoekoperator die in de vorige sectie besproken is, is het implementeren van *random restart* relatief eenvoudig. In pseudo-code ziet dat er als volgt uit:

```
RandomRestart (HuidigeSudoku, Input)
    // Input is de originele input waaruit de begintoestand gehaald is.
    // Totdat een globaal optimum is gevonden, blijven we zoeken.
    while (Evalueer(HuidigeSudoku) > 0)
        NieuweSudoku = ZoekOperator(HuidigeSudoku);
        // Als een lokaal optimum wordt bereikt, herstarten.
        if (NieuweSudoku == null)
            // Maak een nieuwe begintoestand en begin opnieuw.
            HuidigeSudoku = Sudoku(n, Input);
        else
            // Anders gaan we door met de gevonden toestand.
            HuidigeSudoku = NieuweSudoku;
    // Een globaal optimum is gevonden.
    Print(HuidigeSudoku);
```

1.1.3 Iterated Local Search

Zoekoperator

Voor het implementeren van *iterated local search* is het nodig geweest de zoekoperator iets aan te passen, of beter gezegd: een variant op de zoekfunctie toe te voegen.

De zoekoperator die voor het beklimmen van de heuvel, het *hill-climbing* deel, gebruikt wordt, loopt alle swap-mogelijkheden in een bepaalde volgorde door, en kiest de eerste die een verbetering is. Voor *iterated local search* is een *random walk* nodig. Een *random walk* geeft niets om het antwoord op de vraag of een opvolger-toestand al dan niet een verbetering is. Het moet een willekeurige opvolger-toestand kunnen kiezen, ongeacht de waarde die de evaluatiefunctie aan die toestand geeft.

Wanneer de oude implementatie gebruikt zou worden, zou het dan steeds voor de eerste optie kiezen: het swappen van de eerste twee niet-vaste vakjes in het blok in de linksbovenhoek van de sudoku. Dat willen we niet, daarom veranderen we de zoekoperator-functie zodat deze twee willekeurige vakjes (binnen hetzelfde blok - natuurlijk) van plaats verwisseld worden:

```
RandomZoekOperator (Sudoku)
    // g en h representeren het blok waarin het verwisselen wordt uitgevoerd.
    g, h = randomInt(0, n) * n;
    // i, j, k en l representeren de twee vakjes die verwisseld worden.
    i, j, k, l = randomInt(0, n);
    while ((i,j) en (k,l) representeren hetzelfde vakje OF
      (i,j) of (k,l) is een vast vakje)
        // Nieuwe waardes voor i, j, k en l.
        i, j, k, l = randomInt(0, n);
    // Verwissel de genoemde vakjes in de sudoku en return het resultaat
    return Sudoku.verwissel(g + i, h + j, g + k, h + l);
```

Zoek algoritme

Gebruik makend van de hierboven beschreven zoekoperator(s) (zowel de willekeurige als de deterministische versie), is de implementatie van *iterated local search* als volgt:

```
IteratedLocalSearch (HuidigeSudoku)
    // Totdat een globaal optimum is gevonden, blijven we zoeken.
    while (Evalueer(HuidigeSudoku) > 0)
        NieuweSudoku = ZoekOperator(HuidigeSudoku);
        // Als een lokaal optimum wordt bereikt, daar weglopen.
        if (NieuweSudoku == null)
            // Maak een random walk van S lang, S kan gevarieerd worden.
            for i = 0 tot S
                HuidigeSudoku = RandomZoekOperator(HuidigeSudoku);
        else
            // Anders gaan we door met de gevonden toestand.
            HuidigeSudoku = NieuweSudoku;
    // Een globaal optimum is gevonden.
    Print(HuidigeSudoku);
```

1.2 Constraint Satisfaction

1.2.1 Algemeen

Representatie

Een *Constraint Satisfaction Problem* (CSP) bestaat uit drie componenten: de variabelen (X), de domeinen (D) en de constraints (C). Hieronder beschrijven we hoe we elk van deze componenten hebben gerepresenteerd in onze code.

- **Variabelen (X).** De variabelen zijn vakjes in de sudoku. Deze vakjes zijn gerepresenteerd als objecten met een waarde en een domein en methoden om informatie over deze twee zaken op te vragen of ze te manipuleren. "Vaste vakjes" zijn gerepresenteerd door vanaf het begin maar n element in het domein te stoppen.
- **Domeinen (D).** De domeinen zitten ingebakken in de objecten die hierboven beschreven zijn, aangezien de relatie variabele-domein in principe 1 op 1 is.
- **Constraints (C).** Constraints zijn ook gerepresenteerd als objecten. We hebben ervoor gekozen om alleen binaire constraints te implementeren, aangezien alle hogere constraints tot binaire constraints te herleiden zijn. Een constraint is een object dat twee vakjes "bij zich heeft" en methoden bevat om te kijken welke vakjes dat zijn en of het constraint geschonden is. Aangezien in dit probleem alleen constraints van de vorm $X \neq Y$ voorkomen is in die laatste methode ingebakken dat getest wordt of de waarde van het ene vakje ongelijk is aan de waarde van het andere vakje.

Het probleem (de sudoku) is gerepresenteerd als een lijst van variabelen en een lijst van constraints.

Elke variabele heeft dan $2(n^2 - 1) + (n - 1)^2$ constraints. Er zitten $n^2 - 1$ andere vakjes in dezelfde rij, en evenveel in dezelfde kolom. Er zitten daarbij nog n^2 vakjes in hetzelfde blok, daarvan moeten we het vakje zelf (1) en de vakjes die al in de rij/kolom zitten (2(n - 1)) nog afhalen. Dan komen we uit op $(n^2 - (1 + 2(n - 1))) = (n - 1)^2$ extra vakjes.

Hieronder laten we zien hoe het probleem gerepresenteerd wordt, door een (deel van de) constructor van het object dat het probleem representeert te tonen.

```
Sudoku(Input)
    // Input is de originele input waaruit de begintoestand gehaald is.
    X = Variabelen[n * n][n * n];
    // l en k voor de blokken, i en j identificerren de vakjes in de blokken.
    for l = 0 tot n
        for k = 0 tot n
            for i = 0 tot n
                for j = 0 tot n
                    waarde = Input[l * n + i][k * n + j];
                    // Als het een leeg vakje is, maken we een variabelen aan met een volledig
                        domein.
                    if (waarde == 0)
                        X[l * n + i][k * n + j] = Variabelen(0, VolDomein);
                    // Anders is het een gevuld vakje, en bevat het domein maar een element.
                    else
                        X[l * n + i][k * n + j] = Variabelen(waarde, EnkelDomein);

    Constraints = Constraint[n * n][n * n][2 * (n * n - 1) + ((n - 1) * (n - 1))];
    // Er worden constraints aangemaakt voor alle mogelijke combinaties van twee vakjes die in
    // dezelfde rij, kolom, of hetzelfde blok zitten. Deze worden bij de juiste vakjes in het
    // array gestopt, zodat aan het eind van het verhaal elk blokje een lijst heeft van de
    // constraints die voor hem gelden. De precieze uitwerking daarvan is voor dit verslag
    // niet relevant en kan in de code worden teruggevonden.
```

Consistency check

Het is belangrijk om na elke stap in het zoeken naar de oplossing na te gaan of met het nemen van deze stap geen constraints geschonden worden.

...
ToDo - als implementatie constraintscheck compleet is

1.2.2 Backtracking

1.2.3 Forward Checking

Chapter 2

Experimentele resultaten

2.1 Hill-climbing

2.1.1 Algemene observaties

Efficiëntie

(Rekentijd en aantal toestanden tot lokaal optimum)

Accepteren gelijke toestanden

2.1.2 Random Restart

Efficiëntie

(Rekentijd en aantal toestanden tot globaal optimum)

2.1.3 Iterated Local Search

Efficiëntie

(Rekentijd en aantal toestanden tot globaal optimum)

2.2 Constraint Satisfaction

2.2.1 Backtracking

2.2.2 Forward Checking