# Practicumverslag Computationele Intelligentie

Cornelis Bouter (studentnummer) en Roel Brouwer (3976866)

27 juni 2014

# Hoofdstuk 1

# Korte gebruikershandleiding

Om het programma te gebruiken, volgt u het volgende stappenplan:

- Pak de zip-file uit op een voor u handige locatie, vanaf nu PATH genoemd;
- Open de console en navigeer naar PATH\src;
- Compileer het programma met het volgende commando: *javac Puzzel.java*;
- Start het programma met *java Puzzel*, met als eerste (en enige) argument de file waarin de sudoku opgeslagen is die u wil oplossen. U vindt 15 verschillende puzzels in PATH, u zou als command line argument bijvoorbeeld *PATH\test_n3_1_ci.txt* kunnen meegeven;
- Het programma start en vraagt u welke manier van zoeken u wil gebruiken om een oplossing te vinden:
  - Typ *r* voor Random Restart Hill-Climbing;
  - Typ *i* voor Iterated Local Search;
  - Typ *b* voor Constraint Satisfaction - Backtracking zonder sorteren;
  - Typ *s* voor Constraint Satisfaction - Backtracking met (eenmalig) sorteren;
  - Typ *f* voor Constraint Satisfaction - Backtracking met forward checking.
  - Typ *m* voor Constraint Satisfaction - Backtracking met forward checking en minimum-remaining-values.
- Het programma zal nu een oplossing voor u gaan zoeken en u informatie over deze oplossing geven.

Een run van het programma zou er zo uit kunnen zien:

```
C:\>       cd PATH\src
PATH\src>  javac Puzzel.java
PATH\src>  java Puzzel PATH\test_n3_1_ci.txt

Voor Random Restart Hill-Climbing, voer "r" in.
Voor Iterated Local Search, voer "i" in.
Voor CSP - Backtracking, voer "b" in.
Voor CSP - Backtracking (eenmalig gesorteerd), voer "s" in.
Voor CSP - Forward checking, voer "f" in.
Voor CSP - Forward checking (MRV), voer "m" in.
r
<PROGRAM OUTPUT>
```

# Hoofdstuk 2

# Implementatie algoritmen

## 2.1 Hill-climbing

### 2.1.1 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:

```
1  ZoekVulling(AlGevuld)
2      // AlGevuld is een boolean array, met voor elk getal dat
3      // in een n bij n blok moet voorkomen een positie.
4      // AantalOngevuld is een variabele waarin het aantal nog
5      // ongevulde vakjes is opgeslagen.
6      TeVullen = (int)(randomFloat(0, 1) * AantalOngevuld);
7      count = 0;
8      // Tot is exclusief.
9      for i = 0 tot n * n
10         if (!AlGevuld[i])
11             if(count == TeVullen)
12                 return i + 1;
13             else
14                 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.

### 2.1.2 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);
```

### 2.1.3 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)
            // De vorige stap was het lokale optimum, die terughalen.
            NieuweSudoku = HuidigeSudoku;
            while (NieuweSudoku.Evalueer() <= HuidigeSudoku.Evalueer())
                // Als het nieuwe optimum niet beter was, keren we terug.
                NieuweSudoku = HuidigeSudoku;
                // Maak een random walk van S lang, S kan gevarieerd worden.
                for i = 0 tot S
                    NieuweSudoku = RandomZoekOperator(NieuweSudoku);
            HuidigeSudoku = NieuweSudoku;
        else
            // Anders gaan we door met de gevonden toestand.
            HuidigeSudoku = NieuweSudoku;
    // Een globaal optimum is gevonden.
    Print(HuidigeSudoku);
```

## 2.2 Constraint Satisfaction

### 2.2.1 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.

Gegeven dat een bepaald vakje gewijzigd is, wordt met behulp van onderstaande methode gecontroleerd of er geen constraints geschonden worden:

```
Consistent (X, Y)
    // X en Y zijn de "cordinaten" van het betreffende vakje in de array.
    for i = 0 tot Constraints[X][Y].length
        // Als een van de constraints geschonden wordt, moet de methode false teruggeven.
        if (Constraints[X][Y][i].ConstraintGeschonden())
            return false;
    return true;
```

### Bepalen initiële vrijheidsgraad

Om het probleem goed te representeren, moeten de domeinen van de verschillende variabelen aan het begin op de goede manier gedefinieerd zijn. Met andere woorden: de waardes die bij voorbaat nooit ingevuld kunnen worden in die vakjes moeten uit hun domein worden gehaald.

Dit lossen we op door voor elk vakje alle constraints een keer door te lopen. Als het andere vakje dat meedoet aan de constraint een ingevuld vakje is, dan moeten we de waarde van dat vakje uit ons domein verwijderen. In pseudocode ziet dat er als volgt uit:

```
BepalenInitieleVrijheidsgraad()
    // Voor elk vakje lopen we alle constraints af die voor het vakje gelden. In de dubbele
    // array X zitten alle variabelen opgeslagen, in Constraints de constraints (per variabele).
    for i = 0 tot X.length
        for j = 0 tot X[i].length
            for c = 0 tot Constraints[i][j].length
                // We kijken naar het vakje in de constraint dat niet X[i][j] is.
                if (Constraints[i][j][c].V1 == X[i][j])
                    // Als het andere vakje dat meedoet aan het constraint een ingevuld vakje is,
                    // verwijderen we de waarde van dat vakje uit het domein van X[i][j].
                    if (Constraints[i][j][c].V2.domeinGrootte == 1)
                        X[i][j].DomeinElementVerwijderen(Constraints[i][j][c].V2.waarde);
                else
                    if (Constraints[i][j][c].V1.domeinGrootte == 1)
                        X[i][j].DomeinElementVerwijderen(Constraints[i][j][c].V1.waarde);
```

### 2.2.2 Backtracking

Gegeven de hierboven beschreven representatie van CSP is het relatief eenvoudig om backtracking te implementeren. Dit gebeurt recursief, waarbij steeds vakjes worden ingevuld, tot het mis gaat. Op dat moment wordt de laatste keuze herzien, totdat alle opties "op" zijn, of er een oplossing gevonden is. De onderstaande pseudocode laat zien hoe wij dit geïmplementeerd hebben.

Er wordt aan enkele hulpmethoden gerefereerd, waarvan geen pseudocode beschikbaar is in dit verslag. Dit zijn vrij eenvoudige methoden, waarvan de implementatie voor zich spreekt. Voor de precieze implementatie verwijzen we naar de code.

```
Backtracking (Sudoku)
    Sudoku.BepalenInitieleVrijheidsgraad();
    NieuweSudoku = BacktrackingRecursief(Sudoku);
    if (NieuweSudoku != null)
        Print(NieuweSudoku);
    else
        Print("Geen oplossing.");

BacktrackingRecursief (Sudoku)
    // Als de sudoku volledig ingevuld is, zijn we klaar in de recursie
    if (Sudoku.VolledigIngevuld())
        return Sudoku;
    // We pakken het eerste vakje (variabele) dat nog niet is ingevuld.
    Vakje = VindVolgende(Sudoku);
    for i = 0 tot Vakje.domein.length
        // Als i in het domein zit, gaan we eens proberen om Vakje in te vullen met i + 1. We zetten
        // de waarde op i + 1 (en verkleinen ook het domein zodat het alleen nog i + 1 bevat).
        if (Vakje.ElementInDomein(i + 1))
            OudDomein = Vakje.domein;
            Vakje.waarde = i + 1;
            // Als er geen constraints geschonden worden, gaan we een niveau dieper de recursie in.
            if (Sudoku.Consistent(Vakje.X, Vakje.Y))
                NieuweSudoku = BacktrackingRecursief(Sudoku);
                if (NieuweSudoku != null) return NieuweSudoku;
            // Als we hier terecht komen is het blijkbaar mislukt en kan i + 1 niet in Vakje worden
            // ingevuld. We maken het vakje dus weer leeg en zetten het domein netjes terug.
            Vakje.waarde = 0;
            Vakje.domein = OudDomein;
    // Er kan niets voor dit vakje worden ingevuld, dus de puzzel is onoplosbaar:
    return null;
```

**Toevoegen van sortering**

Voordat we een gesorteerde lijst van vakjes kunnen gaan gebruiken voor de backtracking, moeten we een methode hebben die een lijst van vakjes kan sorteren.

De methode die dit eenmalige sorteren voor ons doet ziet er, in pseudocode, als volgt uit:

```
1  EenmaligSorteren ()
2      // Sla eerst de tweedimensionale array van variabelen plat
3      for i = 0 tot X.length
4          for j = 0 tot X[i].length
5              Gesorteerd[i * X[0].length + j] = X[i][j];
6      // Sorteer de array, met als comparator een functie die de domeingroottes vergelijkt.
7      Arrays.sort(Gesorteerd, Comparator() {
8              Compare(V1, V2)
9                  return V1.domeinGrootte - V2.domeinGrootte;
10         });
```

Het is dan een kleine stap om dit in het bestaande backtracking algoritme toe te voegen. Aan de pseudocode op de vorige pagina moet het volgende gebeuren:

- Tussen regel 2 en 3 moet het volgende worden toegevoegd:

```
1  Sudoku.EenmaligSorteren();
```

- Regel 13 en 14 moeten worden vervangen door:

```
1  // We pakken het volgende vakje uit de gesorteerde lijst die we met EenmaligSorteren() hebben
2  // aangemaakt.
3  Vakje = VindVolgendeGesorteerd(Sudoku);
```

### 2.2.3 Forward Checking

De implementatie van forward checking lijkt heel sterk op die van normale backtracking. We hebben één extra methode nodig, een methode die na het invullen van een vakje de domeinen van de vakjes die worden beïnvloed aanpast. Dat doet *PasDomeinAan*. Deze methode checkt gelijk of een domein door de verandering niet leeg wordt, en returnt *false* als dit ergens wel gebeurt.

```
1  PasDomeinAan (I, J)
2      for int c = 0 tot Constraints[I][J].length
3          // We kijken steeds naar het andere vakje, het vakje dat niet gelijk is aan X[I][J].
4          // Herinner dat X een dubbele array is met de variabelen
5          if (Constraints[I][J][c].V1 == X[I][J])
6              // Verwijder de waarde van X[I][J] uit het domein van V2
7              Constraints[I][J][c].V2.DomeinElementVerwijderen(X[I][J].waarde);
8              // Als het domein nu leeg wordt, moeten we stoppen.
9              if (Constraints[I][J][c].V2.DomeinGrootte() == 0) return false;
10         else
11             // Symmetrisch aan het bovenstaande
12             Constraints[I][J][c].V1.DomeinElementVerwijderen(X[I][J].waarde);
13             // Als het domein nu leeg wordt, moeten we stoppen.
14             if (Constraints[I][J][c].V1.DomeinGrootte() == 0) return false;
15     // Als we hier aankomen is alles goed gegaan.
16     return true;
```

Met deze methode in de hand ziet de (pseudo-)code voor backtracking met forward checking er als volgt uit:

```
ForwardChecking (Sudoku)
    Sudoku.BepalenInitieleVrijheidsgraad();
    Nieuwesudoku = ForwardCheckingRecursief(Sudoku);
    if (NieuweSudoku != null) {
        Print(NieuweSudoku);
    else
        Print("Geen oplossing.");

ForwardCheckingRecursief (Sudoku sudoku)
    // Als de sudoku volledig ingevuld is, zijn we klaar in de recursie
    if (Sudoku.VolledigIngevuld())
        return Sudoku;
    // We pakken het eerste vakje dat nog niet is gevuld.
    Vakje = VindVolgende(sudoku);
    for i = 0 tot Vakje.domein.length
        // Als i in het domein zit, gaan we eens proberen om Vakje in te vullen met i + 1. We
        // zetten de waarde op i + 1 (en verkleinen ook het domein zodat het alleen nog i + 1
        // bevat).
        if (Vakje.ElementInDomein(i + 1))
            OudDomein = Vakje.domein;
            Vakje.waarde = i + 1;
            // Als er geen constraints geschonden worden, gaan we een niveau dieper de recursie in.
            if (Sudoku.Consistent(Vakje.X, Vakje.Y))
                // We kopieren de sudoku, en passen in de nieuwe sudoku de domeinen aan
                ExtraSudoku = Sudoku;
                if (ExtraSudoku.PasDomeinAan(Vakje.X, Vakje.Y))
                    NieuweSudoku = ForwardCheckingRecursief(ExtraSudoku);
                    if(NieuweSudoku != null) return NieuweSudoku;
                Vakje.waarde = 0;
                Vakje.domein = OudDomein;
    return null;
```

### Toevoegen van sortering

We hebben al code om eenmalig te sorteren, nu hebben we nog een methode nodig die iedere keer opnieuw sorteert:

```
OpnieuwSorteren()
    // Sorteer de array met vakjes, met als comparator een functie die de domeingroottes
    // vergelijkt.
    Arrays.sort(Gesorteerd, Comparator() {
        Compare(V1, V2)
            return V1.domeinGrootte - V2.domeinGrootte;
    });
```

Het is dan een kleine stap om dit in het bestaande forward checking algoritme toe te voegen. Aan de pseudocode hierboven moet het volgende gebeuren:

- Tussen regel 2 en 3 moet het volgende worden toegevoegd:

```
Sudoku.EenmaligSorteren();
```

- Regel 13 en 14 moeten worden vervangen door:

```
// We sorteren de lijst met vakjes opnieuw...
Sudoku.OpnieuwSorteren();
// ...en pakken het volgende vakje uit de lijst
Vakje = VindVolgendeGesorteerd(Sudoku);
```

# Hoofdstuk 3

# Experimentele resultaten

## 3.1 Hill-climbing

### 3.1.1 Algemene observaties

#### Efficiëntie

De tijd die het algoritme nodig heeft om een lokaal maximum te bereiken laat zich opvallend goed middellen. De gemiddelden van het aantal stappen dat nodig is van 10-15 lokale maxima liggen heel dicht bij elkaar - zolang het om dezelfde puzzel gaat. Tussen de puzzels is meer variatie te zien. Het lijkt erop dat het landschap van bepaalde puzzels rijker bezaaid is met lokale maxima dan dat van andere.

Voor puzzels waar $n = 3$ ligt dit aantal in onze tests meestal rond de 7, en komt vrijwel nooit boven de 30.

Voor puzzels waar $n = 4$ ligt het aantal stappen dat nodig is om een lokaal maximum te bereiken meestal tussen de 30 en de 40, en bij $n = 5$ tussen de 70 en de 90.

Voor het vinden van een lokaal maximum is verwaarloosbaar veel tijd nodig. Als dat je doel is, krijg je bij het stellen van de vraag direct een antwoord. Wij zijn echter meer genteresseerd in globale maxima, oplossingen voor de puzzel. Om die te bereiken is meer nodig dan hill-climbing alleen. Hieronder bespreken we de efficiëntie van de twee mogelijkheden die wij geïmplementeerd hebben.

#### Accepteren gelijke toestanden

Bij de implementatie van hill-climbing kan ervoor gekozen worden om bij het zoeken naar een opvolgertoestand toestanden die niet strikt een verbetering zijn ten opzichte van de huidige toestand te accepteren als opvolger. Dit kan bereikt worden door de $>$ in regel 20 van de pseudocode voor de ZoekOperator (p. 2) te veranderen in een $\geq$.

In onze implementatie is dat een groot nadeel, omdat daarmee de mogelijkheid wordt geschapen tot het oneindig in cirkeltjes lopen - dat wordt zelfs erg waarschijnlijk. Op het moment dat de eerste swap die een opvolgertoestand oplevert die gelijk/beter is dan de huidige toestand een toestand oplevert die even goed is als de huidige toestand, zal bij de volgende stap de omgekeerde operatie (het terugswappen van de net verwisselde vakjes) de eerste swap zijn die gelijk/beter is dan de toestand. Omdat de mogelijke verwisselingen steeds in dezelfde volgorde worden beschouwd, betekent dit een oneindig heen-en-weer springen tussen twee toestanden, zonder ooit bij een lokaal maximum aan te komen.

Wanneer we dus kiezen voor $>$ in plaats van $\geq$ hebben we meer kans op een (snel) antwoord. Het is dus efficiënter en veiliger om daarvoor te kiezen.

### 3.1.2 Random Restart

Voor een uitgebreid(er) overzicht van de test-resultaten, zie *appendix A*.

#### Efficiëntie

De lokale maxima hebben we hierboven al besproken - het bereiken van lokale maxima vanuit een willekeurige begintoestand gaat snel. Maar dat is niet wat we willen.

Het is een puur kansspel - een oplossing wordt gevonden op het moment dat de willekeurige begintoestand in het attractiegebied van het/een globaal optimum blijkt te liggen. Tot die tijd blijft het algoritme bergjes beklimmen om er vervolgens achter te komen dat de top van het bergje geen oplossing is. Het is mogelijk dat de eerste begintoestand al direct tot de oplossing leidt, er is ook een kans dat het optimum niet in eindige tijd gevonden wordt (als er toevallig nooit een begintoestand in het attractiegebied van het globale optimum wordt gegenereerd). Meestal echter moet er enkele honderden of duizenden keren opnieuw gestart worden om een oplossing te vinden.

Hoe groot de kans is dat het globale optimum gevonden wordt verschilt ook per puzzel, bij de ene puzzel lijken er minder optima te zijn dan bij de andere en/of heeft het globale optimum een groter attractiegebied.

Voor puzzels met $n > 3$ wordt er zeer zelden binnen 15 minuten een oplossing gevonden. Voor puzzels met $n = 3$ kan dit nog wel. Met een gemiddelde van 25 stappen nodig voor het vinden van een lokaal optimum, lijkt het landschap voor puzzel 1 iets minder optima te bevatten dan die van puzzel 2 t/m 5 (gemiddeld zo'n 7 stappen nodig).

Bij puzzel 1 wordt een oplossing gemiddeld na zo'n 16 seconden gevonden (iets meer dan 100.000 stappen en 4000 herstarts). Terwijl de puzzels 2 t/m 4 binnen een kwartier nog niet met een antwoord komen. Het lijkt erop dat het globale maximum van puzzel 1 een relatief groot attractiegebeid heeft.

Dat dit niet één op één samenhangt met het feit dat er minder optima zijn in puzzel 1, bewijst puzzel 5. Bij puzzel 5 zijn gemiddeld 6 stappen nodig voor het bereiken van een lokaal optimum. Toch wordt een globaal optimum gemiddeld binnen 4 seconden (iets meer dan 30.000 stappen, 1700 optima) gevonden. De laatste klim, naar het globale optimum, is dan ook gemiddeld 31 stappen lang, in plaats van zes. Dit laat zien dat het attractiegebied van het globale optimum groter is dan dat van de meeste lokale maxima, en niet kleiner dan dat van puzzel 1.

Alles bij elkaar hangt het dus sterk af van de vorm van het landschap hoe efficiënt deze zoekmethode is. Bepaalde sudoku's kunnen behoorlijk snel worden opgelost, zolang het globale optimum een redelijk groot attractiegebied heeft. Anders is handmatig oplossen waarschijnlijk sneller.

### 3.1.3 Iterated Local Search

Voor een uitgebreid(er) overzicht van de test-resultaten, zie *appendix B*.

#### Efficiëntie

(Rekentijd en aantal toestanden tot globaal optimum)

## 3.2 Constraint Satisfaction

### 3.2.1 Backtracking

#### Efficiëntie

#### Effect van sorteren

### 3.2.2 Forward Checking

#### Efficiëntie

#### Effect van sorteren

# Hoofdstuk 4

# Reflectie

## 4.1 Implementatie algoritmen

Niet alle algoritmen waren even gemakkelijk te implementeren. De volgorde waarin ze in dit verslag behandeld worden is er een van oplopende moeilijkheid.

Het opzetten van de representatie van het probleem kostte het meeste tijd in het implementeren van de hill-climbing algoritme. Random restart en iterated local search waren relatief eenvoudig te implementeren toen de basis eenmaal stond.

Voor Constraint Satisfaction geldt ongeveer hetzelfde. Het opzetten van de representatie bleek het lastigste. Het implementeren van backtracking was iets eenvoudiger, en het toevoegen van de optie tot sorteren was snel gedaan. Backtracking met forward checking implementeren kostte weer iets meer tijd, maar ook hier was het toevoegen van de optie tot sorteren snel gebeurd. Beide algoritmen waren duidelijk lastiger te implementeren dan de hill-climbing algoritmen. Zo was ook de basis van hill-climbing eenvoudiger te implementeren dan die van Constraint Satisfaction.

## 4.2 Experimenten

...TODO... Cornelis

# Appendices - introductie

Voor het testen van de verschillende algoritmen hebben we vijf puzzels van elke soort ($n = 3, 4, 5$) gebruikt. En hiervan is afkomstig van de website van het vak, de andere vier komen van *menneske.no*. De puzzels zijn te vinden in de txt-bestanden met de namen test_n$x$_$nr$_$bron$.txt, waarbij $x \in \{3, 4, 5\}, nr \in \{1, 2, 3, 4, 5\}$ en $bron \in \{ci, menneske\}$.

De testresultaten zijn uitgesplitst naar het soort puzzel (de $n$-waarde) en de specifieke puzzel waarop getest is. In de paragraaf bij de puzzel wordt vervolgens precies aangegeven wat getest is.

# Appendix A

# Testresultaten Random Restart Hill-climbing

## A.1 9 * 9 puzzels (n = 3)

Voor elke puzzel in deze sectie zijn 15 testruns uitgevoerd. Van deze runs wordt het volgende weergeven:

- Lokale maxima. Het algoritme bereikt eerst een aantal lokale maxima voor het globale maximum bereikt wordt. Van de eerste 15 lokale maxima die bereikt zijn wordt hier weergeven:
  - Het gemiddeld aantal stappen dat het kostte om vanaf de begintoestand naar het lokale maximum te komen;
  - De gemiddelde waarde van de evaluatiefunctie voor deze lokale maxima (m.a.w. hoeveel cijfers staan er gemiddeld nog verkeerd?).
- Globaal maximum. Uiteindelijke bereikt het algoritme meestal een globaal maximum (als dat niet binnen 15 minuten gebeurd is, bevatten de kolommen over het globale maximum geen informatie). Van het globale maximum wordt weergeven:
  - Het aantal stappen dat gezet is voor het globale maximum bereikt was;
  - De tijd (in milliseconden) die het kostte het globale maximum te bereiken;
  - Hoeveel lokale maxima langsgekomen zijn voordat het globale maximum gevonden was (= hoeveel keer een random restart heeft plaatsgevonden);
  - Hoeveel stappen er vanaf de laatste herstart gezet zijn om het globale maximum te bereiken.

### A.1.1 Puzzel 1 (test_n3_1_ci)

| | Lokale maxima | | Globaal maximum | | | |
|---|---|---|---|---|---|---|
| Testnr. | stappen | evaluatie | stappen | tijd (ms) | lokale optima | laatste klim |
| 1 | 24 | 12 | 83801 | 12661 | 3243 | 31 |
| 2 | 24 | 9,9 | 28602 | 4376 | 1114 | 31 |
| 3 | 25 | 8,9 | 79208 | 11866 | 3085 | 44 |
| 4 | 24 | 9,3 | 88237 | 13550 | 3447 | 31 |
| 5 | 25 | 10,9 | 236198 | 35407 | 9245 | 33 |
| 6 | 26 | 8,8 | 203321 | 30898 | 7909 | 31 |
| 7 | 26 | 10,2 | 74913 | 11450 | 2920 | 32 |
| 8 | 26 | 11,1 | 25339 | 3873 | 975 | 26 |
| 9 | 24 | 9,7 | 168835 | 25533 | 6597 | 29 |
| 10 | 24 | 11,2 | 24687 | 3768 | 954 | 21 |
| 11 | 26 | 10,3 | 63134 | 9521 | 2456 | 26 |
| 12 | 26 | 9,5 | 218689 | 32873 | 8529 | 28 |
| 13 | 26 | 9,8 | 87454 | 13186 | 3405 | 30 |
| 14 | 25 | 10,2 | 126969 | 18006 | 4947 | 25 |
| 15 | 25 | 10,2 | 51102 | 7718 | 1977 | 37 |
| Gemiddeld: | 25,1 | 10,1 | 104032,6 | 15645,7 | 4053,5 | 30,3 |

### A.1.2 Puzzel 2 (test_n3_2_menneske)

| | Lokale maxima | | Globaal maximum | | | |
|---|---|---|---|---|---|---|
| Testnr. | stappen | evaluatie | stappen | tijd (ms) | lokale optima | laatste klim |
| 1 | 8 | 15 | *Werd niet binnen 15 minuten gevonden* | | | |
| 2 | 8 | 16 | | | | |
| 3 | 7 | 15 | | | | |
| 4 | 10 | 15 | | | | |
| 5 | 7 | 16 | | | | |
| 6 | 8 | 15 | | | | |
| 7 | 6 | 17 | | | | |
| 8 | 7 | 16 | | | | |
| 9 | 8 | 15 | | | | |
| 10 | 7 | 18 | | | | |
| 11 | 8 | 15 | | | | |
| 12 | 7 | 17 | | | | |
| 13 | 8 | 15 | | | | |
| 14 | 8 | 16 | | | | |
| 15 | 8 | 15 | | | | |
| Gemiddeld: | 7,7 | 15,7 | | | | |

### A.1.3 Puzzel 3 (test_n3_3_menneske)

| | Lokale maxima | | Globaal maximum | | | |
|---|---|---|---|---|---|---|
| Testnr. | stappen | evaluatie | stappen | tijd (ms) | lokale optima | laatste klim |
| 1 | 7 | 17 | *Werd niet binnen 15 minuten gevonden* | | | |
| 2 | 7 | 17 | | | | |
| 3 | 7 | 17 | | | | |
| 4 | 7 | 18 | | | | |
| 5 | 8 | 16 | | | | |
| 6 | 7 | 16 | | | | |
| 7 | 7 | 16 | | | | |
| 8 | 7 | 16 | | | | |
| 9 | 9 | 17 | | | | |
| 10 | 8 | 17 | | | | |
| 11 | 7 | 17 | | | | |
| 12 | 8 | 17 | | | | |
| 13 | 7 | 18 | | | | |
| 14 | 7 | 17 | | | | |
| 15 | 7 | 17 | | | | |
| Gemiddeld: | 7,3 | 16,9 | | | | |

### A.1.4 Puzzel 4 (test_n3_4_menneske)

| | Lokale maxima | | Globaal maximum | | | |
|---|---|---|---|---|---|---|
| Testnr. | stappen | evaluatie | stappen | tijd (ms) | lokale optima | laatste klim |
| 1 | 7 | 16 | *Werd niet binnen 15 minuten gevonden* | | | |
| 2 | 7 | 14 | | | | |
| 3 | 7 | 15 | | | | |
| 4 | 6 | 16 | | | | |
| 5 | 8 | 15 | | | | |
| 6 | 7 | 15 | | | | |
| 7 | 7 | 14 | | | | |
| 8 | 7 | 16 | | | | |
| 9 | 7 | 16 | | | | |
| 10 | 8 | 15 | | | | |
| 11 | 7 | 15 | | | | |
| 12 | 7 | 15 | | | | |
| 13 | 6 | 16 | | | | |
| 14 | 7 | 15 | | | | |
| 15 | 7 | 16 | | | | |
| Gemiddeld: | 7 | 15,3 | | | | |

### A.1.5 Puzzel 5 (test_n3_5_menneske)

| | Lokale maxima | | Globaal maximum | | | |
|---|---|---|---|---|---|---|
| Testnr. | stappen | evaluatie | stappen | tijd (ms) | lokale optima | laatste klim |
| 1 | 6 | 16 | 2017 | 280 | 86 | 32 |
| 2 | 7 | 15 | 10128 | 1235 | 433 | 32 |
| 3 | 6 | 14 | 13049 | 1566 | 555 | 34 |
| 4 | 7 | 16 | 60000 | 6989 | 2566 | 32 |
| 5 | 7 | 15 | 11866 | 1406 | 509 | 35 |
| 6 | 6 | 16 | 60588 | 7048 | 2602 | 25 |
| 7 | 6 | 14 | 9328 | 1146 | 391 | 27 |
| 8 | 7 | 15 | 5366 | 673 | 226 | 28 |
| 9 | 6 | 18 | 9808 | 1184 | 412 | 28 |
| 10 | 6 | 16 | 49993 | 5839 | 2146 | 34 |
| 11 | 5 | 16 | 15863 | 1933 | 6683 | 27 |
| 12 | 5 | 16 | 60006 | 6895 | 2564 | 30 |
| 13 | 6 | 17 | 64164 | 7435 | 2753 | 29 |
| 14 | 6 | 15 | 33467 | 3862 | 1421 | 32 |
| 15 | 6 | 14 | 83975 | 9668 | 3592 | 37 |
| Gemiddeld: | 6,1 | 15,5 | 32641,2 | 3810,6 | 1795,9 | 30,8 |

## A.2 16 * 16 puzzels (n = 4)

Voor elke puzzel in deze sectie zijn 5 testruns uitgevoerd. Bij geen enkele van de puzzels werd binnen 15 minuten een oplossing bereikt. Er wordt dus alleen maar data weergegeven van de eerste 10 gevonden lokale maxima:

- Het gemiddeld aantal stappen dat het kostte om vanaf de begintoestand naar het lokale maximum te komen;
- De gemiddelde waarde van de evaluatiefunctie voor deze lokale maxima (m.a.w. hoeveel cijfers staan er gemiddeld nog verkeerd?).

### A.2.1 Puzzel 1 (test_n4_1_ci)

| Testnr. | stappen | evaluatie |
|---|---|---|
| 1 | 31 | 101 |
| 2 | 30 | 100 |
| 3 | 32 | 99 |
| 4 | 28 | 102 |
| 5 | 29 | 101 |
| Gemiddeld: | 30 | 100,6 |

### A.2.2 Puzzel 2 (test_n4_2_menneske)

| Testnr. | stappen | evaluatie |
|---|---|---|
| 1 | 35 | 83 |
| 2 | 35 | 87 |
| 3 | 35 | 89 |
| 4 | 36 | 85 |
| 5 | 38 | 86 |
| Gemiddeld: | 35,8 | 86 |

### A.2.3 Puzzel 3 (test_n4_3_menneske)

| Testnr. | stappen | evaluatie |
|---|---|---|
| 1 | 36 | 89 |
| 2 | 39 | 81 |
| 3 | 38 | 84 |
| 4 | 33 | 87 |
| 5 | 35 | 89 |
| Gemiddeld: | 36,2 | 86 |

### A.2.4 Puzzel 4 (test_n4_4_menneske)

| Testnr. | stappen | evaluatie |
|---|---|---|
| 1 | 36 | 95 |
| 2 | 35 | 91 |
| 3 | 36 | 89 |
| 4 | 39 | 91 |
| 5 | 36 | 89 |
| Gemiddeld: | 36,4 | 91 |

### A.2.5 Puzzel 5 (test_n4_5_menneske)

| Testnr. | stappen | evaluatie |
|---|---|---|
| 1 | 29 | 91 |
| 2 | 32 | 88 |
| 3 | 32 | 87 |
| 4 | 29 | 89 |
| 5 | 33 | 87 |
| Gemiddeld: | 31 | 88,4 |

## A.3 25 * 25 puzzels (n = 5)

Voor de eerste twee puzzels in deze sectie zijn 5 testruns uitgevoerd, voor de laatste drie zijn dat er 3. Bij geen enkele van de puzzels werd binnen 15 minuten een oplossing bereikt. Er wordt dus alleen maar data weergegeven van de eerste 10 gevonden lokale maxima:

- Het gemiddeld aantal stappen dat het kostte om vanaf de begintoestand naar het lokale maximum te komen;
- De gemiddelde waarde van de evaluatiefunctie voor deze lokale maxima (m.a.w. hoeveel cijfers staan er gemiddeld nog verkeerd?).

### A.3.1 Puzzel 1 (test_n5_1_ci)

| Testnr. | stappen | evaluatie |
|---|---|---|
| 1 | 72 | 269 |
| 2 | 67 | 260 |
| 3 | 70 | 270 |
| 4 | 72 | 265 |
| 5 | 69 | 268 |
| Gemiddeld: | 70 | 266,4 |

### A.3.2 Puzzel 2 (test_n5_2_menneske)

| Testnr. | stappen | evaluatie |
|---|---|---|
| 1 | 82 | 210 |
| 2 | 90 | 202 |
| 3 | 83 | 212 |
| 4 | 82 | 205 |
| 5 | 87 | 199 |
| Gemiddeld: | 84,8 | 205,6 |

### A.3.3 Puzzel 3 (test_n5_3_menneske)

| Testnr. | stappen | evaluatie |
|---|---|---|
| 1 | 87 | 219 |
| 2 | 90 | 220 |
| 3 | 83 | 227 |
| Gemiddeld: | 86,7 | 222 |

### A.3.4 Puzzel 4 (test_n5_4_menneske)

| Testnr. | stappen | evaluatie |
|---|---|---|
| 1 | 85 | 215 |
| 2 | 81 | 219 |
| 3 | 85 | 220 |
| Gemiddeld: | 83,7 | 218 |

### A.3.5 Puzzel 5 (test_n5_5_menneske)

| Testnr. | stappen | evaluatie |
|---|---|---|
| 1 | 83 | 226 |
| 2 | 87 | 225 |
| 3 | 84 | 220 |
| Gemiddeld: | 84,7 | 223,7 |

# Appendix B

# Testresultaten Iterated Local Search

Het vinden van lokale maxima is uitgebreid getest bij de tests voor random restart. Voor iterated local search verandert hier weinig aan. Alle hieronder staande data heeft dus betrekking op het vinden van globale optima.

## B.1 9 * 9 puzzels (n = 3)

### B.1.1 Puzzel 1 (test_n3_1_ci)

### B.1.2 Puzzel 2 (test_n3_2_menneske)

### B.1.3 Puzzel 3 (test_n3_3_menneske)

### B.1.4 Puzzel 4 (test_n3_4_menneske)

### B.1.5 Puzzel 5 (test_n3_5_menneske)

## B.2 16 * 16 puzzels (n = 4)

### B.2.1 Puzzel 1 (test_n4_1_ci)

### B.2.2 Puzzel 2 (test_n4_2_menneske)

### B.2.3 Puzzel 3 (test_n4_3_menneske)

### B.2.4 Puzzel 4 (test_n4_4_menneske)

### B.2.5 Puzzel 5 (test_n4_5_menneske)

## B.3 25 * 25 puzzels (n = 5)

### B.3.1 Puzzel 1 (test_n5_1_ci)

### B.3.2 Puzzel 2 (test_n5_2_menneske)

### B.3.3 Puzzel 3 (test_n5_3_menneske)

### B.3.4 Puzzel 4 (test_n5_4_menneske)

### B.3.5 Puzzel 5 (test_n5_5_menneske)

# Appendix C

# Testresultaten Constraint Satisfaction - backtracking

## C.1 9 * 9 puzzels (n = 3)

### C.1.1 Puzzel 1 (test_n3_1_ci)

### C.1.2 Puzzel 2 (test_n3_2_menneske)

### C.1.3 Puzzel 3 (test_n3_3_menneske)

### C.1.4 Puzzel 4 (test_n3_4_menneske)

### C.1.5 Puzzel 5 (test_n3_5_menneske)

## C.2 16 * 16 puzzels (n = 4)

### C.2.1 Puzzel 1 (test_n4_1_ci)

### C.2.2 Puzzel 2 (test_n4_2_menneske)

### C.2.3 Puzzel 3 (test_n4_3_menneske)

### C.2.4 Puzzel 4 (test_n4_4_menneske)

### C.2.5 Puzzel 5 (test_n4_5_menneske)

## C.3 25 * 25 puzzels (n = 5)

### C.3.1 Puzzel 1 (test_n5_1_ci)

### C.3.2 Puzzel 2 (test_n5_2_menneske)

### C.3.3 Puzzel 3 (test_n5_3_menneske)

### C.3.4 Puzzel 4 (test_n5_4_menneske)

### C.3.5 Puzzel 5 (test_n5_5_menneske)

# Appendix D

# Testresultaten Constraint Satisfaction - forward checking

## D.1 9 * 9 puzzels (n = 3)

### D.1.1 Puzzel 1 (test_n3_1_ci)

### D.1.2 Puzzel 2 (test_n3_2_menneske)

### D.1.3 Puzzel 3 (test_n3_3_menneske)

### D.1.4 Puzzel 4 (test_n3_4_menneske)

### D.1.5 Puzzel 5 (test_n3_5_menneske)

## D.2 16 * 16 puzzels (n = 4)

### D.2.1 Puzzel 1 (test_n4_1_ci)

### D.2.2 Puzzel 2 (test_n4_2_menneske)

### D.2.3 Puzzel 3 (test_n4_3_menneske)

### D.2.4 Puzzel 4 (test_n4_4_menneske)

### D.2.5 Puzzel 5 (test_n4_5_menneske)

## D.3 25 * 25 puzzels (n = 5)

### D.3.1 Puzzel 1 (test_n5_1_ci)

### D.3.2 Puzzel 2 (test_n5_2_menneske)

### D.3.3 Puzzel 3 (test_n5_3_menneske)

### D.3.4 Puzzel 4 (test_n5_4_menneske)

### D.3.5 Puzzel 5 (test_n5_5_menneske)