Verslag CI Backtracking Sudoku

Pieter de Marez Oyens (5720508)

dr. ir. D. Thierens

13 - Juni - 2017

Inhoudsopgave

Representatie van de sudoku puzzel

Datastructuur

Een sudoku puzzel bestaat uit enkele onderdelen. Om de uitleg te veralgemeniseren zal hier een 9 bij 9 puzzel worden besproken, maar omdat de te oplossen sudoku puzzels allemaal vierkant zijn gelden dezelfde principes ook voor de grotere puzzels. Eerst zal kort de opbouw van een puzzels in zijn algemeen besproken worden, en vervolgens zal worden uitgelegd hoe elk onderdeel in code wordt gerepresenteerd.

De structuur van een puzzel bestaat uit een veld, coördinaten op het veld en de getallen die ingevuld zijn of kunnen worden op elk coördinaat punt.

In code

Het Veld: Grid.cs

Het skelet van elke puzzel, in de code *Grid* genoemd, is gebaseerd dit artikel: http://norvig.com/sudoku.html, geschreven door Peter Norvig.

In de code is dit een statische die op basis van de grote van een bord (de hoogte of breedte) de coördinaten aanmaakt van elk vlak.

Het idee achter deze coördinaten is als volgt: een sudoku bord bestaat uit rijen en kolommen. Elke rij krijg een letter van A tot I en elke kolom krijgt een cijfer van 1 tot 9. Deze worden gecombineerd door middel van een cartesisch product waardoor door vlakken/coördinaten A1 (links boven) tot I9 (rechts onder) ontstaan.

Hieruit volgen de twee belangrijkste elementen van deze klasse en deze structuur.

- *Squares:* Deze verzameling vlakken heet in code *squares* en is een *statische array* van strings. Als er een methode is die over alle vlakken van een bord moet heen *loopen*, dan is deze array beschikbaar voor elk bord. Deze lijst is ook zeer belangrijk voor de *Board* klasse, hierover later meer.
- *Peers:* Dit zijn alle vlakken waarbij, volgens sudoku regels, één uniek vlak mee gemoeid is. Dat wil zeggen: alle vakken die op dezelfde rij en kolom staan, en in hetzelfde vierkant zitten, exclusief het vlak zelf. Elk vak heeft als zodanig 20 buren. Bijvoorbeeld, A1's buren: A2, A3, A4, A5, A6, A7, A8, A9 (horizontaal) | B1, C1, D1, E1, F1, G1 H1 I1 (verticaal) |B2, B3, C2, C3 (zelfde vierkant). De peers zullen nogmaals voor elke puzzel hetzelfde zijn en is daarom *statisch.* De structuur waarin deze peers worden opgeslagen is een *Dictionary* met als *KeyValuePair<string, HashSet<string>>*. Elke key is een uniek vlak op het veld, en het paar zijn al zijn buren (peers). Er is gekozen voor een dictionairy omdat, als we bij de peers van een vlak moeten wezen, we al het vlak weten, dus er is geen reden om over een list of array te loopen. Het feit dat er voor de value een hashset is gekozen was tijdens het creeëren van de structuur om ervoor te zorgen dat er geen dubbele waarden zouden voorkomen, maar is nu in principe overbodig. Er was echter geen tijd om dit te refactoren naar bijvoorbeeld een lijst of array.

EenPuzzel: Board.cs

Dit is de datastructuur voor een sudoku puzzel die doorzocht en opgelost kan worden. De puzzel is een *Dictionairy* met als *KeyValuePair<string, List<int>>.* Elke key representeert een vlak van het veld en elke value alle mogelijke getallen die ingevuld kunnen worden.

Er is gekozen voor een dictionairy omdat elk vlak van een puzzel een uniek coördinaat heeft en de access naar vlakken zo snel kan zijn. De list<int> value is gekozen omdat men nooit van te voren weet of een getal als is ingevuld voor een puzzel of niet, en tijdens het oplossen met normale back tracking en al helemaal met forward checking, de getallen die met een vlak geassocieerd kunnen zijn, sterk kan variëren. Een lijst biedt dan de flexibiliteit die een array niet biedt.

Welke functies/methoden zijn erop gedefinieerd?

Er zijn redelijk wat functies om het skelet van een puzzel op te krijgen, maar het is niet boeiend om die hier in detail te bespreken. Hiervoor verwijs ik naar de code en klasse *Grid.cs*. Deze is nauwkeurig becommentarieert en geeft de lezer duidelijk inzicht in hoe het skelet tot stand is gekomen.

Alle functie die specifiek op het puzzel/board zijn gedefinieerd staan allemaal in de board functies klasse (*BoardFunctions.cs*) en de constraint propagation klasse (*ConstraintPropagation.cs)*. Dit zijn statische klassen die functies/methodes bevatten die op elke puzzel aangeroepen kunnen worden. Deze functies zijn:

- MakeStartingBoardConsistent(board): krijgt een bord en returned een bord. De methode loopt over elk vlak van een puzzel en kijkt of er maar één getal in de lijst van mogelijke getallen staat van een bord. Als dit waar is, worden alle peers van dit vlak consistent gemaakt door het getal van het huidige vlak uit de lijst te halen van mogelijke getallen van elke peer.

- MoveLegal(square, board): krijgt een vlak en een puzzel mee als argumenten, het resultaat is een boolean waarde. Als een getal wordt ingevuld, kijkt deze methode of het ingevulde getal een legale bord oplevert. Dit wordt controleert om te kijken of het ingevulde getal in het vlak al reeds voorkomt in het vlak van een van zijn peers. De eisen zijn daarbij dat de lijst van mogelijke cijfers van een peers één groot is en zo ja, de getallen hetzelfde zijn. Als de waar is, is het gekozen getal illegaal en returned de methode false. Als dit voor geen enkele peer geldt is de zet legaal en returned de methode true.

- SquaresToSolve(board, methode): krijgt een puzzel en een string die de methode van oplossen representeert, het resultaat is een lijst van alle vlakken die nog ingevuld dienen te worden. Deze methode maakt een lijst aan van de vlakken die nog ingevuld dienen te worden. Dit gebeurd simpel weg door over een puzzel heen te loopen en voor elk vlak te kijken of de lijst van mogelijke getallen groter dan één lang is. Als een lijst groter is dan één, dan wordt de key van dat vlak aan een lijst van string toegevoegd. Deze lijst heet *squareList* en wordt voor iedere puzzel opnieuw geïnstantieerd. Omdat het loopen over een puzzel van linksboven naar rechtonder gaat, zal deze lijst automatisch geordend zijn van links naar rechts op te lossen vlakken. Voor de methode rechts naar links wordt deze lijst simpelweg opgedraaid. Voor ordening op domain size wordt er voor elk vlak gekeken hoeveel getallen er mogelijk ingevuld kunnen worden, de vlakken met de minste mogelijkheden komen vooraan te staan.

Forward Checking datastructuur

Dezelfde datastructuur van Grid en Board worden gebruikt voor het oplossen van puzzel met behulp van forward checking. Echter is er wel wat aan toegevoegd. De Board klasse krijg een extra overloaded instantie methode erbij. Deze methode ontvangt een board en maakt hier een deep copy van. Deze kopie, zonder referenties op geen enkele wijze aan het origineel, is nodig voor het Forward Checking algoritme. Onze implementatie van dit algoritme houdt een geschiedenis bij van borden die onafhankelijk moeten zijn van het bord waarvoor op dat moment getallen in vlakken worden ingevuld. Als er nog referenties zouden bestaan zullen de borden op de geschiedenis stack ook gewijzigd worden en zal het algoritme met foutieve oplossingen komen.

Implementatie chronologische BackTracking algoritme en Forward checking

Ons chronologische backtracking algoritme werkt met deze stappen:

1. Verkrijg het bord dat opgelost moet worden. (Solve Methode)
2. Maak het bord consistent. (Solve Methode)
3. Verkrijg een lijst met alle vlakken die leeg zijn en ingevuld dienen te worden. Deze lijst wordt zodanig geordend dat de volgorde van vlakken in de lijst van links naar rechts, rechts naar links, of geordend op domein grootte zijn. (Solve Methode)
4. Voor backtracking uit het op bord, met als start positie het eerst vlak uit de lijst te doorzoeken vlakken. (Backtrack Methode)
5. Backtracking geeft True of False terug. True als het bord is opgelost, false als dat niet is gelukt. Als true wordt teruggegeven door backtrack dan wordt het opgeloste bord teruggegeven, zo niet wordt null teruggegeven. (Solve Methode)

Hieronder in detail wordt met pseudo-code besproken:

- De oplos methode.
- Hoe backtracking wordt gedaan en de daartoe behoren hulp methodes.

Solve Methode:

```
// Geef aan deze methode, een bord dat nog niet is opgelost, in welke volgorde vlakken
doorzocht moeten worden als string
Solve(Board board, string searchMode) return board
{
	MakeStartingBoardConsitent(board);

	SquaresToSolve = FindSquaresToSolve(board, searchMode);

	startingSquare = SquaresToSolve[first element]

	if (searchmode == “backtrack”)
		if (backtrack(startingSquare))
			return board;
	if (searchmode == “backtrack+forwardChecking”)
		if (backtrackForwardChecking(startingSquare))
			return board;


	return null;
}
```

Backtracking methode:

```
Backtrack(string current position) return boolean
    {

        if (time is over 10 minutes)
            return false;

        if (position == null)
            return true;

        for each (number in (possible numbers for current square))
        {
            Current square = number;

            if (Movelegal(current square, board))
            {
                if (backtrack(next square to solve))
                    return true;
            }
        }

        Current square = possible numbers for current square;

        return false;
    }
```

De backtracking methode gebruikt een puzzel die voor de zoek klasse lokaal beschikbaar is. Het is een recursieve methode die telkens wordt aangeroepen op een nieuw vlak waarvoor een getal moet worden gekozen.

Er wordt allereerst gekeken over er nog geen 10 minuten voorbij zijn, dit is in de opdracht de tijd die het zoeken maximaal mag kosten. Als deze verstreken is mag er niet meer gezocht worden en is er geen oplossing gevonden.

Als de positie die mee wordt gegeven null is, dan zijn alle mogelijk vlakken succesvol ingevuld en dit betekent dat het bord dan ook volledig is ingevuld. De oplossing is gevonden en er mag true terug worden gegeven.

Als het bord nog niet is opgelost wordt er gekeken welk getal in het huidige vlak ingevuld kan worden. Elk vlak heeft een lijst met mogelijk getallen en deze worden een voor een bekeken. Er wordt telkens een getal ingevuld en gekeken of het resulterende bord legaal is. Zo ja dan wordt backtracking opnieuw aangeroepen met het volgende vlak de onderzocht moet worden. Zo nee dan het volgende getal geprobeerd.

Zodra geen enkel getal een legaal bord oplevert dan worden alle mogelijk getallen in het huidige vlak terug gezet en gaan we een stap terug.

Backtracking met forward search methode:

```
backTrackForwarchSearch(string currentPosition) return boolean
{

    /* same 10 minute check as regular BT */

    Previous board = new Board(current board)

    if (possible Numbers > 1)
        board history.Push(currentPosition, previous board)

    int lastNumber = possible numbers[last element]

    for each (number in possible numbers)
    {
         if (more than one number to check AND
             number IS NOT lastNumber      AND
             board NOT ON boardHistory)
                boardHistory.Push(currentPosition, previous board);

        current square = number;


        if (ForwardChecking(current board, current position))
        {
            if (visited squares DOES NOT contain (current position))
                visited squares ADD(current position);

             if (backTrackForwarchSearch(
                        Most constrained variable(currentPosition, boardToSearch)))
                return true;
        }

        else
        {
             currentBoard = boardHistory.Pop;
        }
    }

    visitedSquares REMOVE currentPosition;

   return false;
}
```

Dit algoritme begint hetzelfde als regulier backtracking, met een 10-minuten check en door te kijken of de doelstaat al is bereikt.

Vervolgens wordt er een bord opgeslagen in de geschiedenis zodat we bij backtracking de vorige staat nog hebben opgeslagen. Als er nog maar 1 mogelijkheid is, is het niet nodig om het bord op te slaan in de geschiedenis, want we zullen bij backtracken deze stap toch overslaan.

Nu vinden we het laatste nummer dat ingevuld moet worden in het vlak. Dit is later nodig om te controleren of het bord op de geschiedenis stack moet worden gezet.

Hierna proberen we alle nummers in te vullen in het vlak, daarnaast kijken we of het bord op de geschiedenis stack moet worden gezet. De logica achter deze check is: heeft hij meer dan 1 nummer

om te controleren, zo niet moeten we naar zijn voorganger terug. Is dit niet het laatste nummer, want dan hoeft het bord niet meer op de geschiedenis gezet te worden want als dit nummer fout gaat moet er naar zijn voorganger teruggegaan worden. En staat dit bord er niet al op, want we willen geen duplicaten.

Vervolgens maken we het bord consistent met behulp van forward checking. Hierbij wordt gecontroleerd of het domein leeg raakt bij het weghalen van de gekozen nummer uit de peers van het huidige vlak. Als dit niet gebeurd kunnen we het huidige vlak toevoegen aan de lijst met bezochte vlakken. Dit is nodig voor de most constrained variable methode.

Volg dit op met backtracking forward search, die de most constrained variable pakt op basis van de huidige positie en het huidige bord. Als dit mislukt, wordt het huidige bord vervangen door het meest recente bord op de boardHistory.

Indien forward checking volledig mislukt, moet het algoritme terug in de boom en moeten de bezochte posities weer uit de lijst gehaald worden voor een correct verder verloop. Hierna wordt false teruggegeven zodat de search methode op basis hiervan weer zijn volgende beslissing kan maken.

Resultaten

Na uitgebreid testen staan hier de resultaten voor 9x9 borden, 16x16 borden, 25x25 borden en de extreme 500 borden, opgelost door Backtracking (BT) met left-to-right(LR)/right-to-left(RL)/domain size (ds) heuristiek en door Backtracking met Forward Checking (BT+FS) met left-to-right(LR)/right-to-left(RL)/domain size (ds)/most-constrained-variable(mcv) heuristiek.

In de linker colom staat alle informatie die wij het programma hebben laten bijhouden over de set borden die we het elke keer gaven.

*BT+FS LR/RL/DS houd alleen rekening met de manier waarop het algoritme start, forward checking gebruikt altijd het most-constrained-variable heuristiek voor elke tussenstap.

9x9 boards (p096)	BT LR	BT RL	BT DS	BT+FS LR	BT+FS RL	BT+FS DS	BT+FS MCV
Total time to solve 50 boards (milliseconds)	5809	9065	30897	1108	1854	894	859
The board that was solved the fastest (milliseconds)	7	6	8	8	8	7	8
The board that was solved the slowest (milliseconds)	911	4085	9636	104	455	62	45
Average time to solve a board (milliseconds)	116	181	617	22	37	17	17
Total recursive BT calls	938224	1420447	5476486	11782	27107	7038	7038
Least recursive BT calls	34	27	26	29	26	26	26
Most recursive BT calls	145170	640954	1764649	1950	8780	763	763
Average recursive BT calls	18764	28408	109529	235	542	140	140
Solved amount of boards within 10 min time limit	50/50	50/50	50/50	50/50	50/50	50/50	50/50

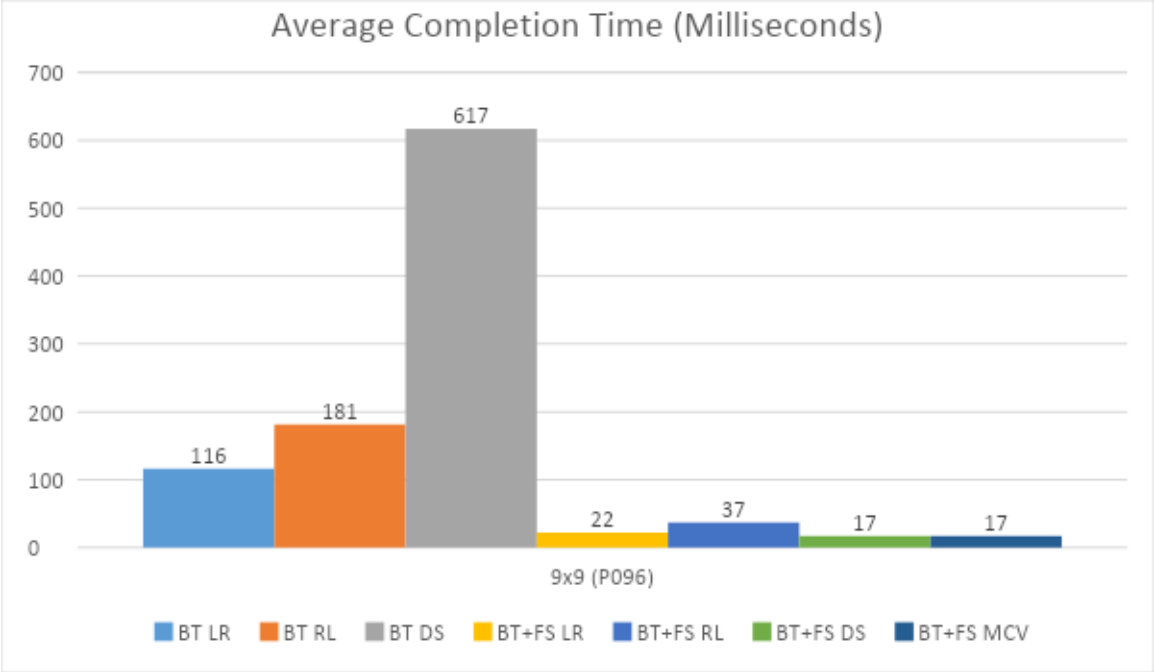
16x16 boards (16x16_1 + 16x16_2 + 16x16_3)	BT LR	BT RL	BT DS	BT+FS LR	BT+FS RL	BT+FS DS	BT+FS MCV
Total time to solve 3 boards (milliseconds)	1294364	602310	1800000	614691	600574	600190	600189
The board that was solved the fastest (milliseconds)	135884	353	600000	150	106	62	61
The board that was solved the slowest (milliseconds)	600000	600000	600000	600000	600000	600000	600000
Average time to solve a board (milliseconds)	431454	200770	600000	204897	200191	200063	200063
Total recursive BT calls	86794523	38882602	144367249	4179080	4010433	3886955	3894161
Least recursive BT calls	14532628	28322	29304830	658	156	226	226
Most recursive BT calls	41942106	38757180	66401730	4080096	4007398	3886302	3893508
Average recursive BT calls	28931507	12960867	48122416	1393026	1336811	1295651	1298053
Solved amount of boards within 10 min time limit	2/3	2/3	0/3	2/3	2/3	2/3	2/3

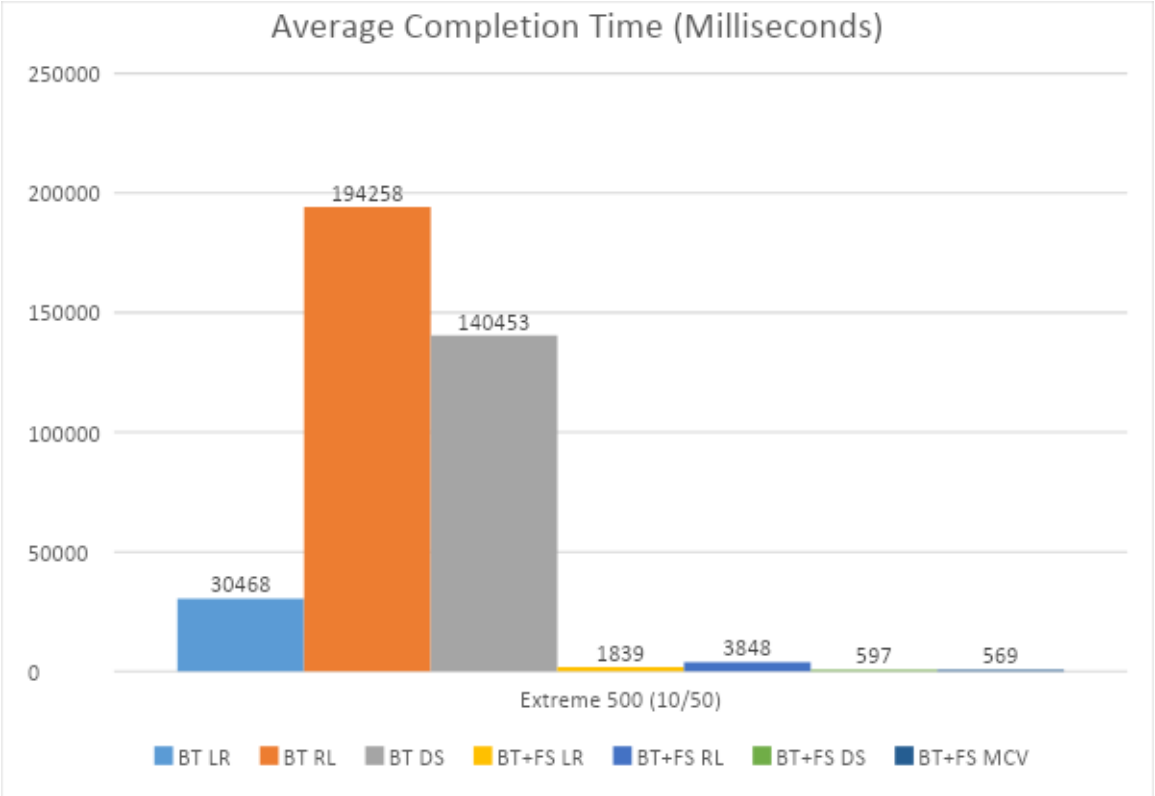
25x25 boards (25x15_1 + 25x25_2)	BT LR	BT RL	BT DS	BT+FS LR	BT+FS RL	BT+FS DS	BT+FS MCV
Total time to solve 50 boards (milliseconds)							600000
The board that was solved the fastest (milliseconds)							600000
The board that was solved the slowest (milliseconds)							600000
Average time to solve a board (milliseconds)							600000
Total recursive BT calls							1465370
Least recursive BT calls							1465370
Most recursive BT calls							1465370
Average recursive BT calls							1465370
Solved amount of boards within 10 min time limit							0/2

Extreme 500*	BT LR	BT RL	BT DS	BT+FS LR	BT+FS RL	BT+FS DS	BT+FS MCV
Total time to solve 10/50 boards (milliseconds)	304688	1942588	1404532	91965	192446	29859	28471
The board that was solved the fastest (milliseconds)	126	9357	207	15	30	18	13
The board that was solved the slowest (milliseconds)	165741	600000	600000	13506	71153	3031	2813
Average time to solve a board (milliseconds)	30468	194258	140453	1839	3848	597	569
Total recursive BT calls	53996946	283080742	229284859	1840980	3788811	525244	525244
Least recursive BT calls	19543	1391110	22879	135	345	76	76
Most recursive BT calls	29590845	89741113	95106217	277485	1401376	53111	53111
Average recursive BT calls	5399694	28308074	22928485	36819	75776	10504	10504
Solved amount of boards within 10 min time limit	10/10	9/10	9/10	50/50	50/50	50/50	50/50

*De eerste 10 voor BT en de eerste 50 voor BT+FS i.v.m. redelijke tijdsbeschouwing

De meest interessante gegevens voor ons waren de wisselende efficiënties van de verschillende heuristieken en het enorme verschil dat forward checking maakte in al onze tests. Op basis van deze bevindingen hebben wij twee grafieken opgesteld om dit het best te representeren.





Logischerwijs correspondeerde het aantal backtracking calls grofweg met de gemiddelde snelheid van het algoritme.

Discussie

Wanneer men naar de resultaten kijkt, vallen er gelijk een paar dingen op:

1. Backtracking met domain size heuristiek is in het slechtste geval veel trager dan left-to-right en right-to-left heuristiek. Wij denken dat dit komt door de willekeurigheid van de zoekboom. Domain size heuristiek zou theoretisch gezien efficienter moeten zijn omdat het intelligenter een vervolgstap in de zoekboom kiest, maar de realiteit is dat een willekeurig beginpunt zoals van links naar rechts of van rechts naar links toevallig eerder tot een antwoord komt in meeste gevallen. Dit zorgt ook voor de enorme uitschieters in BT DS, met het minst en het meest aantal recursieve backtracking calls.
2. Backtracking met forward checking is bij het oplossen van 9x9 borden duidelijk vele malen efficienter dan normale backtracking, waarbij forward checking met als beginpunt mcv de kroon spant. Het aantal backtracking calls is significant lager en de snelheden liggen ook veel dichter bij elkaar. Wat opvalt is dat de resultaten van BT+FS DS en BT+FS MCV praktisch hetzelfde zijn, waarbij het verschil in tijd verwaarloosbaar is. Dit komt natuurlijk om dat deze methoden ook essentieel hetzelfde zijn: mcv bepaald elke stap de domain size, en forward checking gebruikt heuristiek toch elke stap, dus initiele ordening zal exact dezelfde te-doorzoeken boom opleveren.
3. Hoewel bij backtracking met forward checking de keuze van de initiele heuristiek maar mee telt voor één stap, zorgt dit er wel voor dat het algoritme op een ander punt in de boom begint. Dit heeft als gevolg dat de beginkeuze van de rl, lr, ds en mcv heuristiek wel degelijk een verschil maakt in de efficientie van de oplossing.
4. Bovenstaande bevinden gelden voor alle borden: 9x9, 16x16, 25x25 en zelf de extreme 500 borden. Geen enkel algoritme was in staat om alle drie de 16x16 borden op te lossen (binnen een door ons redelijk bevonden tijdslimiet van 10 minuten per bord). De borden die wel gelukt zijn, zijn wederom te wijten aan het instappunt van de algoritmes. Backtracking met forward search en de mcv heuristiek komt weer als winnaar uit de bus met het aantal backtracking calls, zoals verwacht.
5. Geen enkel algoritme gepaard met geen enkele heuristiek was in staat om de 25x25 borden op te lossen in het door ons gestelde tijdslimiet. Met het simpele feit in ons achterhoofd dat het testen van een bord met een limiet 10 minuten duurt, er twee borden zijn en 7 algoritme/heuristiek combinaties (dit zou ons op zijn best 2.5 uur kosten), hebben we besloten de rest van de gegevens achterwegen te laten, nadat ons snelste algoritme geen van beide borden op kon lossen.

   We hebben wel nagedacht over een oplossing voor het hierboven beschreven probleem. Naast een efficienter algoritme of simpelweg meer proceskracht, leek het ons mogelijk om aan strengere constraint propagation te doen, door simpele trucjes en wiskundige waarheden die sudoku-spelers over de jaren hebben opgebouwd toe te voegen, zodat de grootte van een zoekboom drastisch afneemt. Natuurlijk kunnen we het programma ook gewoon laten draaien tot het bord is opgelost, maar wat servetjes-wiskunde wijst ons al uit dat dit langer gaat duren dan voor dit experiment wenselijk.