Chronologische Backtracking gecombineerd met Most-Constrained-Variable heuristiek

Silas Peters, 4419197

De stappen die ik ondernam om tot mijn uiteindelijke code te komen, in dit document omschreven, zijn goed terug te zien op https://github.com/SilasPeters/SudokuCracker/tree/main/SudokuCrackerCBT .

Terminologie:
Tegel: een enkel vakje van de Sudoku die de waardes 1 - 9 kan bevatten.
Blok: zo'n 3x3 groep van tegels, waar er 9 van zijn in de sudoku.
Naburige tegels: de naburige tegels van een tegel, zijn alle andere tegels die in dezelfde kolom, rij of blok zitten.

Eerste implementatie

Chronologische Backtracking - Ons originele idee

Het algoritme krijgt een incomplete sudoku. Als allereerste worden alle tegels knoopconsistent gemaakt. Dan itereert het algoritme over alle tegels – van linksboven naar rechtsonder – om een geschikte expansie te vinden voor elke tegel. Dit doet hij met de volgende recursieve 'subroutine':

Hij itereert over elke mogelijke waarde X van de tegel T die dus binnen het domein van T valt. Hij geeft de tegel de waarde X (tenzij de waarde van T een geven is), en gaat in recursie naar de volgende tegel aan de hand van depth-first-search. Zodra de subroutine uit die diepere laag van recursie terugkomt, geeft die laag terug aan de huidige laag of de recursie met succes een oplossing wist te vinden, gegeven de huidige toestand. (Merk op dat de wijzigingen aan de toestand die door de recursieve aanroep zijn gemaakt niet persistent zijn.) Indien succesvol, returned dus deze functie dat een oplossing is gevonden als deze tegel de waarde X heeft. Zo niet, gaat hij opnieuw in recursie met deze keer de volgende mogelijke waarde als X (die binnen het domein van T valt). Als geen enkele waarde binnen het domein van de huidige tegel leidt tot een oplossing, geeft deze recursieve functie aan dat in de huidige toestand van de sudoku geen oplossing te vinden is. Nu 'backtracked' het algoritme en worden alle wijzigingen die aan de toestand werden gedaan op deze laag ongedaan gemaakt. Note: in de rest van dit document gebruik ik de woorden 'iteratie' en 'recursieve aanroep' als synoniemen, want dat zijn ze in deze context.

Backtracking momenten

Binnen de iteraties wordt tussendoor gecontroleerd of ondernomen acties niet tot een onmogelijke toestand van de sudoku leiden, gegeven de constraints van een klassieke

sudoku. Zo kan vroegtijdig onnodige branching voorkomen worden. Dit wordt op twee punten gedaan. Na het invullen van X in de huidige toestand van de sudoku wordt gekeken of de toestand van de resulterende sudoku nog een geldige partiële oplossing is. In andere woorden, er wordt gecontroleerd of een wijziging aan de tegel niet het feit verandert dat alle kolommen, rijen en blokken voldoen aan de constraint dat er geen dubbele waardes voor mogen komen.

Als de resulterende waardes van de tegels dus leiden tot een mogelijke oplossing - en de constraints niet worden verbroken - wordt daarna gekeken hoe dit de domeinen van de buren van de tegel die net is ingesteld beïnvloedt. Dit is onderdeel van het 'look-ahead' gedeelte van de Chronologische Backtracking. Redundante waardes worden uit de domeinen gefitlerd. Als na het bijwerken van de domeinen van de naburige tegels blijkt dat een domein leeg is, is er geen oplossing mogelijk voor de huidige toestand van de sudoku.

In beide gevallen geeft de subroutine dus aan dat na het invullen van X geen oplossingen meer over zijn. Als beide gevallen niet voorkomen, gaat de subroutine in recursie; de huidige branch is het waard.

Terminatie

Het algoritme termineert wanneer alle recursieve subroutines niet verder kunnen/hoeven te itereren over hun huidige branch/laag in de (impliciete) zoekboom. Indien de sudoku een geldige oplossing heeft, termineert het algoritme doordat een subroutine de laatste (lege) tegel heeft gecontroleerd en heeft bepaald dat de huidige toestand van het algoritme een mogelijke oplossing is. Alle subroutines sluiten dan af met een signaal van success, en de code die de recursie aanriep rapporteert de laatst bekende toestand van de sudoku.

Als er iets mis ging, of als de sudoku geen oplossing heeft, termineert het programma nadat de eerste subroutine (de hoogste laag in de recursie), met alle mogelijke waardes van T die binnen het domein van T valt, geen success ondervind door in recursie te gaan (net als alle andere subroutines dus). Alle mogelijkheden zijn dus langsgegaan. Dit betekent echter niet dat alle waardes van elke tegel zijn geprobeerd: de domeinen konden voor vroegtijdige maar terechte backtracking zorgen. De code die de recursie aanriep rapporteert de originele sudoku met eventueel wat progressie die was ondervonden tijdens het zoeken - een incomplete sudoku dus.

Het probleem met de implementatie

De representatie van de Sudoku

Wat hierboven niet genoemd is, is de exacte implementatie. De details zijn terug te vinden (en uitgelegd) in de broncode, maar ik zal bij deze een overzicht geven. Ik representeer de toestanden met een ``struct Sudoku``, welke uit niks anders bestaat

dan een array van tegels, en methodes om die array uit te lezen (maar niet om te manipuleren). Een tegel wordt gerepresenteerd met een ``struct Tile`` welke op zijn beurt weer uit niks anders bestaat dan een ``ushort`` en bijpassende methodes. De ``ushort`` bestaat uit 16 bits die samen de waarde, het domein en het feit of de tegel gefixeerd is opslaan. De bijpassende methodes maken deze bits toegankelijk om uit te lezen en te manipuleren, alsof het niks bijzonders is. Voor de rest werkt heel het genoemde algoritme met Sudoku instanties en hun tegels.

Het probleem

Het is al opvallend dat ik hierboven zulke minimalistische representaties gebruik, en zelfs af hang van bit-manipulaties. Dit heeft te maken met het probleem met de representaties van wat ik en mijn toenmalig team de vorige keer hebben ingeleverd. Heel het algoritme hierboven heeft een belangrijk uitgangspunt: alle wijzigingen in een recursieve aanroep zijn *niet* persistent. Dit heeft als grote voordeel dat we de wijzigingen die ons algoritme aanmaakt op de toestand van de sudoku, niet hoeven bij te houden en terug te draaien wanneer een subroutine aangeeft dat de huidige toestand geen oplossing biedt en we willen backtracken. Dit scheelt ontzettend veel rekenwerk en voorkomt bovendien ingewikkelde logica. Echter bleek het na wat pogingen niet mogelijk om alle wijzigingen die in een recursieve aanroep gebeuren niet persistent te maken. Wijzigingen 'lekten' door naar de recursie-laag erboven, wat zorgde voor dat er niet correct een oplossing gevonden kon worden. Uiteindelijk hebben we iets ingeleverd wat heel ons idee omgooide om hier om heen te werken, maar dit vertraagde alles en maakte veel onleesbaar. Nu, in mijn tweede poging, heb ik meer onderzoek gedaan naar 'value-types' en 'reference types' en ontdekte dat meer datastructuren dan verwacht 'reference-type' zijn, waardoor wijzigingen persistent worden.

De oplossing

Om wijzigingen niet persistent te maken, moest ik alleen maar 'value-types' gebruiken. Oftewel, zelfs arrays moesten zo veel mogelijk vermeden worden. Daarom wordt een tegel opgeslagen in een ``struct`` (welke value type is) en bevat hij letterlijk alleen maar een ``ushort`` (ook value type). Een ``struct Sudoku`` moet echter wel op een reëele manier een lijst aan tegels opslaan, want het werd wel heel omslachtig om ook dat in een bitstream op te slaan. Dus hier werd ik genoodzaakt toch een array te gebruiken. Daarom heb ik een ``Sudoku.Clone()`` methode geïmplementeerd, welke een ongerelateerde kopie maakt van een sudoku, zodat wijzigingen in die instantie van de sudoku niet 'lekken' naar de originele sudoku. ``TryForwardCheck()`` maakt hiet als enige gebruik van.

In het kort heb ik problemen data-persistentie opgelost, waardoor ons originele idee correct geïmplementeerd kan worden.

Het resultaat

Correctheid

Het algoritme is getest op alle 5 de gegeven sudokus. Alle 5 de oplossingen waren correct. Ook heb ik het algoritme getest op een incomplete sudoku die geen oplossing heeft. Hiervoor vond het algoritme inderdaad geen oplossing, dus tot op zo ver zijn er geen false-positives.

Rekentijd

De complexiteit van het algoritme tot nu toe is Ω(1) en O(9^*n*) waar *n* gelijk is aan het aantal niet gefixeerde tegels. *n* staat hier expres niet voor het totale aantal tegels, omdat dat aantal constant is in een sudoku. Ω(1), omdat in het gunstigste geval de hele sudoku al gevuld is. O(9^*n*), omdat de zoekboom een graad heeft van 9 en er *n* niveaus in zitten.

Benchmarks

Method	Sudoku	Mean	Error	StdDev	Gen0	Allocated
CBT	0	53.11 us	0.138 us	0.129 us	0.3662	28.17 KB
CBT	1	89.39 us	0.146 us	0.137 us	0.6104	46.27 KB
CBT	2	745.72 us	1.528 us	1.355 us	4.8828	380.02 KB
CBT	3	142.50 us	0.230 us	0.204 us	0.9766	72.88 KB
CBT	4	42.63 us	0.063 us	0.059 us	0.3052	22.66 KB
CBT	5	2,269.66 us	2.977 us	2.486 us	15.6250	1189.86 KB

Merk op dat sudoku 5 geen mogelijke oplossing heeft. Dit geeft dus een inzicht in hoe langzaam het algoritme is als hij veel moet backtracken/trashen.

Onder de sudokus die een oplossing hebben, is de mean gemiddeld 215,07 us. Dit is aanzienlijk snel. Ook is er maar gemiddeld 110 KB aan geheugen nodig. Ik heb in mijn leven niet al te veel benchmarks gezien, maar dit ziet er in mijn ogen erg efficiënt uit.

Verbeteringen

Voordat ik een Most-Constrained-Variable heuristiek ga toepassen, wil ik eerst nog kijken naar wat aandachtspuntjes die ik tegenkwam, die het huidige algoritme kunnen versnellen.

Terminatie-conditie

Het eerste wat ik zie als verbeterpunt, is dat ``TrySearch()`` (de recursieve functie) als eerste controleert of de sudoku al volledig is ingevuld, en dus is opgelost. Het ding is dat er al wordt bijgehouden wat de index van de huidige tegel is die word gecontroleerd. Deze index word geïncrementeerd bij elke volgende tegel. Als deze index gelijk is aan 81, betekent dit dus dat de laatste tegel net is ingevuld.

Dit betekent dat ik niet elke keer hoef te testen of elke tegel al een waarde heeft; ik houd toch al bij hoeveel tegels een waarde hebben. Ik pas het algoritme dus zo aan dat ``Sudoku.AllTilesFilled()`` niet wordt aangeroepen. In plaats van dat wordt de huidige tegel index vergeleken met 81.

Controlleren voor partiële oplossing

Een deel van de instructies waren dat je na elke iteratie van het CBT algoritme, je eerst moet controleren of je nog steeds een partiële oplossing hebt na de laatste expansie. Echter, lijkt mij dit onnodig. In het begin maak je alles knoopconsistent; elk domein is zo minimaal mogelijk en elke waarde in elk domein is een valide waarde in de huidige toestand. Zodra je een waarde toewijst aan een tegel, worden alle naburige domeinen opnieuw aangescherpt. Dus, is er ten alle tijden nooit een waarde in een domein die een constraint zal verbreken. Waarvoor wordt dan gecontroleerd of een deeltoestand een partiële oplossing is?

Als ik een 'breakpoint' zet bij de code die alleen wordt aangeroepen als een deeltoestand niet een partiële toestand is, dan termineert mijn algoritme alsnog zonder ooit die breakpoint langs te zijn gegaan. Oftewel, er is geen geval voor alle 6 de sudokus dat mijn algoritme ooit een niet-partiële oplossing aanmaakt.

Mijn volgende aanpassing is dus om deze stap compleet over te slaan.

Resultaat

Correctheid

Het algoritme geeft nog steeds exact dezelfde oplossingen. Ook heb ik mijn wijzigingen beargumenteerd, dus ik zie voldoende reden om aan te nemen dat mijn implementatie nog steeds correct is.

Rekentijd

De optimalisaties hebben niet het aantal expansies aangepast, dus de complexiteit is nog steeds Ω(1) en O(9^*n*).

Benchmarks

Na het aanpassen van de terminatie-conditie:

Method	Sudoku	Mean	Error	StdDev	Gen0	Allocated
CBT	0	53.28 us	0.235 us	0.220 us	0.3662	28.17 KB
CBT	1	88.48 us	0.075 us	0.070 us	0.6104	46.27 KB
CBT	2	742.98 us	3.399 us	3.179 us	4.8828	380.02 KB
CBT	3	140.16 us	0.539 us	0.504 us	0.9766	72.88 KB
CBT	4	42.40 us	0.060 us	0.056 us	0.3052	22.66 KB
CBT	5	2,272.74 us	2.418 us	2.261 us	15.6250	1189.86 KB

Geen tot nauwelijks verschil, maar wel simpelere code. Opvallend is dat het geheugengebruik identiek is. Dit kan aanduiden dat exact evenveel expanties plaatsvonden.

Na het verwijderen van de controle voor een partiële oplossing:

Method	Sudoku	Mean	Error	StdDev	Gen0	Allocated
CBT	0	50.00 us	0.051 us	0.048 us	0.3052	28.17 KB
CBT	1	78.10 us	0.153 us	0.135 us	0.4883	46.27 KB
CBT	2	650.69 us	0.745 us	0.697 us	3.9063	380.02 KB
CBT	3	127.63 us	0.091 us	0.081 us	0.7324	72.88 KB
CBT	4	38.50 us	0.080 us	0.075 us	0.2441	22.66 KB
CBT	5	2,063.86 us	1.258 us	1.176 us	11.7188	1189.86 KB

Opvallend is dat het geheugengebruik nog steeds niet is verminderd. Dat valt te verklaren met het feit dat het 'allocated memory' het geheugen is dat in beslag word genomen door de 'managed heap': een gedeelte van het geheugen wat beheerd word door de garbage collector, welke niet al te efficiënt is. Ik stop super veel variabelen (ook voor het voorkomen van persistente wijzigingen) in de 'stack', welke een alternatieve plek is in het geheugen. Deze plek is dan ook letterlijk een stack, en veel efficiënter dan de heap. Zo efficiënt zelfs, dat de stack in 64-bits computers beperkt is tot 4MB omdat nooit meer nodig is. De benchmarks houdt dus alleen het geheugengebruik van de heap bij, omdat dat resource-intensive is - de stack niet. Mijn conclusie is dus dat ``IsPartialAnswer()``, welke een methode is die gebruikt werd voor het controleren van

de deeltoestanden - welke nu dus is weggelaten - letterlijk alleen maar de stack benut. Vandaar dat je het verschil in geheugengebruik niet terug ziet komen in de benchmarks.

Kijkend naar de mean rekentijden, kosten de sudokus die een oplossing hebben gemiddeld 188,984 us om op te lossen. Dit is 12% minder dan eerst. Dit is best een verbetering.

Conclusies

De aanpassingen behoud ik, aangezien het geen nadelen toch zich brengt en het de code simpeler en sneller maakt. Een 12% versnelling is erg mooi meegenomen.

Most-Constrained-Variable implementatie

Plan van aanpak

Om mijn huidige implementatie van het Chronological Backtracking algoritme uit te breiden zodat het een Most-Constrained-Variable heuristiek aanhoudt, wou ik eerst proberen een ordering bij te houden van alle tegels, gebaseerd op de groote van hun domein. Een ordering werd veel genoemd in de slides, en is het meest intuïtief. Je wilt immers weten welke lege tegel het kleinste domein heeft.

Eerste ideeën

Mijn eerste idee was om alle tegels op te slaan in een een heap. Deze heap zou alle tegels die nog leeg zijn bovenaan hebben. De tegels die leeg zijn worden onderling nog gesorteerd op wie de kleinste domein heeft. Op deze manier kan met logaritmische complexiteit bijgehouden worden welke tegel nu most-constrained is.

Een ander idee was om een stack bij te houden van lineaire orderingen. Elke laag in de stack representeert de volgorde waarop de tegels gesorteerd zijn aan de hand van de MCV heuristiek. Zo wordt bij elke recursieve aanroep van het algoritme de stack met één ordering uitgebreid. Als de toestand niet tot een oplossing leidt, dan wordt de bovenste waarde van de stack gehaald.

Gewoon simpel zoeken

Het probleem met de ideeën die hiervoor genoemd zijn, is dat het te complex wordt om de volgorde bij te houden. Het enige moment waarop de theoretische ordering verandert, is wanneer een Forward Check wordt gedaan welke 2*8 domeinen

aanscherpt van naburige tegels. Het gevolg is dus dat de ordering die tot op dat punt gold, nu totaal op zijn kop gegooid moet worden omdat 16 van de 81 waardes bijgewerkt zijn. De kracht van de stack en heap oplossing is dat je niet alle tegels bij elke iteratie compleet opnieuw sorteert, maar de ordering bijwerkt gebaseerd op mutaties in de toestand. Echter, zoals ik net liet zien, zijn deze wijzigingen erg ingrijpend. Dus, met alle complexiteit die komt kijken bij een heap of stack die bijgehouden moet worden, in combinatie met de helft elke keer alsnog moeten sorteren, is dit waarschijnlijk niet van grote nut. In ieder geval moet er een snellere variant zijn.

Dus nam ik even een stapje terug. Het nadeel van een heap of stack is dus de complexiteit van de aanpak. Dus wat nou als ik lekker simpel denk? Toen kwam ik op de conclusie dat het bijhouden van wijzigingen in de ordering nooit erg efficiënt zal zijn, omdat er zo veel verandert per iteratie. Dus is een simpelere oplossing om geen stack of heap of een andere datastructuur te implementeren, en gewoon bij elke iteratie alle tegels langs te gaan om te zoeken naar een lege tegel met het kleinste domein. Ja, ik ga dan altijd 81 tegels langs per iteratie, maar op deze manier hoef ik niet te sorteren, een stack of heap bij te houden, en is de code ook nog eens een stuk simpeler. Dus hieruit kwam mijn eerste idee: elke iteratie doet nog steeds precies hetzelfde als eerst, maar voert zijn acties uit op de most constrained variable waarnaar aan het begin van de iteratie gezocht is.

Resultaat

Correctheid

De logica van het algoritme heb ik niet bijgewerkt. Het enige wat is verandert is welke tegel werd behandeld in een iteratie. Dit is altijd een lege tegel, tenzij er geen lege meer over zijn (wat de terminatie conditie is). In principe is dit idee simpel en goed te implementeren. Ook geeft het algoritme nu nog steeds dezelfde correcte antwoorden op de test sudokus. Ik vindt dit genoeg reden om aan te nemen dat mijn algoritme nog steeds correct is.

Rekentijd

Ten opzichte van vanilla CBT is mijn algoritme alleen complexer geworden in dat het bij elke iteratie altijd alle 81 tegels langs gaat om te zoeken naar de MCV. Dit is dus niet een factor van n, maar een constante. De big-oh en big-omega notaties zijn dus niet veranderd.

Benchmarks

Method	Sudoku	Mean	Error	StdDev	Gen0	Allocated
CBT	0	48.21 us	0.141 us	0.132 us	0.3052	28.17 KB
CBT_MCV	0	97.45 us	0.249 us	0.233 us	0.6104	56.27 KB
CBT	1	81.25 us	0.216 us	0.202 us	0.4883	46.27 KB
CBT_MCV	1	116.54 us	0.473 us	0.420 us	0.7324	66.56 KB
CBT	2	680.53 us	0.442 us	0.369 us	3.9063	380.02 KB
CBT_MCV	2	228.38 us	0.240 us	0.224 us	1.4648	127 KB
CBT	3	124.41 us	0.376 us	0.352 us	0.7324	72.88 KB
CBT_MCV	3	105.41 us	0.308 us	0.288 us	0.7324	60.77 KB
CBT	4	38.74 us	0.113 us	0.105 us	0.2441	22.66 KB
CBT_MCV	4	84.29 us	0.176 us	0.165 us	0.6104	50.2 KB
CBT	5	2,078.41 us	4.665 us	4.363 us	11.7188	1189.86 KB
CBT_MCV	5	43.99 us	0.253 us	0.236 us	0.3052	25.56 KB

Iteraties per sudoku:

Sudoku	CBT	CBT_MCV	
0	130	50	
1	218	56	
2	1585	111	
3	331	53	
4	110	46	
5	2792	10	
Total:	2374	326	(Negeert sudoku 5)
Avg.:	474	65	(Negeert sudoku 5)

Onthoud dat sudoku 5 hier geen oplossing heeft.

Conclusies

CBT met een MCV heuristiek verminderd het aantal iteraties per sudoku aanzienlijk: gemiddeld 65 iteraties t.o.v. 474 iteraties is veel minder. Ook eist CBT_MCV bij elke sudoku minder iteraties dan bij CBT. Je zou dus verwachten dat CBT_MCV ten alle sneller is. Echter, heeft de MCV implementatie alleen voor een versnelling gezorgd bij sudoku 2 en 3 (en 5). De initiële kosten om de MCV te bepalen zijn dus bij sudoku 0, 1 en 4 groter dan de kosten die bespaard worden door minder vertakkingen te hebben.

Dit kan te verklaren zijn doordat de MCV heuristiek alleen effectief is bij toestanden waar de domeinen erg in grootte variëren, en waar de constraints vrij strak zijn. Waarschijnlijk is dit dus niet het geval bij sudoku 0, 1 en 4. Kans is ook een factor: misschien zijn sudoku 0, 1 en 4 net zo ingericht dat in het begin alleen de hogere waardes in de domeinen tot een oplossing leiden; terwijl mijn implementatie eerste de lagere waardes probeert. Dit kan dus nog steeds zorgen voor veel onnodige vertakkingen.

Ook is te zien dat memory usage gecorreleerd is aan de rekentijd. De benchmarks van de CBT implementatie gebruiken nog exact evenveel geheugen als voorheen. Verder is het humoristisch om te zien hoeveel minder iteraties CBT_MCV nodig heeft om tot de conclusie te komen dat sudoku 5 geen oplossing heeft.

In het algemeen is MCV niet altijd een winnaar. De gebruiker moet zelf inschatten of hij de nieuwe heuristiek wilt toepassen.

Een verbetering op het het MCV: buckets

Een laatste idee die ik heb om de Most-Constrained-Variable heuristiek efficiënt toe te passen, is om niet een ordering bij te houden maar 9 'buckets'. In bucket X zitten alle tegels met een domein van grootte X. De MCV is dan de eerste de beste waarde in bucket 1. Wanneer een domein van een tegel wordt bijgewerkt, wordt de tegel uit zijn oude bucket gehaald en in zijn nieuwe bucket gezet. Op die manier zijn die 16 wijzigingen aan domeinen per iteratie een stuk efficiënter.

Helaas heb ik geen goede implementatie voor dit weten te bedenken. Ik heb bijvoorbeeld geprobeerd om een 'Buckets' klasse aan te maken, wat niks anders is dan 9 HashSets van coordinaten van tegels. Het probleem zit hem echter weer in mijn uitgangspunt om met value-types te werken. Om wijzigingen binnen iteraties niet persistent te houden moet ik overal value-types gebruiken, of een ``Clone()`` methode aanmaken. Echter zijn alle vormen van arays of lists of een andere vorm van collectie bij

definitie reference-type. Omdat dit primitieve types zijn kan ik hier geen ``Clone()`` methode tegenaan mikken in mijn eigen implementatie van een collection. Nu heb ik bij ``Sudoku`` wel een ``Clone()`` methode aan weten te maken, maar hier ging het om een veel simpelere datastructuur. In principe heb ik mijzelf in de vingers gesneden door af te hangen van value-type variabelen.

Ik heb geprobeerd de bestaande ``PriorityQueue`` klasse van C# te gebruiken, mijn eigen klasse te maken zoals hierboven genoemd, en andere aanpakken zoals een ``SortedSet``. Door een ``SortedSet`` te gebruiken in plaats van alles constant sorteren wordt al snel complexer kwa rekentijd dan gewoon simpel dingen hersorteren - en alweer is een ``SortedSet`` reference type. En hoe zou een ``SortedSet`` dingen sorteren?

Ik trek nu de conclusie dat value-types bedoeld zijn voor variabelen die binnen één methode een tijdelijke waarde representeren, of verder zo simpel mogelijk zijn. Helaas hangt heel mijn implementatie van mijn algoritme af van het gebruik van value-types, dus om dit aan te passen moet ik praktisch heel mijn prakticum opnieuw doen. Zoals verder uitgelegd een paar regels terug zie ik ook geen haalbare oplossing die nog efficiënter is én value-type-based.

Discussie

CBT is in het algemeen een sterk en intuïtief algoritme, welke sudokus in no-time oplost. Om dit nog verder uit te kunnen breiden met een Most-Constrained-Variable heuristiek, laat zien hoe de complexiteit van verdere uitbreidingen ten koste kunnen gaan van de snelheid - en potentiëëel niet op kunnen wegen tegen de voordelen van de optimalisatie. Zo is het een terugkerend thema in algoritmiek dat de beste ideeën vaak de simpele ideeën zijn. Ik zelf vind dat de toegevoegde heuristiek wel een toevoeging kan zijn, omdat het ideaal is om snel te termineren indien een sudoku niet op te lossen is; het is ook extreem handig om te kunnen bepalen of een sudoku op te lossen valt. Om ook te kunnen concluderen of het in het algemeen een goede toevoeging is, moet dit algoritme getest worden op een stuk of 100 sudoku's, zodat we een representatief verschil kunnen zien in prestaties met of zonder.

In toekomstige pogingen om MCV te onderzoeken kan het beste niet met value-types gewerkt worden, omdat dit al voor veel te veel gedoe heeft gezorgd. Voor de rest vindt ik de stappen die ik afnam erg reproduceerbaar, ook in andere vormen.