Sudoku 2 documentatie

Alieke de Heer (6532527), Andreas Meeldijk (0892734), Mote van Reij (0990345)

**Inhoudsopgave**

[**Algoritme**](#_6fezzx4omta3) **2**

[Chronological Backtracking (CBT)](#_xbbqf0pmto1v) 2

[**Implementatie**](#_bnj9v6vc92mt) **3**

[Het inlezen van de sudoku’s](#_qeza8ccqcz5k) 3

[Algemene toelichting opbouw code](#_xbolgmnd83oy) 3

[Extra algoritmes geïmplementeerd](#_91j7suvskbok) 4

[**Resultaten**](#_kcma3awttlpq) **5**

[Rekentijd](#_rlr5d2g0vx5k) 5

[Extra experiment](#_yfycn8ahlypw) 5

[**Conclusie**](#_ttxchw9i8oky) **5**

[**Discussie**](#_g2miaxjc97wc) **6**

[Resultaten](#_unmwq6gbas4r) 6

[Samenwerking](#_jy59d7igqpn) 6

# Algoritme

Het zoekprobleem voor deze opdracht is een sudoku bestaande uit 9x9 vakken. Deze zijn opgedeeld in negen 3x3 blokken. In elk blok moeten de cijfers één tot en met negen allemaal precies eenmalig voorkomen.

De *begintoestand* van dit zoekprobleem is een sudoku die deels is ingevuld. De al ingevulde vakken mogen niet meer worden aangepast. In elk blok mogen de cijfers één tot en met negen eenmalig worden gebruikt. De *zoekruimte* bestaat uit alle partiële oplossingen, waarbij de al ingevulde cijfers altijd op dezelfde plek zullen staan. De *toestandsruimte* is elke invulling waarbij de gefixeerde vakjes op hun vaste plek staan. De *doeltoestand* is de toestand waarbij zowel in elk blok, in elke rij en in elke kolom de cijfers één tot en met negen precies eenmalig voorkomen.

Bij deze sudoku opdracht zien we het zoekprobleem als een Constraint Satisfaction Problem (CSP). Dit houdt in dat elk variabel vakje een domein heeft van de waarden 1 tot en met 9. Bij een sudoku worden de domeinen van variabele vakken ingeperkt door de waarden van de gegeven vakken. De inperking van de domeinen, maakt dat het een reflexieve constraint is. Deze reflexieve constraints worden toegepast door het probleem een keer knoop consistent te maken.

## Chronological Backtracking (CBT)

Chronologische backtracking is een variant van backtracking. Het verschil is dat een knoop alleen geëxpandeerd wordt als deze een partiële oplossing van het probleem is. Een partiële oplossing is een partiële toestand die geen enkele constraint van het CSP schendt. Het werkt door systematisch alle mogelijke oplossing paden te proberen, beginnend bij een begintoestand, en terug te gaan naar eerdere stappen als een pad doodlopend blijkt te zijn. Dit proces wordt herhaald totdat een oplossing wordt gevonden of totdat alle paden zijn geprobeerd. Het chronologische aspect van het algoritme komt doordat het pad dat het laatst is geprobeerd, als eerste wordt teruggedraaid als het een doodlopend pad blijkt te zijn. Hierdoor wordt er efficiënt gebruik gemaakt van de reeds bekeken oplossing paden. Het algoritme zal dus voor elke vakje uit de sudoku alle mogelijkheden uit zijn domein proberen. Als dit leidt tot een domein dat leeg raakt, worden de domeinen geback tracked en wordt het volgende uit het domein geprobeerd.

# Implementatie

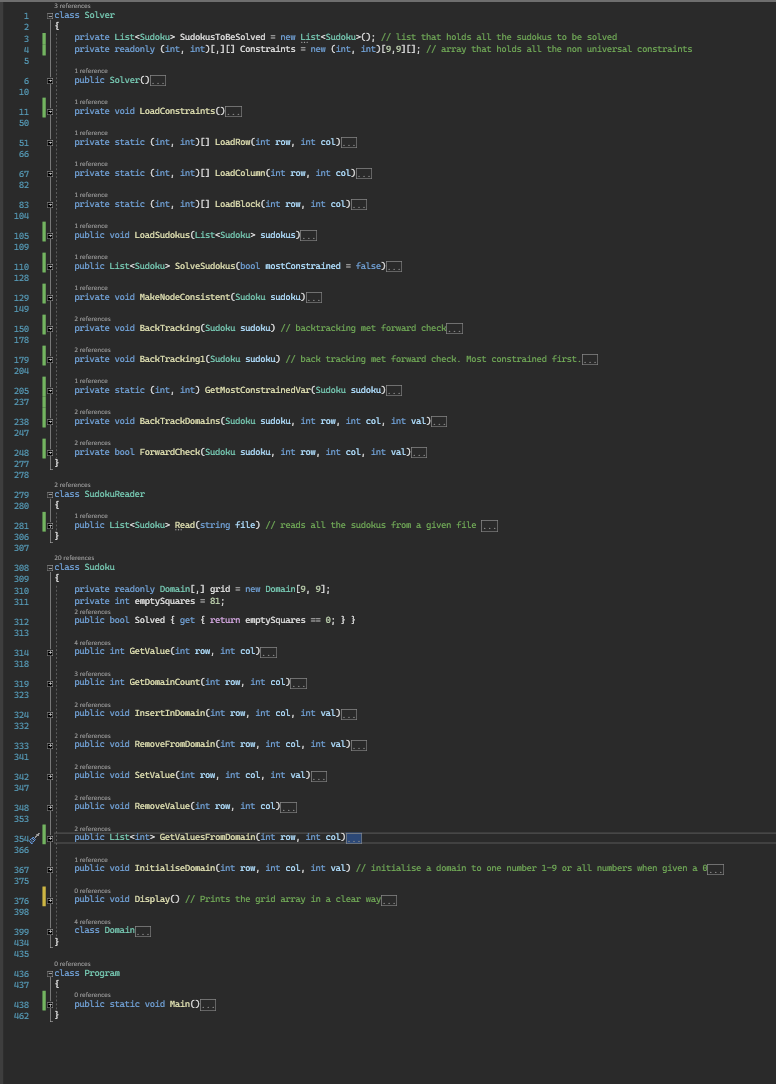
## Het inlezen van de sudoku’s

De sudoku’s worden ingelezen met behulp van de *Read* methode in de klasse *SudokuReader*. Deze methode wordt aangeroepen met een string van de locatie waar het bestand dat wordt ingelezen zich bevindt. Dit (.txt) bestand wordt vervolgens ingelezen met een StreamReader. Vervolgens worden alle getallen uit het bestand opgeslagen in een array en wordt een Sudoku object gemaakt waarbij elk vakje een domein krijgt en een bijbehorende waarde (0 als niet-gefixeerd). Dit wordt voor alle sudoku’s uit het bestand gedaan en deze worden opgeslagen in een lijst.

Om een bestand in te lezen is het dus van belang de string waarde van de variabele “path” in de Main methode van de “Program” klasse aan te passen naar de filepath van het bestand dat wordt ingelezen.

## Algemene toelichting opbouw code

De klasse *Sudoku* representeert een sudoku als een 2D-array met voor elk vakje een domein. In deze klasse bevindt zich dan ook een *nested-class* genaamd *Domain*. Dit is een object dat een domein als array bijhoudt, een waarde en een lengte van het domein. Verder heeft deze klasse een aantal hulp methodes die m.b.v. de rij, de kolom en de waarde de domeinen of de waardes kunnen veranderen of opvragen. De *InitialiseDomain* methode geeft een vakje in de sudoku een domein 1 t/m 9 als de waarde 0 is en anders een domein gelijk aan de waarde, dit is het geval als er sprake is van een gegeven waarde. De *Display* functie wordt gebruikt om de sudoku array (grid) op een overzichtelijke manier uit te printen op de console.



De *Solver* klasse bevat een array met alle non-universele constraints. Dit is een 2D array met voor elk vakje in de sudoku een array met alle constraints erin. Deze constraints worden geïnitialiseerd met de *LoadConstraints* methode die gebruik maakt van 3 andere initialisatie methodes (*LoadRow*, *LoadColumn* en *LoadBlock*). De hoofdmethode lost de nog op te lossen sudoku’s op d.m.v. de methodes *MakeNodeConsistent* en *BackTracking* (of *BackTracking1*). De *BackTracking* methode checkt m.b.v. de *ForwardCheck* methode of domeinen leeg raken als een waarde gekozen wordt. De domeinen kunnen ook geback-tracked worden door *BackTrackDomains*.

Om het backtracken makkelijker te maken wordt er gewerkt met een +1/ -1 systeem voor de waarden 1 t/m 9 in de domeinen. Als een waarde nog in het domein zit heeft deze de waarde 1 in de array. Als de waarde door een constraint uit het domein moet worden gehaald wordt deze waarde 1 verlaagd. Als een beperking wordt opgeheven door backtracking dan wordt deze waarde weer met 1 verhoogd. Dit voorkomt dat je bij het backtracken een waarde onrechtmatig terug stopt in het domein als er meerdere vakjes een beperking leggen op een domein.

## 

## Extra algoritmes geïmplementeerd

Om het programma sneller te maken hebben een tweede methode gemaakt voor het backtracken. Deze methode implementeert de most constrained variable heuristiek. Deze methode wordt aangeroepen door in de Main methode de *SolveSudokus* methode van de *Solver* klasse aan te roepen met de boolean true als parameter. Deze boolean waarde verander je bij de variabele *mostConstrained* in de *Main* methode. Het idee achter deze toevoeging is dat backtracking het vakje kiest dat het meest constrained is (het kleinste domein heeft). Dit wordt gedaan in de methode *GetMostConstrainedVar*. De most constrained variable heuristiek heeft als doel om zo vroeg mogelijk inconsistenties te vinden en daardoor minder onnodige combinaties te doorzoeken. De heuristiek werkt het beste voor relatief strakke CSP-problemen waarbij er verschil is tussen de domeingroottes. Doordat een sudoku een probleem is dat aan deze eisen voldoet, is deze heuristiek aardig effectief. Daarnaast zijn sudoku's vaak zo gemaakt dat ze door mensen op te lossen moeten zijn. Dit doen mensen vaak door te achterhalen dat er nog maar één cijfer mogelijk is voor een vakje (het domein is dan één). Omdat je dus vaak dit soort vakjes hebt met een domein van één, die door de heuristiek als eerste ingevuld worden en sowieso juist zijn, verklein je eerst je probleem verder zonder onnodige combinaties te maken.

# 

# Resultaten

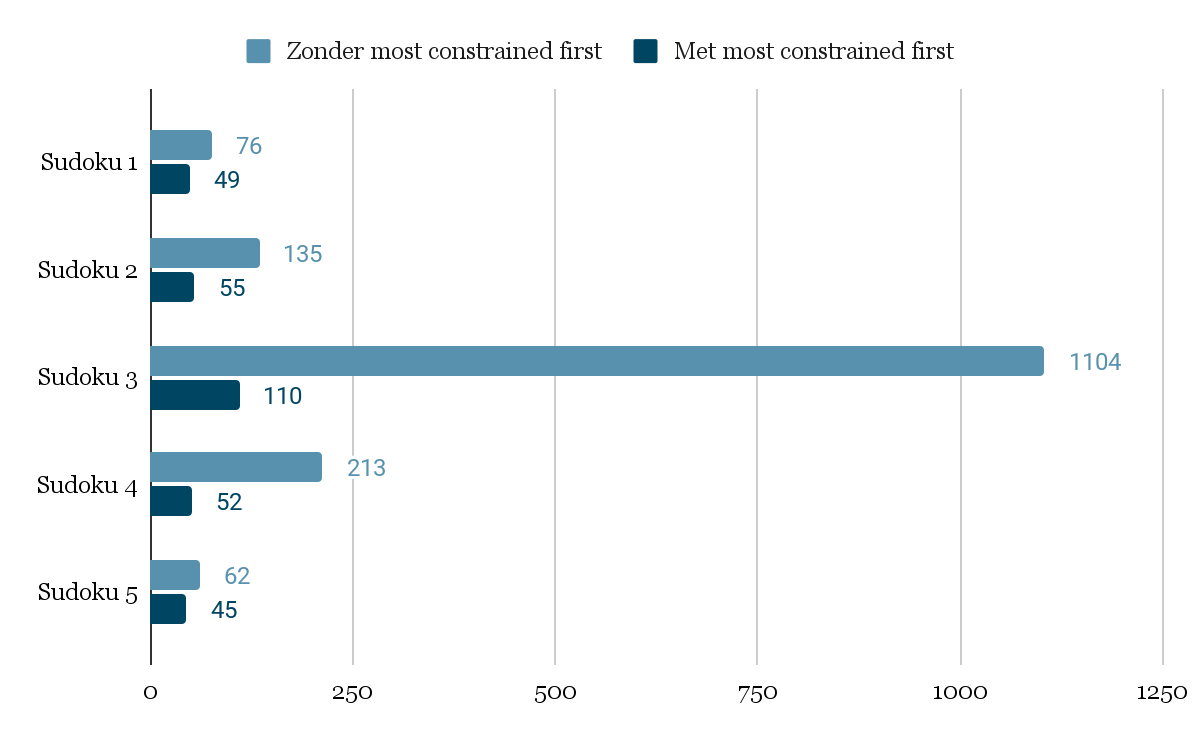
## 

## Rekentijd

Om te meten hoe snel en efficiënt ons programma nou is hebben we het 50 keer laten runnen en laten timen hoe lang dit duurt. Gemiddeld voor deze 50 runs kost het **8 milliseconden** als we gebruik maken van backtracking zonder de *most constrained first* techniek (dus de *Backtracking* methode). Als we hier wel gebruik van maken (*Backtracking1* methode) kost het gemiddeld **4 milliseconden** (dit is de gemiddelde tijd voor het runnen van alle 5 sudoku’’s uit het bestand). Dit is dus een significant snellere methode dan het iterated local search algoritme (ILS-algoritme) en de most constrained first implementatie is ook 2x sneller dan de normale backtracking. Dit komt dus ook overeen met onze verwachting van de prestatie.

## Extra experiment

Om nog beter te zien hoe efficiënt ons algoritme is hebben we voor elke sudoku bijgehouden hoe vaak het een waarde uit een domein probeert. Dit is dus het totaal over de hele sudoku. Dit hebben we voor zowel backtracking zonder most constrained first als met constrained first. De resultaten hiervan zijn weergegeven in grafiek 1 hieronder.



grafiek 1: hoe vaak een sudoku waardes uit domeinen probeert (aantal pogingen)

Hieruit blijkt dat sommige sudoku’s makkelijker op te lossen zijn dan andere, dit komt relatief gezien overeen met de runtijd en moeilijkheid van deze sudoku’s. Wat verder opvalt, is dat het algoritme veel minder pogingen nodig heeft als de meest constrained variabele eerst wordt gekozen. Dit komt overeen met onze verwachting en uitleg in het deel [*Extra algoritmes geïmplementeerd*](#_k6o0m2nfamso).

# Conclusie

Na het uitvoeren van dit practicum zijn we tot een aantal conclusies gekomen. Zo is het CBT algoritme significant sneller dan het ILS algoritme en is het significant sneller om de variabele af te gaan op volgorde van kleinste domein (most constrained). Verder konden wij het backtracken efficiënt doen door voor elk vakje een domein bij te houden. Aangezien een variabele afhankelijk is van zijn blok, kolom en rij is dit hiermee makkelijk bij te houden.

# Discussie

## Resultaten

Uit de resultaten blijkt dat het chronogical backtracking algoritme (CBT-algoritme) significant sneller is dan het iterated local search algoritme en dat het significant sneller is om de variabelen af te gaan op volgorde van het meest constrained domein. De resultaten zijn bij het CBT-algoritme ten opzichte van het ITL-algoritme, niet afhankelijk van random walks, wat er voor zorgt dat er geen vorm van kans in de snelheid van het oplossen van de sudoku's zit. Wel zouden resultaten op verschillende computers, met andere specificaties verschillend kunnen uitvallen. Ook zijn de tijden van de twee manieren van backtracking in het CBT-algoritme afhankelijk van de manier van implementatie, maar de onderlinge vergelijking tussen beide implementaties is wel betrouwbaar. Dit omdat beide implementaties verder hetzelfde algoritme gebruiken. Het enige wisselende element is de backtracking methode.

## Samenwerking

De samenwerking voor deze opdracht is prettig verlopen. We waren op tijd begonnen en hebben vooral in de eerste paar werkcolleges na de kerstvakantie veel samen aan de opdracht gezeten en uitgedacht hoe we het programmeren gingen aanpakken. Dit is iets wat erg prettig is ervaren, het feit dat we eerst goed gingen zitten om te bedenken hoe we het gingen aanpakken en tegen welke problemen we eventueel later zouden kunnen aanlopen. Hierdoor, tegen de tijd dat we echt begonnen met het belangrijkste deel programmeren, verliep dit relatief soepel.

Nog steeds vonden we het verdelen van de programmeertaken wat moeilijker, maar hebben we vooral in het begin veel samen in de werkgroep gedaan of dat er in de werkgroep een begin werd gemaakt en dat daarna dit werd opgepakt door iemand om dat af te maken. Nog steeds zou het voor een volgende keer misschien een verbeterpunt zijn om bij het uitdenken van het programmeren meteen te bedenken wie wat gaat doen en zo de taken te verdelen.

Voor het verslag zijn we deze keer ook eerder begonnen, en konden we er iets meer de tijd voor nemen. Dit was erg prettig, zeker omdat dit de vorige keer uiteindelijk best wel op laatste moment werk aankwam.

Al met al is de samenwerking prettig verlopen en kijken we positief terug. Ook was het een leuke opdracht om aan te werken en waren de werkcolleges erg gezellig.