Verslag CompInt

van Viggo de Breij, Lex Klaassen & Thijmen van der Meijden

## Inleiding

In dit verslag wordt beschreven hoe het chronological backtracking algoritme met behulp van het forward checking algoritme kunnen worden gebruikt om Sudoku-puzzels op te lossen. Sudoku is een populair puzzel- en logisch denkspel dat bestaat uit een raster met 9x9 vakjes, waarbij sommige vakjes zijn ingevuld met cijfers en andere leeg (en in ons geval 0) zijn. Het doel van het spel is om de lege vakjes in te vullen met cijfers van 1 tot 9, zodanig dat elke rij, kolom en blok (3x3 vakjes) exact één keer de cijfers 1 tot 9 bevat. Het algoritme voor chronological backtracking wordt gebruikt om de zoekruimte van mogelijke oplossingen te verkennen, terwijl het algoritme voor forward checking wordt gebruikt om de zoekruimte te verkleinen door ongeldige keuzes te elimineren. Samen werken deze algoritmen samen om efficiënt en effectief de juiste oplossing voor de puzzel te vinden. We zullen de resultaten bespreken en ook vergelijken met de resultaten van de vorige opdracht.

## Methode(s)

De sudoku wordt gerepresenteerd door een 1d array. De input string wordt dus letterlijk omgezet naar een (0-based) array. Dit brengt iets meer moeilijkheden met zich mee maar vanuit intuïtie denken we dat het efficiënter is dan een 2d array. Ook vinden we het een leuke uitdaging en zo geven we het een beetje onze eigen twist van het programma.

Het programma bevat 2 basis algoritmes; CBT\_rec & ForwardCheck.

**CBT\_rec**

‘CBT\_rec’ is ons chronological backtracking algoritme. Het is een recursief algoritme met 2 base cases en een inductie gedeelte. De eerste base case checkt of de sudoku die het algoritme onder handen neemt een doeltoestand is. Als dat zo is zijn er geen 0’en meer in de array van waardes en wordt de variabele ‘nul\_index’ -1 omdat de IndexOf functie dat returnt als er geen index is gevonden met waarde 0. De tweede base case kijkt of het domein leeg is gemaakt van de nul\_index in de sudoku. Als dat zo is moet het een stap terug omdat er dan ergens een fout is gemaakt (omdat er nergens een leeg domein kan voorkomen). De inductiestap maakt een successor aan van de huidige sudoku en kloont de waardes en domeinen van de huidige sudoku om er aanpassingen aan te kunnen maken zonder dat de huidige sudoku aangepast wordt. Dan gaat het verder werken met het kind om te kijken of die kan resulteren in een juiste oplossing door te checken of de successor zich aan de constraints houdt. Als dat zo is dan moet je de waarde die je net in het vakje hebt ingevuld verwijderen uit de mogelijke waardes en moet het verder in recursie met de successor sudoku. Als het niet kan (en dus niet in recursie gaat) dan doet het dat nog een keer maar dan gaat het in recursie met de parent.

**ForwardCheck**

Deze methode is het forward check algoritme. Het neemt een vakje onder handen en gaat kijken of het mogelijk is om de waarde van dat vakje uit de domeinen te halen van de indexen van de rij, kolom en 3x3 blok (omgeving). De manier waarop we door de omgeving gaan is door eerst de eerste index van de kolom waar de meegegeven index zich in bevind uit te rekenen door “index % 9” en dan telkens plus 9 doen als we naar de volgende index in de kolom willen gaan. Voor de rij is het principe hetzelfde alleen nu rekenen we het begin van de rij uit door “index - index % 9” te doen en dan elke keer + 1 te doen. Voor het doorlopen van de indexen van het blok hebben we lang gewerkt aan een formule die het linker bovenste index terug zou geven van elk van de indexen van dat blok. Dit is het uiteindelijk geworden:

“blok\_index = index - index % 9 % 3 - (index - index % 9 - ((index - index % 9) / 9 - (index - index % 9) / 9 % 3) \* 9)” (deze formule stond in geitenbloed op mijn plafond geschreven). Dit is een samenvoeging van meerdere stappen, zo rekent het de kolom en rij uit van het linker bovenste vakje en gebruikt deze om te kijken hoe vaak het naar boven en links moet. Vanuit daar kunnen we de rest van de indexen van het blok infereren door het statisch te definiëren hoeveel die index verschilt met het begin van het blok. Zo kunnen we makkelijk met een enkel for loopje door alle 27 indexen van de omgeving loopen.

Als ergens in de omgeving (na aanpassing) een van de domeinen leeg wordt dan returnt het ForwardCheck algoritme false aangezien er dan niet gehouden wordt aan de contraints. Als het wel kan returnt het true. Het past dus wel de sudoku altijd aan. Ook al is de aanpassing problematisch voor de andere domeinen. We rollbacken het probleem in het backtrack algoritme.

Ook bevat het nog wat andere essentiële methodes.

**MaakSudoku**

Met deze methode zetten we een array van string om naar de startwaardes van de sudoku, en geven we elk van deze waardes een domein van 1 tot en met 9, vervolgens gebruiken we changedomainofneighbours om het startveld knoop-consistent te maken door de domeinen van de lege vakken aan te passen op basis van de vakken die van te voren zijn ingevuld.

**RemoveVal**

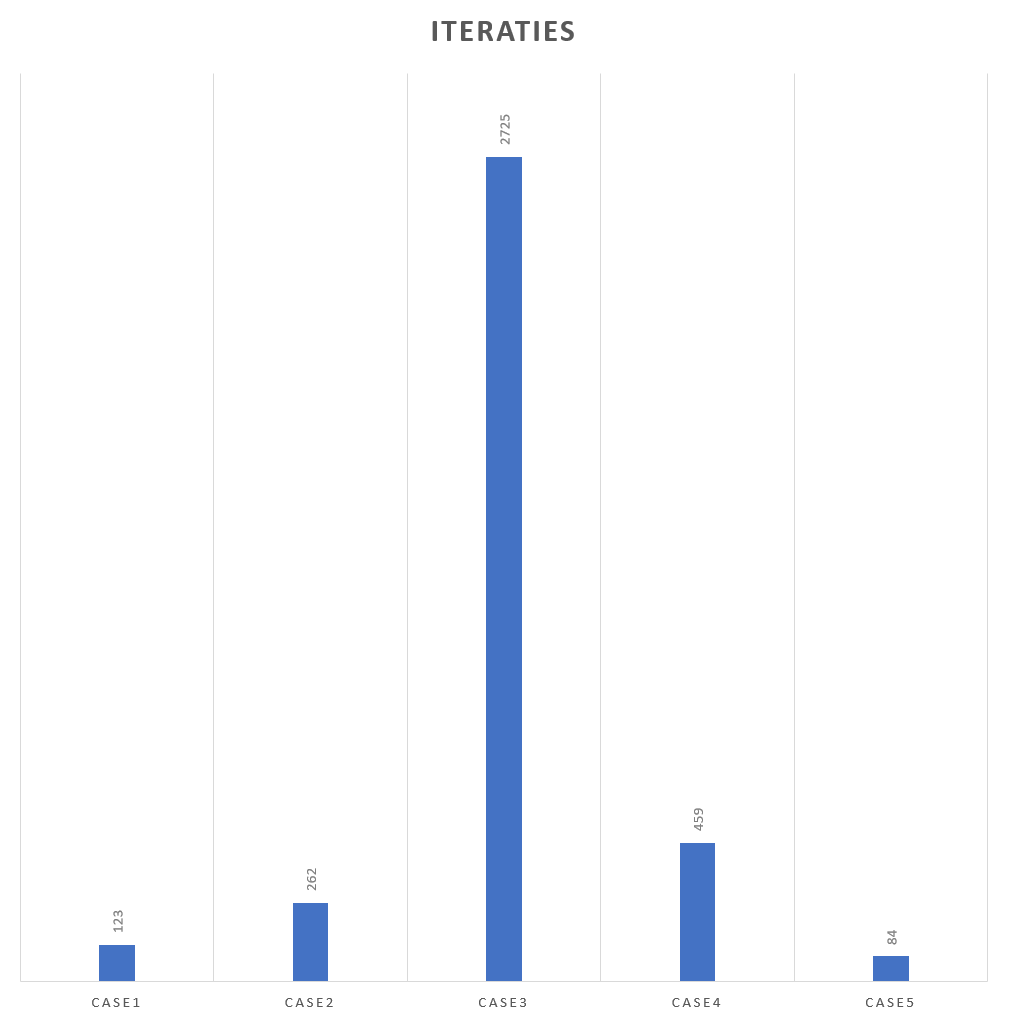
Deze methode verwijdert een meegegeven waarde uit het domein van een meegegeven vakje. Het zet eerst alle waardes in het domein in een normale array omdat dat makkelijker bewerken is. Vervolgens verwijdert hij die meegegeven waarde uit die array en zet het de array weer om in een array van de juiste lengte. Daarna vervangt het het oude domein.

**PrintWaarde**

‘PrintWaarde’ schrijft de waardes van alle sudoku vakken in een mooie representatie van een sudoku grid. Deze methode is er puur voor ons om te kijken hoe onze code werd geïmplementeerd op de sudoku. Het is ook handig voor de gebruiker om een beeld te krijgen van de puzzel en om de opgeloste sudoku weer te geven.

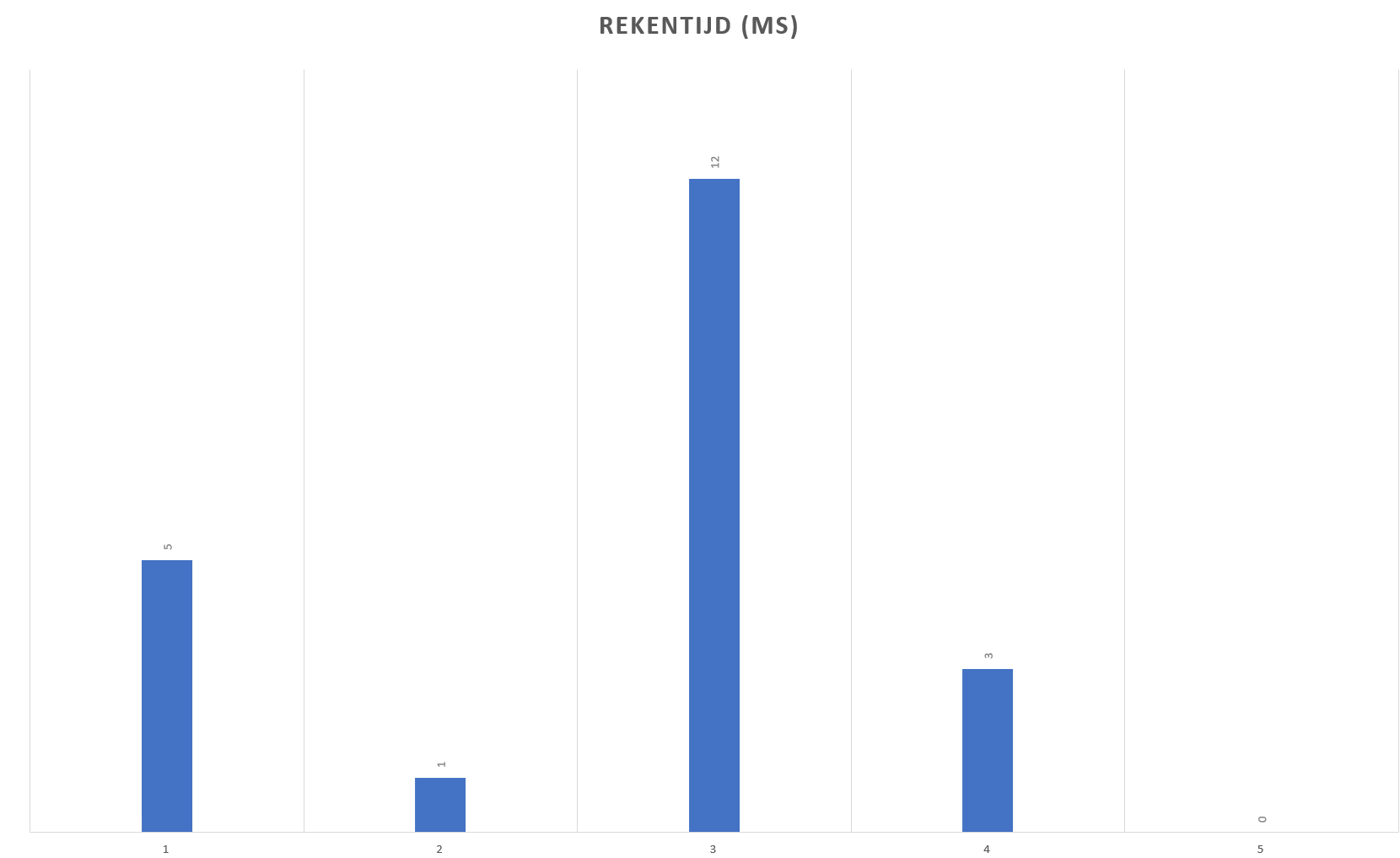
## Resultaten

We hebben een aangepaste versie van ons basisprogramma gemaakt dat elke case ‘experimenten’ aantal keer test. Het print vervolgens het gemiddeld aantal tijd (in ms) en iteraties per case. In onze resultaten hebben we de gemiddelden genomen van 50 tests.



Grafiek

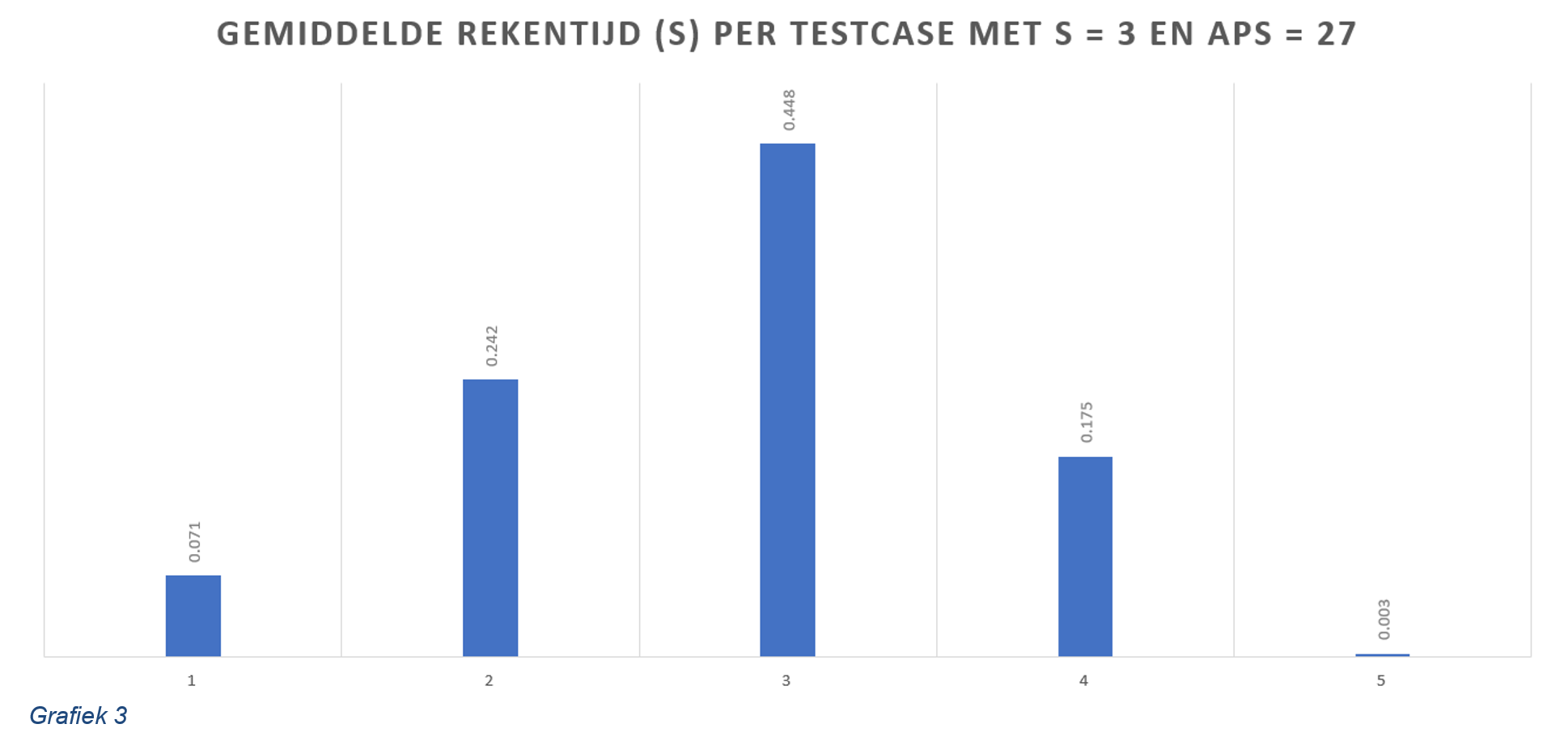
In grafiek 1 kan je het aantal iteraties zien dat het duurt voordat de sudoku is opgelost.



Grafiek

In grafiek 2 zie je de gemiddelde rekentijd (in ms) dat het duurt om 1 sudoku op te lossen.

Hieronder volgen wat resultaten van ons vorige programma om te kunnen vergelijken.



In grafiek 3 zie je de gemiddelde rekentijd (in s) per case met de beste combinatie van variabele.

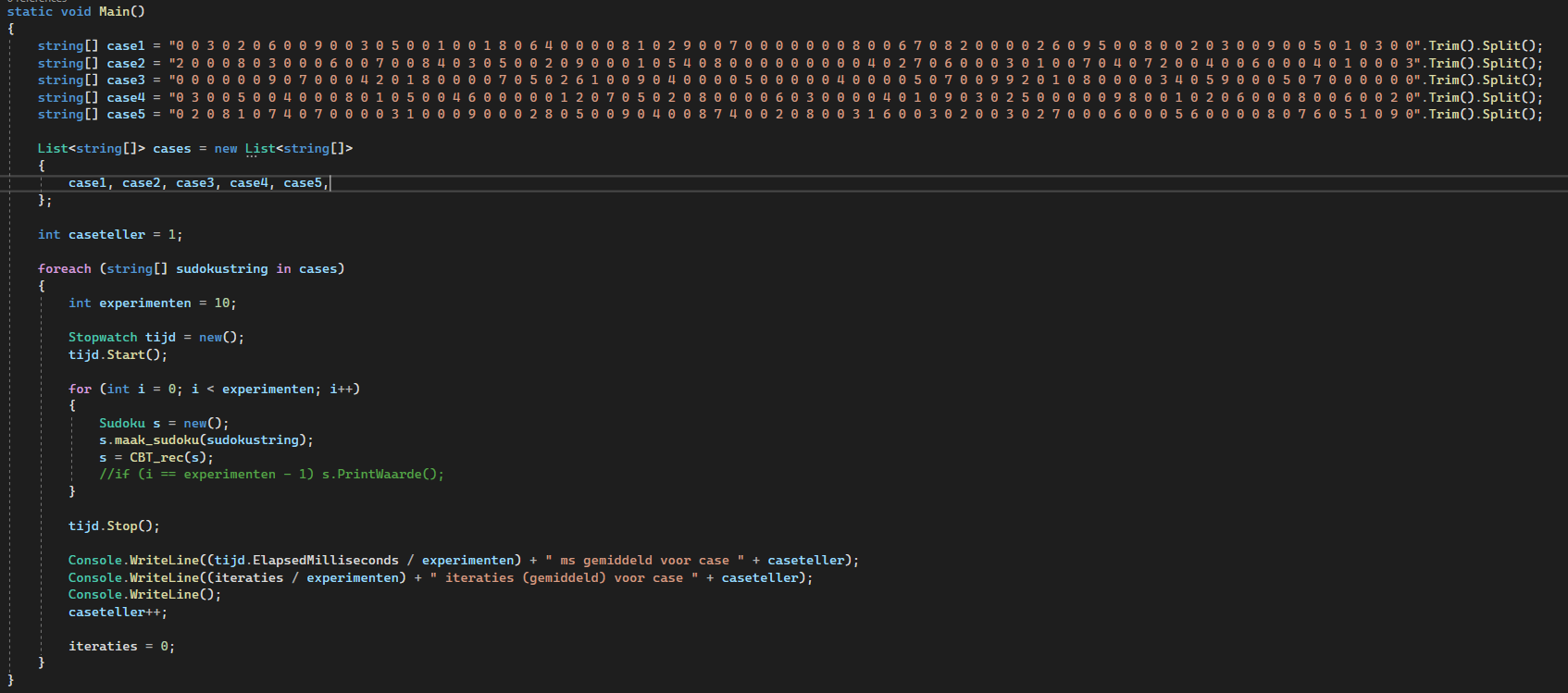
## Conclusie

Er is te zien dat case 3 het moeilijkst is om op te lossen en dat case 5 het makkelijkst is. Dit is terug te zien in het aantal iteraties en de rekentijd. Iets om op te merken is dat cases 2 en 4 meer iteraties hebben dan case 1 maar wel een kortere rekentijd. Dit zou kunnen komen door het aantal experimenten of bijvoorbeeld de computer waarop de resultaten zijn getest. Dit betekent dus niet perse dat deze resultaten overeenkomen met de theoretische resultaten.

Het is overduidelijk dat deze Sudoku Solver, met behulp van CBT met pruning, veel sneller is dan ILS. De trend van het aantal iteraties komt overeen met de gemiddelde rekentijd uit opdracht 1. Echter komt de gemiddelde rekentijd niet echt heel erg duidelijk overeen met de vorige opdracht. CBT lost het in het wel bij elke case sneller op maar er is maar een kleine gelijke trend. Dit is namelijk dat case 3 nog steeds het langste duurt om op te lossen en dat case 5 de makkelijkste is. Echter is er te zien dat het oplossen van case 2 in deze versie makkelijker is dan case 1 terwijl het juist andersom was in de vorige opdracht.

## Discussie

Het verloop van het programmeren liep wat stroef. Aan het begin dachten we dat het best een makkelijke opdracht zou zijn. Het bleek na de eerste tegenslag eigenlijk best een moeilijke opdracht. Een paar hordes waar we tegenaan liepen; Het uitvogelen van de formule om het linker bovenste vakje van het blok uit te rekenen op basis van een willekeurige index van dat blok heeft voor de nodige koppijn gezorgt. We hebben het backtracken geprobeerd met een stack en hebben tellertjes per vakje geïmplementeerd om bij te houden welke waardes al een keer geprobeerd waren maar dat werkte niet voor ons. Tevens kwamen we ook meermaals in de knoop met het kopiëren van arrays zonder dat ze naar elkaar refereerde, dit loste we uiteindelijk op door de “Clone()” methode te gebruiken. Uiteindelijk hebben we besloten om het met parent pointers te doen en de backtrack methode recursief te maken. We zijn erg blij met het resultaat, vooral omdat het ons flink wat denkwerk en hoofdpijn heeft gekost.

We hebben een aangepaste ‘experiment’ versie van ons programma gemaakt omdat dat veel makkelijker is om gemiddelde rekentijden en iteraties te berekenen per case. Hieronder is de aangepaste Main methode van de experimenten branch te zien.