Verslag CompInt

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

## Inleiding

In dit verslag wordt beschreven hoe het hillclimb algoritme met behulp van het random walk 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 hillclimb algoritme is een zoekalgoritme dat kan worden gebruikt om een oplossing te vinden voor dit soort problemen door stap voor stap naar betere oplossingen te zoeken. Het algoritme kan vast komen te zitten in lokale minima, om dit te verhelpen is het random walk algoritme er ook in verwerkt. In dit verslag zullen we beschrijven hoe de algoritmes werken en hoe we het toepassen om Sudoku-puzzels op te lossen. Ook hebben we verschillende variabelen getest en de uitkomst daarvan zullen we analyseren.

## 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 3 basis algoritmes; Iterated Local Search, Hill Climb en Random Walk.

**Hill Climb**

Het Hill Climb algoritme declareert als eerste de blok indexen in de 1d array, de beste score (wat in eerste instantie de huidige score van het bord is), en de array van het beste veld, wat ook in eerste instantie het huidige veld is. Ook declareert het een HashSet van 2d-tuples om de paren die al geswapt zijn bij te houden. Dan selecteert het algoritme door middel van de random klasse een willekeurige blok index waar de swaps in uitgevoerd zullen worden. Het algoritme declareert ook dynamisch op basis van de geselecteerde blok index de bijbehorende indexen van de vakjes in het blok. Dit gaat als volgt: de index van het eerste element van het blok is gelijk aan de geselecteerde blok index, deze noemen we “j”, de volgende elementen van die rij zijn ook de volgende elementen in de array dus “j + 1” en “j + 2”. De indexen van de elementen in andere rijen zijn gelijk aan de eerste rij plus 9. Dus voor de tweede rij: “j + 9”, “j + 10” en “j + 11”; En de derde rij “j + 18”, “j + 19” en “j + 20”. Dit geeft alle indexen van het blok zoals het in de 1d array op te halen valt. Dan loopt het algoritme door alle indexen van het blok en voor elke index loopt het nog een keer door de indexen van het blok om dan te checken of de twee indexen al geswapt zijn. Zo niet, dan maakt het een nieuwe sudoku waarvan het veld een kopie is van het huidige veld en probeert de twee indexen van de loop te swappen door middel van onze Swap methode die een boolean teruggeeft als het succesvol was. Zo ja, dan voegt het de twee geswapte indexen omgekeerd toe aan de lijst met al geweeste paren (omgekeerd zodat het niet tweemaal dezelfde indexen probeert te swappen). Dan wordt de heuristiek na de swap uitgerekend en dit samen met de huidige score wordt vergeleken met de beste score tot nu toe. Als deze lager of hetzelfde is dan de beste wordt het beste veld gelijk gesteld aan de gemaakte kopie en wordt de score ook aangepast naar de nieuwe, potentieel lagere heuristische waarde. Na beide loops wordt het echte huidige sudoku veld verandert naar het beste veld en de score ook.

**Random Walk**

Het random walk algoritme kiest eerst een willekeurig blok van 3x3 waarin hij cijfers gaat omwisselen. Dit gebeurt door eerst 1 beginindex van alle blokken te pakken en met behulp van dat nummer slaan we alle vakjes in dat blok op in een array. Dan worden alle ‘vaste’ vakjes uit die array gegooid en roepen we de ‘Swap’ methode aan met 2 willekeurig gekozen vakjes uit de overgebleven array. Daarna wordt ‘Heuristiek’ aangeroepen om de score te updaten.

**Iterative Local Search**

ILS is een algoritme dat hill climb met random walk combineert met bepaalde parameters. Deze parameters zijn ‘S’ en ‘APS’ (kort voor ‘aantal plateau stappen’). S bepaalt het aantal stappen dat random walk doet voordat de sudoku weer terug wordt gegeven aan hill climb. De parameter APS bepaalt na hoeveel dezelfde scores random walk wordt aangeroepen om zo uit een plateau te komen.

Het programma bevat verder nog wat andere methodes.

**String\_to\_sudoku**

‘string\_to\_sudoku’ zet de input string om in een 1d array en geeft elk al ingevuld nummer de eigenschap ‘vast = true’.

**Vakkenvuller**

‘Vakkenvuller’ wordt aangeroepen op de input sudoku array en vult elk 3x3 blok met zijn missende nummers. De essentie van de vakkenvuller methode is als volgt; Het loopt drie keer, vermenigvuldigt deze ‘r’ waarde met 27 om zo de juiste rij indexen van de blokken te geven. Dan loopt het nog drie keer per ‘r’ waarde en vermenigvuldigt deze ‘k’ waarde met 3 om zo het linker bovenste vakje van het juiste blok op te halen. Hierop worden namelijk alle indexen van het blok gebaseerd, evenals hoe dat gebeurt in het Hill Climb algoritme. Dan gebruiken we een lijst van de getallen van 1 tot en met 9 waar we door loopen om de vaststaande waardes in de sudoku eruit te halen. Dan houden we een lijst over met alle waardes die we nog in het blok moeten invullen. Deze shuffelen we en vullen de niet vaste waardes met de nog overige getallen van de lijst. Dan roepen we de methode aan om de beginscore uit te rekenen van het ingevulde bord.

**BeginScore**

‘beginScore’ berekent de score van de input sudoku waarbij elk blok is ingevuld met de missende nummers. Dit doet het door 9 keer te loopen en per loop een array te maken voor de rij en eentje te maken voor de kolom. Dit is de ‘occurence’ lijst, deze houdt dus bij hoe vaak een getal voor is gekomen. Dan kun je de index van de rij afleiden door de loop index keer 9 te doen. De kolom index is gelijk aan de loop index. Dan loopen we weer 9 keer met het variabele ‘l’ om voor de rij en kolom apart de occurence bij te houden. In de code doen we dit door de waarde ‘rij\_index + l’ op te halen uit de sudoku, dit geeft een waarde die dus is voorgekomen. Dat doen we min 1 om zo bij de juiste index in de occurrence lijst plus 1 te doen. Hetzelfde principe geldt ook voor de kolommen, alleen gebruiken we de waarde ‘kolom\_index + l \* 9’ om de juiste waardes van de kolom te gebruiken. Daarna kunnen we bij de ‘beginscore’ het absolute verschil tussen de occurrence waarde en 1 optellen. Als elk getal 1 keer voorkomt in de rij of kolom is de score dus 0, voor elke dubbele wordt er praktisch twee keer plus 1 gedaan. Dit is waarom we op het einde de score delen door twee.

**Heuristiek**

Om te bepalen welke swaps het beste zijn voor de HillClimb methode, gebruiken we Heuristiek. Heuristiek krijgt 2 indexen mee, dit zijn de indexen die we willen swappen. Per index moeten we de desbetreffende kolom en rij checken, aangezien dit de enige plekken zijn waar de heuristische score kan veranderen. Van beide deze rijen en kolommen berekenen we vervolgens de score voor en nadat de twee indexen van waardes wisselen. Dit doen we op dezelfde manier als uitgelegd tijdens BeginScore. Vervolgens rekenen we het verschil uit tussen de score voor en na de swap. Dit verschil wordt teruggegeven aan de aanroep.

**Swap**

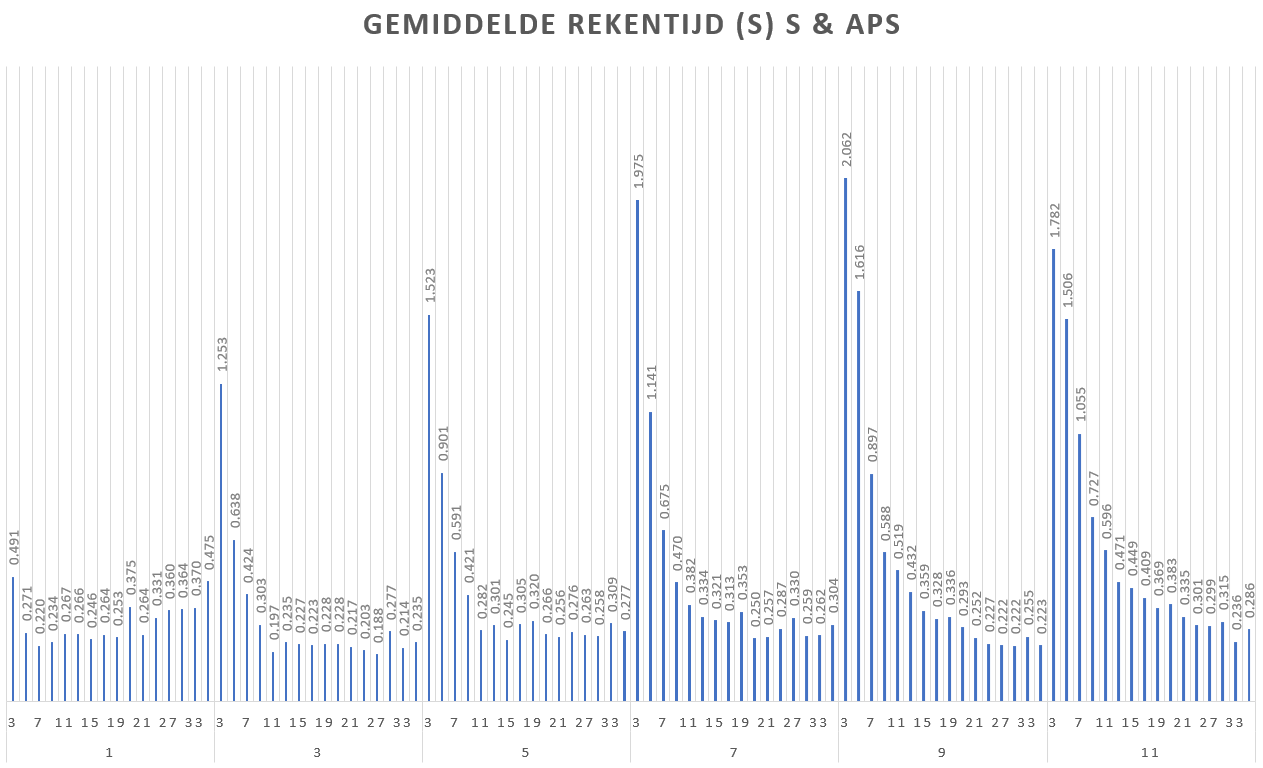
‘Swap’ vraagt als input een sudoku array en twee indices. Vervolgens wisselt hij de waardes van die twee indices als die niet ‘vast’ zijn en niet gelijk aan elkaar zijn. Er is gekozen om die voorwaarde in de swap methode te zetten omdat dat veiliger is dat je niet per ongeluk een vast getal kan omwisselen en niet een index met zichzelf poogt te wisselen. Tevens geeft de methode een boolean terug omdat we dan hillclimb makkelijk kunnen controleren of de swap succesvol was of niet.

**Overige hulp-methodes**

Als laatste zijn er nog twee echt pure hulpmethodes; ‘PrintWaarde’ en ‘PrintVast’. ‘PrintWaarde’ schrijft de values van alle sudoku vakken in een mooie representatie van een sudoku grid. ‘PrintVast’ doet hetzelfde maar dan niet met de values maar met de waardes van de variabele ‘vast’. Deze methodes waren er puur voor ons om te kijken hoe onze code werd geïmplementeerd op de sudoku. Ze zijn ook handig voor de gebruiker om een visueel beeld te krijgen van de puzzel.

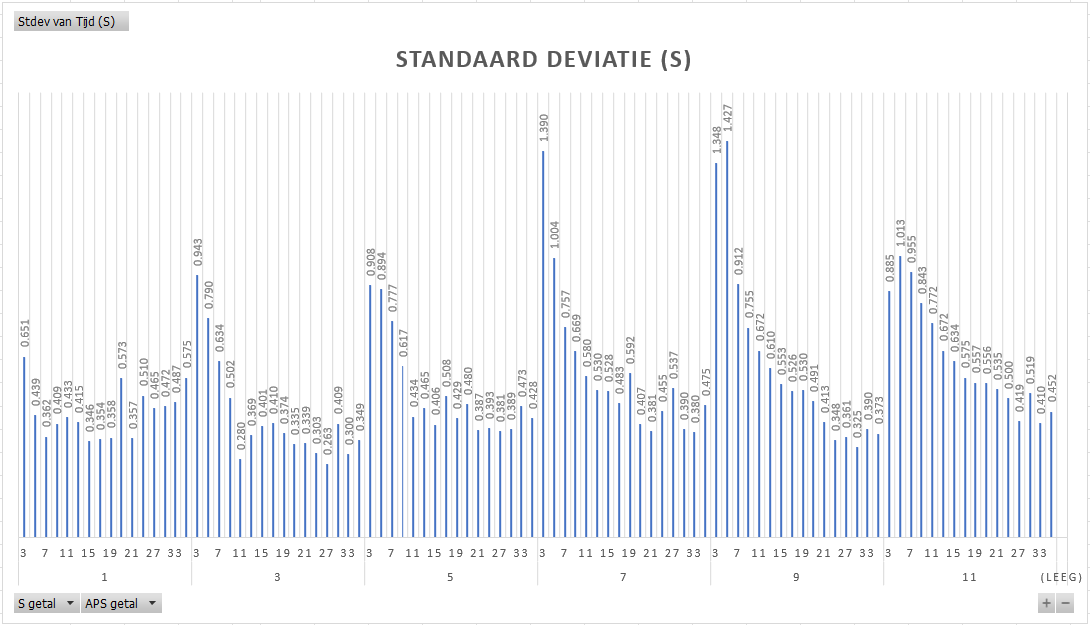
## Resultaten

We hebben een aangepaste versie van ons programma dat elke testcase met verschillende permutaties van S en APS variabelen gemaakt. Om dit niet door kans te laten beïnvloeden wordt ook elke combinatie 20 keer herhaald. Dit zijn dus de gemiddelde waardes van 20 tests per variabele. Elke grafiek over gemiddelde rekentijd is uitgedrukt in seconden (S). Dit niet dezelfde S als de variabele S. In grafiek 1 is de gemiddelde rekentijd (in seconden) uitgezet tegen alle (geteste) permutaties van S en APS.



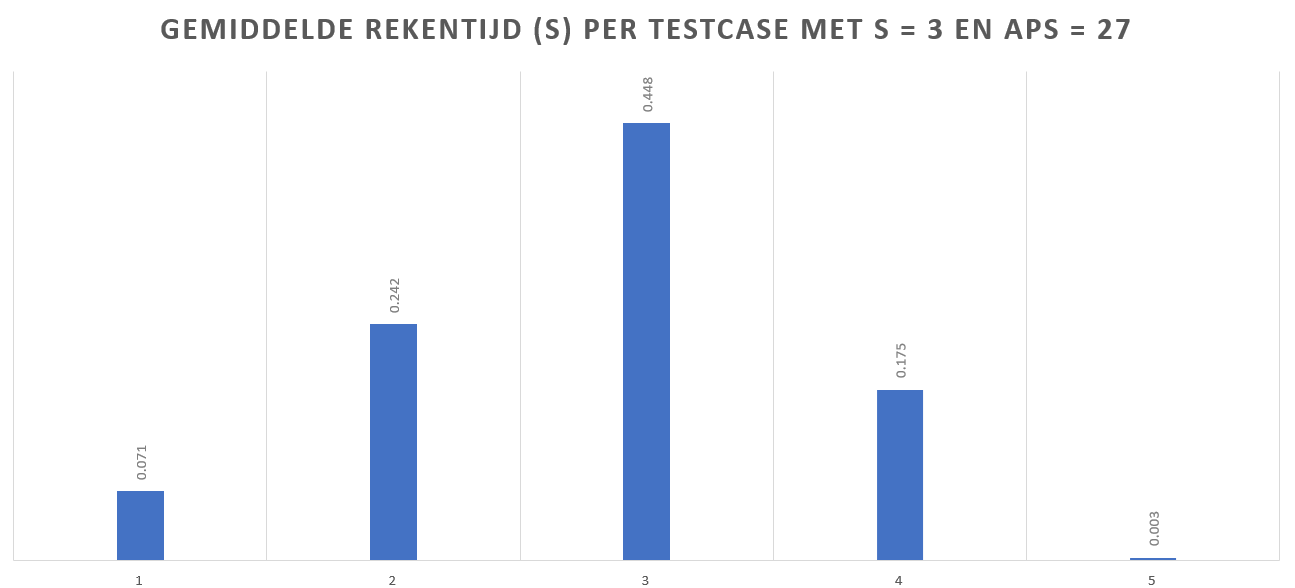
Grafiek

In grafiek 2 is de standaarddeviatie weergegeven voor elke S en APS waarde.



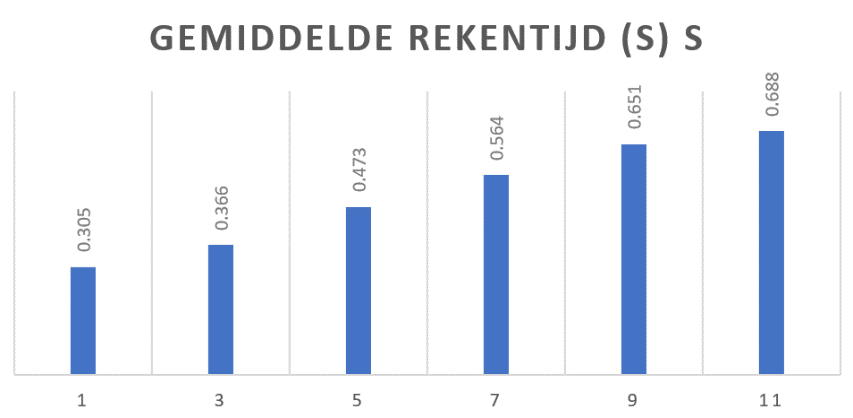
Grafiek

Met de combinatie van S (=3) en APS (=27) (verder beschreven in de conclusie) zijn alle 5 de testcases tegenover elkaar gezet in grafiek 3. Dit laat de ‘moeilijkheidsgraad’ per testcase zien.

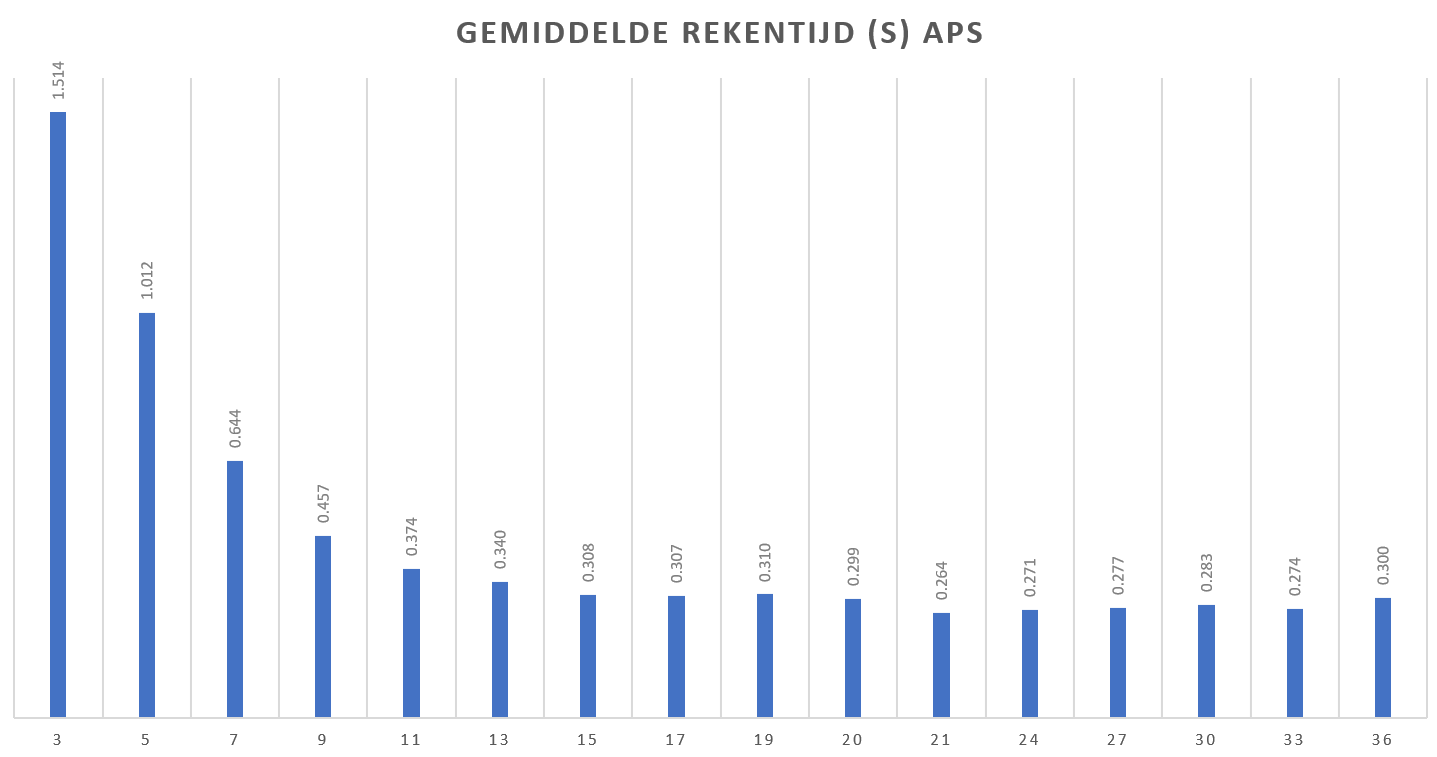


Grafiek

In grafiek 4 is de gemiddelde rekentijd tegenover S gezet. In grafiek 5 is de gemiddelde rekentijd tegenover APS gezet.



Grafiek



Grafiek

## Conclusie

Zoals te zien valt in de resultaten, zijn er duidelijke trends te zien in de waardes van de variabelen. Laten we beginnen met de APS. Het valt duidelijk te zien dat je niet te snel wil beginnen met random walk, in de resultaten wil je rond de 25 zitten. Anders gezegd betekent dit ‘als APS extreem laag is, zijn de resultaten zichtbaar slechter dan als APS hoger is’. Dit zal komen omdat je Hillclimb de kans moet geven om zeker te weten dat het op een plateau vast zit. Want anders is er een kans dat je jezelf onnodig de verkeerde kant op stuurt met random walk. Ook wil je niet te lang ronddolen, aangezien je na een bepaalde hoeveelheid iteraties toch elk blok wel al hebt gezien.

Met de S valt er te zien dat je deze juist niet al te groot wil hebben. Want vaak zijn blijkbaar 1 à 3 stappen meestal genoeg om Hillclimb uit een plateau te halen.

De variabele combinatie van S = 3 en APS = 27 geeft in totaal de kortste gemiddelde rekentijd volgens grafiek (nummer). In grafiek (nummer) is te zien dat er bij die combinatie van S en APS tevens ook de laagste standaarddeviatie is van ongeveer 0.263 seconden.

Case 3 is duidelijk gemiddeld de langzaamste en dus het moeilijkste terwijl case 5 erg makkelijk op te lossen is door ons programma volgens grafiek (nummer). Dit heeft volgens ons te maken met het feit dat er bij case 5 niet zo veel mogelijke gaten te vullen zijn.

Er is een exponentiële correlatie tussen APS en de rekentijd van de sudoku solver te zien in grafiek (nummer). Overigens is de correlatie tussen S en de rekentijd is lineair volgens grafiek (nummer).

## Discussie

We overwogen ook op een gegeven moment om nog een controle toe te voegen in de heuristische methode. Deze zou kijken of twee indexen op dezelfde rij of kolom stonden en die rij of kolom dan niet twee keer maar maar een enkele keer te controleren. Dit konden we doen door goed gebruik te maken van de modulo en absolute verschillen. Uiteindelijk hebben we dit niet geïmplementeerd omdat we het niet nodig vonden. Ook omdat we dan voor alle 2 de rijen en kolommen de volgende check moesten toevoegen wat omslachtig was:

* check kolom niet als l%9 - k%9 = 0
* check rij niet als abs(l-k) < 3

We hebben een aangepaste ‘experiment’ versie van ons programma gemaakt omdat dat veel makkelijker is om gemiddelde rekentijden te berekenen dan alles handmatig in Excel te stoppen. Dit gunde ons de mogelijkheid om per testcase per s waarde en per APS waarde 20 keer het ILS algoritme aan te roepen. Bovendien was de data die we daaruit kregen, voornamelijk het aantal seconden, makkelijk uit te lezen en te gebruiken in excel omdat we het programma de data naar een tekstbestand lieten schrijven. Onderaan de pagina staat een foto van de main methode van onze experimenten branch.

Er is gekozen om het algoritme te stoppen na maximaal 100.000 stappen omdat de meeste sudoku’s met goede variabelen wel binnen dat aantal stappen op te lossen zijn. Er zijn variabele combinaties die zo slecht zijn die de sudoku nog steeds niet kunnen oplossen binnen dat aantal stappen, en dat zullen ze waarschijnlijk ook niet kunnen binnen een telbaar aantal stappen. Daarom hebben we gekozen om ze af te kappen voordat het programma heel lang blijft hangen op een moeilijk op te lossen sudoku.

Ook al hebben we erg veel data verzamelt, is er nog steeds een redelijke kans dat dit niet volledig accuraat is, maar om voor alle variabelen van APS en S alle verschillende combinaties voor elke case te testen zodat het compleet betrouwbaar is, heb je veel meer tijd nodig dan dat wij nog beschikbaar hadden. Alsnog zijn we erg tevreden met hoe we te werk zijn gegaan met het testen.

Het verloop van het programmeren is zeer goed gegaan. We zijn er redelijk vlot en met minimale tegenslag doorheen gegaan. Er waren een paar lastige stukken waarbij we scherp moesten nadenken vooral ook omdat we het in een 1d array hebben geïmplementeerd. Ook zijn we tevreden met de efficiëntie die we hier en daar in de data hebben gezien. Het was ook zeker een leuke opdracht die echt wel te doen is.

Hieronder is de experimenten branch main methode te zien. Hier kan je zien hoe we de 20 herhalingen hebben toegepast en daaruit de gemiddeldes berekend.

