Algoritm för Sudokulösning med mänskliga strategier

ROBERT HEDIN
och JACOB RYDH

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2012

Algoritm för Sudokulösning med mänskliga strategier

R O B E R T H E D I N
o c h J A C O B R Y D H

DD143X, Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik 300 högskolepoäng
Kungliga Tekniska Högskolan år 2012
Handledare på CSC var Mikael Goldmann
Examinator var Mårten Björkman

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2012/hedin_robert_OCH_rydh_jacob_K12035.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Sammanfattning

Studien behandlar algoritmer utformade i syfte att dra nytta av mänskliga strategier, för att effektivt lösa ett Sudokuspel. Resultaten visar på viss tidseffektivitet vid mer komplexa spel, i relation till en Brute force med backtracking. Däremot var effektiviteten inte så markant att den kunde mäta sig med en SAT lösning. Vid enklare spel uppmättes inga signifikanta skillnader mellan Brute force och algoritmen baserad på mänskliga strategier.

Abstract

This paper aims to assess the effectiveness of an algorithm that is designed to used strategies that human players use when they solve a Sudoku puzzle. The results show that, for more complex puzzles, the human method algorithm performs better in terms of speed compared to Brute force with backtracking. However, the improvement was not at a level that is comparable to a SAT based solution. For simple puzzles, there was no distinct difference between the Brute force and human method algorithm.

Innehåll

1 Sudoku

Sifferpusslet Sudoku är idag ett folkkärt spel som dagligen publiceras i de stora svenska dagstidningarna.

Sudoku vann popularitet i Japan under 1980-talet när förlaget Nikoli introducerade det i Japan. Namnet Sudoku är av japansk härkomst och den svenska översättningen blir något i stil med "siffran får bara vara med en gång". Under 2000-talet har dock spelet spridit sig till västvärlden och är idag ett av de mest populära pusselspelen.

1.1 Spelets utformning

Sudodupusslet består av en kvadratisk bräda med tre gånger tre boxar där vardera box innehåller tre gånger tre rutor. Spelet har alltså totalt nio gånger nio rutor.

I början av spelet kommer ett antal av rutorna att vara ifyllda med ett nummer mellan ett och nio. Målet är att fylla i de tomma rutorna med siffror mellan ett och nio på så vis att ingen siffra återfinns mer än en gång i varje rad, kolumn och box. Det betyder att alla rader, kolumner och boxar kommer att få ett totalt siffervärde av 45 när de innehållande siffrorna adderas. De initialt utsatta siffrorna är placerade så att varje spel får en unik lösning[4].

Antalet siffror som är utplacerade i början av spelet avgör dess svårighetsgrad. Det är således minst 19 siffror utplacerade även i de svåraste spelen. Var siffrorna är utplacerade påverkar också puzzlets svårighetsgrad. De flesta pussel går att lösa rent analytiskt genom att studera de siffror som redan är utplacerade och med hjälp av en uteslutningsmetod komma fram till vad en viss ruta ska ha för nummer.

2 Bakgrund

2.1 Tidigare forskning

Då Sudoku är ett populärt spel och ett spel som är NP-svårt[11], finns det som förväntat redan många artiklar som beskriver möjliga sätt att lösa spelet. Felgenhauer och Jarvis (2006) beskriver matematiken bakom ett Sudoku samt fastställer problemets storlek genom att beräkna antalet unika pussel som kan skapas med en 9 x 9 stor spelplan. Det fastslogs att $6.671 \cdot 10^{21}$ olika unika pussel finns[5][6].

Weber (2005) beskriver hur ett Sudoku kan lösas med en SAT lösare. En liknande lösning föreslogs även av Lynce och Ouaknine (2006)[10][8].

Mantere och Koljonen (2006) visade att Sudokupussel även kan lösas med Genetisk programmering men de kom fram till att det inte var en effektiv lösningsmetod och föreslår istället en Brute force algoritm med backtracking[9].

Det finns ett flertal strategier som mänskliga spelare använder sig av för att lösa Sudokupussel. Vaderlind (2005) presenterar följande strategier[2]:

1. Arbeta med listor. Skriv i varje ruta ned en lista med möjliga kandidattal.

2. Lista med enskilt kandidattal. Identifiera rutor vars kandidatlistor består av ett enskilt tal och placera ut talet. Genom att i varje tom ruta skriva ned de siffror som är möjliga att placera kan dels de rutor som enbart innehåller en kandidatsiffra identifieras. Dessa kan direkt placeras ut då man med säkerhet vet att det är rätt alternativ. Efter utplaceringen uppdaterar spelaren listorna så att de inte innehåller några ogiltiga alternativ.

3. Lista med ett unikt kandidattal. Om det i en rad, kolumn eller box finns en ruta som har ett kandidattal som återfinns unikt för denna ruta så ska det kandidattalet placeras ut i denna ruta.

4. Bundna par. Med hjälp av listorna går det även att identifiera bundna par, vilket är två värden som bara kan finnas i två specifika rutor, men där det inte går att avgöra vilken som ska placeras var. Eftersom man vet att siffrorna måste placeras i något av de två givna alternativen går det att stryka dem från alla övriga rutornas kandidatlista.

5. Bundna tripplar. I likhet med regel två kan bundna tripplar identifieras, vilket uppstår då tre rutor vars kandidatlista enbart består av tal tagna av en mängd innehållandes tre tal.

6. Rader versus boxar. Om det finns en rad där ett antal kandidattal enbart återfinns i rutor som tillhör samma box så går det att i den boxen stryka dessa tal från de övriga rutorna i kandidatlista om de ligger i den boxen.

7. Gissning. När inga andra metoder fungerar kan spelaren tvingas chansa på en lösning. Välj helst en ruta med så få kandidattal som möjligt och lägg chansningen i en av dem.

3 Frågeställning

Det är stora skillnader mellan de algoritmer som används av datorer och de strategier mänskliga spelare nyttjar för att lösa Sudokuspel. Det är möjligt att en kombination av de båda kan leda till algoritmer som mer effektivt löser spelen. Den frågeställning som denna studie ska undersöka är således:

Kan implementering av Vaderlinds (2005) strategi 1-4, som är ämnade för mänskliga spelare, göra Brute force algoritmen mer tidseffektiv?

4 Metod

För att mäta effekten av den mänskliga algoritmen görs mätningar mot en Brute force samt SAT algoritm. Anledningen till att mätningen även sker med en SAT algoritm är för att se om effektförhöjningen är så signifikant att den får en tidseffektivitet jämförbar med SAT algoritmen.

4.1 Brute force

För Sudokuproblemet blir det en algoritm som hela tiden chansar på ett nummer tills den får en lösning som stämmer. Eftersom en Sudokubräda kan fyllas i på approximativt $6.7 \cdot 10^{21}$ olika sätt riskerar man att lösningen blir väldigt långsam i jämförelse med en mer uttänkt algoritm.

Brute force algoritmen som används vid mätningen börjar med att leta fram rutor som inte är ifyllda. Den börjar leta längst upp till vänster: om rutan är ifylld tar den ett steg till höger. Om alla rutor i raden är ifyllda hoppar den ned en rad och börjar igen leta från vänster till höger. När algoritmen hittar en ruta som inte är ifylld söker den efter möjliga kandidatsiffror. Algoritmen kommer att gå igenom alla rader, kolumner och boxar som rutan befinner sig i för att fastställa vilka siffror som är giltiga kandidater. Algoritmen kommer att välja det lägsta nummer som är tillgängligt och placera ut det. När siffran är utplacerad börjar algoritmen om och börjar söka efter nästa tomma ruta. Om en ruta är tom men saknar kandidatsiffror har en chansning blivit fel. Algoritmen går då ett steg tillbaka och byter den sist utlagda siffran mot en ny kandidatsiffra. Om det i detta läge inte finns någon giltig siffra tar den ytterligare ett steg tillbaka. I värsta fall måste den gå tillbaka till den först utlagda siffran[1]. Nedan presenteras psudokod för Brute force algoritmen.

Algorithm 1 Solve($matrix[n][n]$)

```
x ← 0
y ← 0
found ← False
while x < n do
    while y < n do
        if matrix[x][y] = 0 then
            found ← True
            break
        end if
        y ← y + 1
    end while
    x ← x + 1
end while
if Found = False then return True
end if

list[n] ← getPossibleNumber
    ▷ Denna funktion returnerar en lista av möjliga värden för vilket en position (x,y) kan besitta
i ← 0
s ← list.size()
while i < s do
    number ← list[i]
    matrix[x][y] ← number
    if Solve(matrix) then return True
    end if
    matrix[x][y] ← 0
    i ← i + 1
end while

return False
```

4.2 Mänskliga Metoden

Till skillnad från Brute force algoritmen kommer den modifierade versionen att innehålla ett antal olika metoder för att placera ut siffror under programmets körtid. Dessa metoder är baserade på hur en människa skulle lösa ett Sudokupussel. Idén är att se om de strategier mänskliga spelare använder sig av för att så snabbt som möjligt lösa ett givet spel även kommer att förbättra körtiden för Brute force algoritmen. Reglerna är implementerade så att de körs i inkrementerande ordning och skulle en regel appliceras börjar programmet om från steg 1. Anledningen till att algoritmen är utformat på detta sätt är att den hela tiden ska utföra den enklaste regeln först och sedan arbeta sig ned till det mer komplicerade och tidskrävande reglerna. Den absolut sista regeln som skall appliceras är chansning vilket fungerar som Brute forcen[1].

Steg 1 Programmet börjar med att tilldela varje position i brädet en lista med alla möjliga kandidattal. Det som här skiljer sig mot Brute force algoritmen är att alla listor sparas. Varje ruta kommer alltså att ha en egen lista innehållandes kandidattal istället för att enbart ha en lista för den aktuella rutan. Det möjliggör att de övriga strategierna kan appliceras då de bygger på att alla rutors kandidatlistor är tillgängliga. Detta steg bygger på strategi 1 av Vaderlinds (2005).

Steg 2 När alla listor är skapade undersöks de för att se om det finns en ruta som bara har ett enda kandidattal och om en sådan ruta upptäcks placeras dess kandidattal och algoritmen går tillbaka till steg 1. Detta steg är baserat på strategi 2. Om ingen sådan ruta upptäcks går algoritmen vidare till steg 3.

Steg 3 Iterera över alla rader, kolumner och boxar för att söka efter ett kandidattal som tillhör endast en ruta. Om ett sådant kandidattal hittas placeras det ut och algoritmen börjar om från steg 1. Hittas inget sådant kandidattal går algoritmen vidare till steg 4.

Steg 4 Denna regel går ut på att iterera över alla rader, kolumner och boxar för att finna två rutor som vardera har två kandidattal och där de två kandidattalen är identiska. Om två sådana rutor hittas vet man att kandidattalen måste placeras i dessa två rutorna. Däremot går det inte att fastställa vilken av de två rutorna som ska ha vilket kandidattal. Dock kan de uteslutas som kandidattal till alla rutor i samma i rad, kolumn eller box som de två rutorna ligger i. Om algoritmen gör en sådan reduktion går den tillbaka till steg 1, annars fortsätter den till steg 5.

Steg 5 Om algoritmen kommit till steg 5 går den igenom alla rutor för att se om det finns någon tom ruta. Om alla rutor är ifyllda är pusslet löst men om det finns tomma rutor så kommer algoritmen att chansa på en av rutans kandidattal. Innan chansningen placeras ut sparas en kopia av spelet som innehåller dels rutornas värde och även dess kandidatlistor. Precis som med den vanliga Brute force algoritmen besöks den första tomma rutan och chansningen görs på det lägsta tillgängliga

kandidattalet. Alla steg kommer fortsättningsvis enbart att appliceras på kopian. Efter utplacering börjar algoritmen om från steg 1 och fortsätter som tidigare, fast på kopian. Om det uppstår ett läge där det i steg 5 finns en tom ruta som saknar kandidattal börjar algoritmen om från den senast tagna kopian och utför en ny chansning. Algoritmen kommer att upprepa Steg 1 till 5 tills den löst hela spelet och returnerar då lösningen. Nedan presenteras pseudokod till algoritmen.

Algorithm 2 Solve($matrix[n][n]$)

```
canApplyRule ← True
stepOne(matrix[][])
repeat          ▷ Anropen för de enskilda strategierna utförs på en referens till matrix
    if stepTwo(matrix[][]) then
        repeat
    else if stepThree(matrix[][]) then
        repeat
    else if stepFour(matrix[][]) then
        canApplyRule ← False
    end if
until canApplyRule = False
x ← 0
y ← 0
found ← False
while x < n do
    while y < n do
        if matrix[x][y] = 0 then
            found ← True
            break
        end if
        y ← y + 1
    end while
    x ← x + 1
end while
if found = False then return True
end if

list[n] ← getPossibleNumber
    ▷ Denna funktion returnerar en lista av möjliga värden för vilket en position (x,y)
kan besitta
i ← 0
s ← list.size()
while i < s do
    number ← list[i]
    matrix[x][y] ← number
    if Solve(matrix) then return True
    end if
    matrix[x][y] ← 0
    i ← i + 1
end while

return False
```

4.3 SAT lösare

SAT lösare eller satisfiabillity solver är en algoritm som undersöker om en given mängd boliska variabler kan tilldelas på sådan sätt att hela uttrycket utvärderas till sant. För att lösa SAT problemet använder vi MiniSAT 2[7]. Det logiska uttrycket måste skrivas på Konjuktiv normalform för att MiniSat 2 ska hantera problemet. MiniSat läser in data från fil, där varje siffra tolkas som en variabel, minustecknet som en negation och en nolla signalerar att klausulen är slut. Mellan varje klausul införs ett $\vee$ för att binda samman alla rader till ett uttryck. När programmet exekverats kommer den först returnera huruvida den lyckades tilldela variablerna på ett sådant sätt att hela uttrycket evalueras till sant. Den skapar även en fil där den listar alla variabler samt den tilldelning den gör.

För att kunna använda MiniSat2 görs först en reduktion där Sudokuproblemet utformas som ett SAT problem på konjuktiv normalform. Nedan presenteras den reduktion som används[10].

Låt x representera den horisontella positionen och y representera den vertikala positionen på en Sudokubräda. Variablen x kan tilldelas alla positioner mellan 1 och 9 där 1 avser rutor som ligger i kolumnen längst till vänster och 9 kolumnen längst till höger. Variabeln y antar positioner mellan 1 och 9 där 1 är rutor på den översta raden och 9 rutor på den understa raden. Variabeln z representerar vilket värde en viss ruta har. Med hjälp av x, y, z skapas variabeln S_{xyz} som är en boolesk variabel och om den är tilldelad 1 så har rutan vid position $[x, y]$ värdet z och är den noll har den inte värdet z. Eftersom det finns 81 rutor och varje ruta kan ha 9 olika värden, resulterar det i $81 \cdot 9$, det till säga, 729 booleska variabler. Följande formel används vid namngivning av de olika variablerna.

$$S_{xyx} = 81 \cdot (x-1) + 9 \cdot (y-1) + (z-1) + 1 \tag{1}$$

Högerledet på ekvation 1 är namnet på den booleska variabeln inte en tilldelning.

Fem restriktioner införs som tillsammans ser till att variablerna tilldelas på ett sådant sätt att de löser Sudokuspelet.

Restriktion 1: Det finns minst ett nummer i varje ruta.

$\wedge_{x=1}^{9} \wedge_{y=1}^{9} \vee_{z=1}^{9} s_{xyz}$

Restriktion 2: Varje nummer får bara finnas en gång per rad

$\wedge_{y=1}^{9} \wedge_{z=1}^{9} \wedge_{x=1}^{8} \wedge_{i=x+1}^{9} (\neg s_{iyz} \vee \neg s_{xyz})$

Restriktion 3: Varje nummer får bara förekomma en gång per kolumn

$\wedge_{x=1}^{9} \wedge_{z=1}^{9} \wedge_{y=1}^{8} \wedge_{i=y+1}^{9} (\neg s_{xyz} \vee \neg s_{xiz})$

Restriktion 4: Varje nummer får bara förekomma en gång per 3x3 box

$\wedge_{z=1}^{9} \wedge_{i=0}^{2} \wedge_{j=0}^{2} \wedge_{x=1}^{3} \wedge_{y=1}^{3} \wedge_{k=y+1}^{3} (\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+x)(3j+k)z})$

$\wedge_{z=1}^{9} \wedge_{i=0}^{2} \wedge_{j=0}^{2} \wedge_{x=1}^{3} \wedge_{y=1}^{3} \wedge_{k=x+1}^{3} \wedge_{l=x}^{3} (\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+k)(3j+l)z})$

Eftersom vissa rutor kommer vara ifyllda i början av spelet läggs även dessa in som restriktioner.

Restriktion 5: Givna startvärden

S_{xyx} där S_{xyx} är en ruta som är ifylld med värde z vid spelets start.

4.4 Testförfarande

Sudokuspelen skapas med hjälp av en Sudokugenerator. Den skapar Sudokun genom att lösa ett tomt Sudoku med en Brute force. För att spel ska slumpas fram kommer denna Brute force välja kandidattal helt slumpmässigt istället för att som den vanliga metoden ta det lägsta tillgängliga talet. När ett korrekt Sudoku har skapats kommer generatorn sedan att ta bort ett förbestämt antal siffror på slumpade positioner från det lösta spelet, vilket resulterar i ett spel som är olöst men lösbart. Eftersom kandidatnycklarna slumpas fram kommer ingen av algoritmerna få någon fördel gentemot de andra.

Inför testningen kommer det att skapas 20 000 spelplaner fördelade på 20 nivåer där nivån bestäms av hur många initialt utsatta siffror spelplanen har. Nivåerna går från 20 upp till 40 utplacerade siffror.

Det kommer även att utföras testning där olika kombinationer av strategierna används för att avgöra vilken kombination som ger bäst resultat. Alla tester kommer dock att innehålla strategi 1 då den är nödvändig för att de övriga strategierna ska fungera. Det kommer utföras 20 000 test jämnt fördelade på de 20 nivåerna.

För att få en tydligare bild av hur väl algoritmerna presterar kommer det även genereras spel som är av storleken 16 x 16. Spel av den här storleken kallas Super Sudoku och är betydligt svårare att lösa. Det finns sådana spel presenterade i bland annat *Paul Vaderlinds stora Sudokubok'*(2005)[2]. På brädor av denna storlek kommer 2000 tester att utföras jämnt fördelade på de olika nivåerna, där nivåerna varierar mellan 60 förutsatta upp till 117 förutsatta siffror.

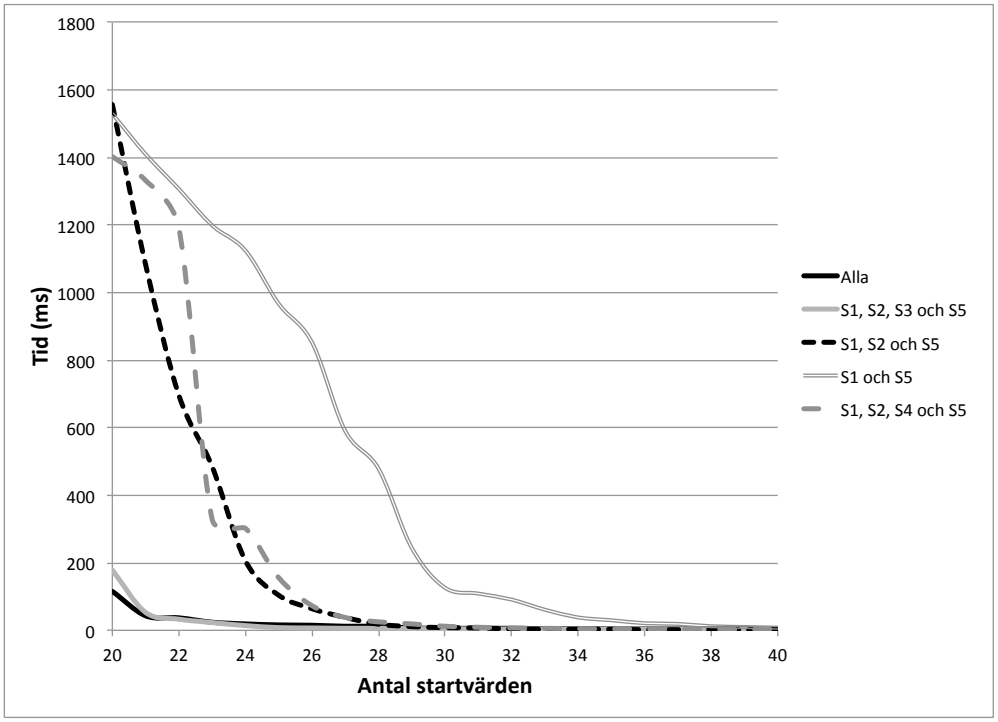
Samtliga tester kommer att genomföras i samma datormiljö för att risken att stöta på feldata ska minimeras.

5 Resultat

5.1 Mätdata för olika kombinationer av mänskliga strategier

Figur 1 nedan visar medelvärdet i millisekunder på varje körning när olika kombinationer av strategierna används.

Figur 1: Graf över tiderna med en matrisstorlek på 9x9 rutor då olika strategier utförs separat.

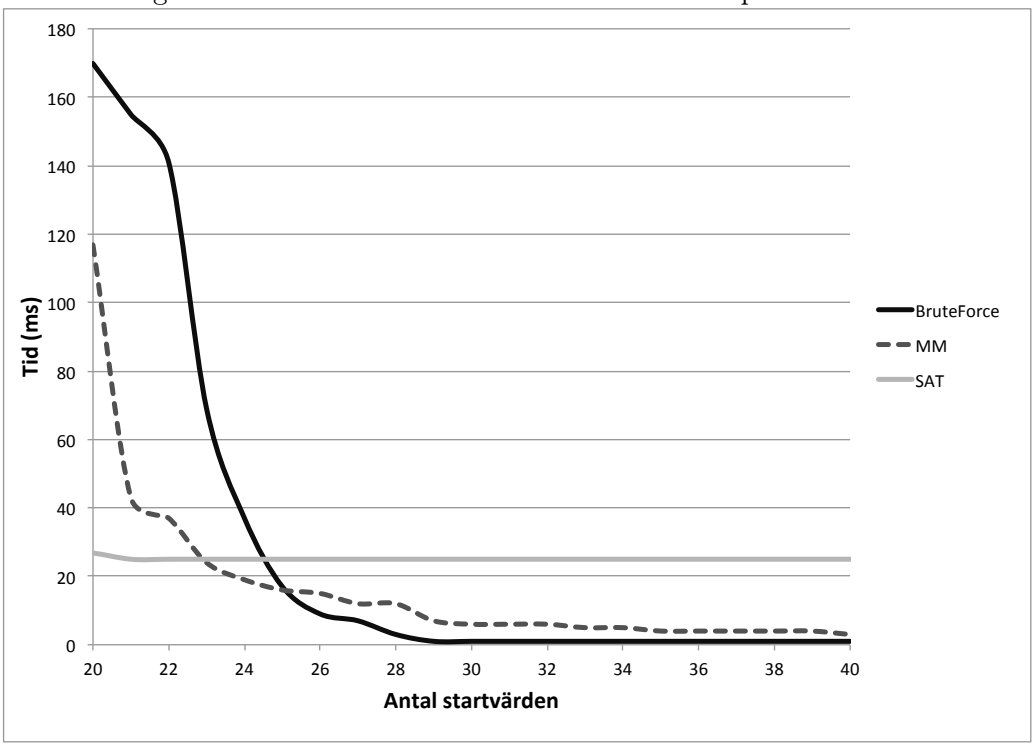


Efter 23 utlagda startvärden presterar den mänskliga algoritmen där strategi 4 är exkluderad bättre än när algoritmen utnyttjar alla strategier. De övriga kombinationerna av strategier presterar sämre än när algoritmen använder sig av alla strategier.

5.2 $9x9$ Sudokubräden

Figur 2 nedan visar medelvärdet i millisekunder på varje nivå för respektive algoritm för den klassiska 9 x 9 Sudokubrädan.

Figur 2: Graf över tiderna med en matrisstorlek på 9x9 rutor.

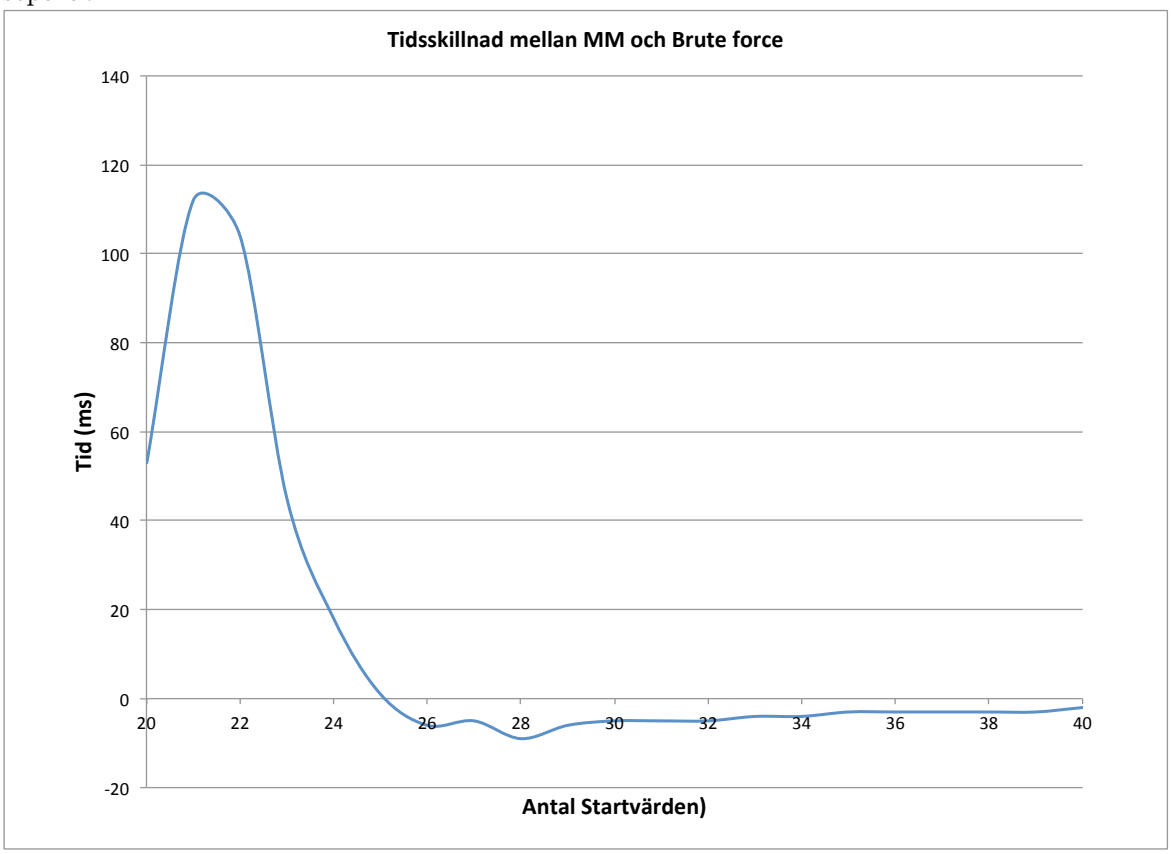


När det är fler än 25 utlagda startvärden blir Brute force snabbare än den mänskliga algoritmen. Skillnaden mellan den enklaste nivån och den svåraste nivån ger för SAT lösaren en skillnad på två millisekunder. Från och med 21 utlagda startvärden till och med 40 uppmäts ingen förändring i tiden.

5.3 Differens mellan mänsklig algoritm och Brute force för $9x9$ Sudokubräden

Figur 3 nedan visar differensen mellan den mänskliga algoritmen och Brute force för Sudokupussel av storleken $9x9$.

Figur 3: Graf över tiderna med en matrisstorlek på 9x9 rutor då olika strategier utförs separat.

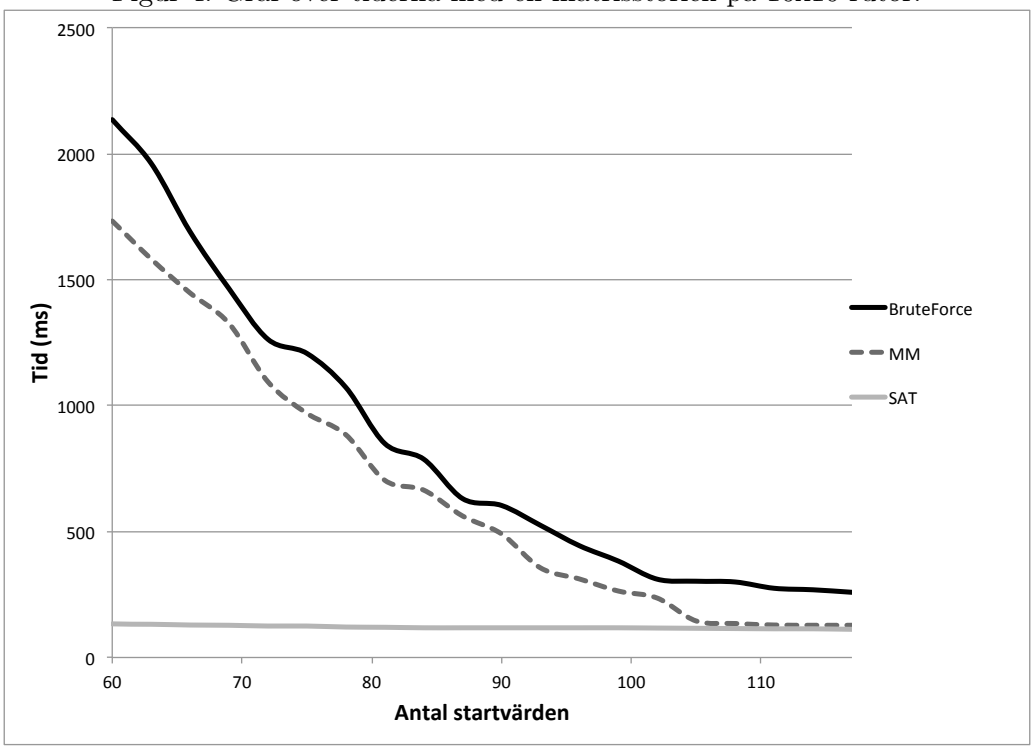


Denna graf är en förtydligande över skillnaden mellan Brute force och den mänskliga algoritmen. Den största tidsskillnaden sker vid 22 utlagda startvärden

5.4 $16x16$ Sudokubräden

Figur 4 nedan visar medelvärdet i millisekunder på varje nivå för respektive algoritm för 16 x 16 Super Sudokubrädan.

Figur 4: Graf över tiderna med en matrisstorlek på 16x16 rutor.



Den mänskliga algoritmen uppvisar snabbare resultat än Brute forcen, dock visar SAT lösaren ett bättre resultat vid alla tester på $16x16$ brädor.

6 Diskussion

6.1 Mätdata då olika strategier utförs separat

Det är nästan ingen skillnad mellan den fullständiga algoritmen då alla strategier utnyttjas jämfört med när strategi 4 är exkluderad. En anledning till att varför dessa två metoder skiljer sig så pass lite kan förklaras med att strategi 4 används sparsamt.

Ensamt ger strategi 1 svagt resultat eftersom det skapas en större mängd listor som aldrig används då enbart listan för den aktuella rutan nyttjas. Det åtgår tid att uppdatera listor som inte nyttjas.

Anledningen till varför strategierna inte fungerar väl individuellt kan förklaras med att algoritmen tvingas till att chansa för många gånger.

6.2 $9x9$ Sudokubräden

Den mänskliga algoritmen presterade bättre på svårare Sudokupussel än Brute force algoritmen. När få rutor innehåller ett värde kan det ta lång tid innan Brute force algoritmen inser att en felgissning har inträffat, och därmed blir chansningarna mer kostsamma. Den måste då arbeta sig tillbaka till den position där felgissningen inträffade, där tiden ökar exponentiellt med antalet steg från felgissningen.

Ju färre utlagda siffror det finns desto fler gissningsalternativ kommer att existera vid varje ruta vilket medför att sannolikheten att gissa rätt blir mindre.

Den mänskliga algoritmen vinner tid på att den kommer att kunna finna fler säkra alternativ vilket leder att till algoritmen inte behöver chansa lika ofta.

Hos pussel som innehåller fler än 25 utlagda startvärden blir skillnaden mellan den mänskliga algoritmen och Brute force algoritmen minimal. En förklaring till detta kan vara att vinsten den mänskliga algoritmen får genom att göra färre gissningar kvittas mot kostnaden det tar för algoritmen att utföra strategi 1 till 4. Tidskostnaden för att gå igenom strategi 1 till 4 är i princip konstant medan Brute force algoritmen är starkt beroende av antalet utlagda siffror.

SAT algoritmen löser Sudokuproblemen på närmast konstant tid. Att den får en högre tid i förhållande till den mänskliga algoritmen för de lättare pusslen, beror på att MiniSat 2 använder sig av filskrivning. Då tiden för filskrivningen vid reduktionen i princip är oberoende av hur många initialt utsatta siffror spelet har, är det troligt att en stor del av lösningstiden kommer från just filskrivningen.

6.3 Differens mellan mänsklig algoritm och Brute force för $9x9$ Sudokubräden

Skillnaden är som störst vid 22 utlagda startvärden. Anledningen till att den är större vid 22 än vid 20 kan antas bero på att det vid 20 utlagda är så pass få att den mänskliga algoritmen tvingas till att chansa i ett tidigt stadium. Varje gång den mänskliga algoritmen tvingas till att chansa blir det steget mer kostsamt än för Brute force algoritmen eftersom mänskliga algoritmen börjar med att gå igenom strategi 1 till 4 innan den utför en chansning.

6.4 $16x16$ Sudokubräden

Den mänskliga algoritmen presterar bättre än Brute force algoritmen vid samtliga tester. En rimlig förklaring är att samtliga testfall blir mer krävande än de för $9x9$ brädan. Det stödjer vårt tidigare antagande om att den mänskliga algoritmen är att föredra vid de mer krävande spelen. Vid testerna ser vi att SAT algoritmen presterar betydligt bättre än de övriga två. Ett rimligt antagande blir att filskrivningen får mindre inverkan på algoritmens prestation i förhållande till de övriga ju mer krävande spelet är.

6.5 Övergripande diskussion

Tillämpning av Vaderlinds (2005) strategier[2] för mänskliga spelare visar sig ge viss tidseffektivisering i relation till Brute force algoritmen vid svårare spelinstanser. Den uppvisar dock inte ett resultat som kan mäta sig med SAT algoritmen.

I förhållande till hur mycket mänskliga strategier kan förbättra effektiviteten för en mänsklig spelare har de uppvisat relativt små förbättring jämfört med en Brute force. En möjlig förklaring till det kan vara att de mänskliga strategierna används på ett annat sätt av en mänsklig spelare. Mänskliga spelare får direkt en översiktsbild av sudokuplanen och kan således välja ut ett antal rutor som de anser har störst chans att lösa. Den datorimplementerade versionen av dessa strategier blir klumpigare eftersom den planlöst stegrar sig igenom ruta för ruta. Mänskliga spelare kan också antas välj den för situationen lämpliga strategin och inte såsom implementeringen gå igenom strategi för strategi i en förutbestämd ordning.

Brute force algoritmen är utformad så att den inte helt planlöst chansar på olika siffror när den försöker lösa pusslet. Den kommer se till att alla siffror den placera ut inte bryter mot någon att spelet regler. Det gör att hastigheten förbättras markant i förhållande till en helt naiv lösning och skillnaden mellan den mänskliga algoritmen och Brute force blir därmed mindre.

Eftersom strategierna är utformade för mänskliga spelare är det naturligt att chansningar tas till när inga andra alternativ finns, då det dels blir väldigt tidskrävande i många lägen, dels för att det kan anses mindre intellektuellt stimulerande att lösa ett

pussel med ett stort antal chansningar.

Skapande av sudokuspel som i denna studie sker med hjälp av sudokugeneratorn, tar inte hänsyn till huruvida pusslet kan lösas utan chansningar eller inte. Detta medför att de sudokuspel som används i detta test kan skilja sig från de spel som exempelvis publiceras i dagstidningar då de flesta mänskliga spelare föredrar pussel där de inte tvingas att chansa. De strategier som rekommenderas till människor är dock utformade utifrån att lösningar ska gå att nå utan chansningar, vilket inte nödvändigtvis är optimalt för spelplanerna i detta test. Om alla spel hade tagits från till exempel dagstidningar är det alltså möjligt att den mänskliga algoritmen hade större prestationsförbättringar än vad den ger med spel från Sudokugeneratorn.

För att emulera den mänskliga spelaren har chansningen lagts som en sista utväg, vilket i vissa fall inte är den mest tidseffektiva lösningen för datorer men det överensstämmer med Vanderlinds (2005)rekommendationer.

Referenser

[1] Kleinberg, J., Tardos, É. (2006) *Algorithm Design*, Publicerad i Boston: Pearson Education Inc.

[2] Vaderlinds, P. (2005) *Paul Vaderlinds stor sudokubok*, Publicerad i Sverige: Telegram Bokförlag.

[3] Pang, S., Li, E., Song, T., Zhang, P. (2010) *Rating and Generating Sudoku Puzzles*, ur: ETCS (Education Technology and Computer Science), Qingdao, Kina, 6-7 Mars 2010, Publicerad i Wuhan: Shandong Univ. of Sci. and Technol.

[4] World Sudoku Federation, *'World Sudoku Federation'*, Tillgänglig på <http://worldsudokufederation.org/>, [Hämtad 24 Mars, 2012]

[5] Felgenhauer, B., Jarvis, F. (2006) *'Mathematics of Sudoku I'*, Tillgänglig på <http://www.afjarvis.staff.shef.ac.uk/sudoku/felgenhauer_jarvis_spec1.pdf>, [Hämtad 24 Mars, 2012]

[6] Felgenhauer, B., Jarvis, F. (2006) *'Mathematics of Sudoku II'*, Tillgänglig på <http://www.afjarvis.staff.shef.ac.uk/maths/russell_jarvis_spec2.pdf>, [Hämtad 24 Mars, 2012]

[7] Eén, N., Sörensson, N. (2005) *'The MiniSat Page'*, Tillgänglig på <http://minisat.se/>, [Hämtad 25 Mars, 2012]

[8] Weber, T. (2005) *'A SAT-based Sudoku Solver'*, Tillgänglig på <http://my-svn.assembla.com/svn/disenosudoku/docs/a-sat-based-sudoku-solver.pdf>, [Hämtad 25 Mars, 2012]

[9] Mantere, T., Koljonen, J. (2006) *'Solving, Rating and Generating Sudoku Puzzles with GA'*, Tillgänglig på <http://www.cs.york.ac.uk/rts/docs/CEC-2007/html/pdf/1445.pdf>, [Hämtad 18 Mars, 2012]

[10] Lynce, I., Ouaknine, J., AIMATH (Artificial Intelligence and Mathematics), 2006, *'Sudoku as a SAT Problem'*, Fort Lauderdale, Florida, USA, Januari 4-6, 2006, Springer.

[11] Yato, T. (2003) *'Complexity and completeness of finding another solution and its application to puzzles'*, Publicerad i Joho Shori Gakkai Kenkyu Hokoku, 2002(103), s.9-16.

A Mätdata

A.1 Differens mellan mänsklig algoritm och Brute force för $9x9$ Sudokubräden

Antal startvärden	Differens mellan mänskliga algoritmen och Brute force (ms)
20	53
21	112
22	104
23	45
24	18
25	1
26	-6
27	-5
28	-9
29	-6
30	-5
31	-5
32	-5
33	-4
34	-4
35	-3
36	-3
37	-3
38	-3
39	-3
40	-2

A.2 $9x9$ Sudokubräden

Antal startvärden	BruteForce (ms)	HumanBruteForce (ms)	SAT (ms)
20	170	117	27
21	155	43	25
22	141	37	25
23	69	24	25
24	37	19	25
25	17	16	25
26	9	15	25
27	7	12	25
28	3	12	25
29	1	7	25
30	1	6	25
31	1	6	25
32	1	6	25
33	1	5	25
34	1	5	25
35	1	4	25
36	1	4	25
37	1	4	25
38	1	4	25
39	1	4	25
40	1	3	25

A.3 Mätdata då olika strategier utförs separat

Antal startvärden	Alla (ms)	S1 (ms)	S1 och S2 (ms)	S1, S2 och S3 (ms)	S1, S2 och S4 (ms)
20	117	1530	1555	178	1403
21	43	1410	1083	53	1332
22	37	1307	696	33	1191
23	24	1199	485	23	329
24	19	1125	206	14	302
25	16	967	104	8	156
26	15	853	64	7	73
27	12	590	38	6	38
28	12	479	18	6	25
29	7	244	11	5	19
30	6	126	8	5	12
31	6	109	7	4	9
32	6	91	4	4	7
33	5	61	3	4	6
34	5	38	3	4	5
35	4	30	3	4	4
36	4	21	2	3	4
37	4	19	2	3	3
38	4	12	2	3	3
39	4	10	2	3	3
40	3	8	2	3	3

A.4 $16x16$ Sudokubräden

Antal startvärden	BruteForce (ms)	HumanBruteForce (ms)	SAT (ms)
60	2135	1735	132
63	1963	1583	132
66	1691	1448	129
69	1465	1327	128
72	1264	1095	125
75	1208	970	125
78	1073	885	121
81	850	704	120
84	788	665	118
87	631	562	118
90	604	491	118
93	525	356	118
96	444	312	118
99	383	264	118
102	311	236	117
105	303	145	116
108	300	135	115
111	275	129	114
114	269	128	114
117	259	128	112

B Källkod

B.1 Brute-force

```
public class BigBruteForce {

    public boolean solve(int [][] matrix) {
        int size =matrix.length;
        int x=0,y=0;
        boolean found = false;

        for(x = 0;x < size; x ++) {
            for(y = 0;y < size; y++) {
                if(matrix[x][y] == 0) {
                    found = true;
                    break;
                }
            }
            if( found ) break;
        }

        if(!found) {
            return true;
        }

        boolean digits[] = new boolean[size+1];

        for(int i = 0; i < size; i++) {
            digits[matrix[x][i]] = true;
            digits[matrix[i][y]] = true;
        }

        int s = (int) Math.sqrt(size);
        int bx = s * (x/s), by = s * (y/s);
        for(int i =0;i<s;i++)
            for(int j = 0; j < s; j++)
                digits[matrix[bx+i][by+j]] = true;

        for(int i = 1 ; i <= size; i++) {
            if(!digits[i]) {
                matrix[x][y] = i;
                if(solve(matrix)) {
                    return true;
                }
                matrix[x][y] = 0;
            }
        }
        return false;
    }
}
```

B.2 SAT

```
import java.io.*;

public class SAT {
        public void reduceToSAT(int[][] matrix) {
                int size = matrix.length;
                int sqrtSize = (int)Math.sqrt(size);
                try{

        FileWriter fstream = new FileWriter("infil.txt");
        BufferedWriter out = new BufferedWriter(fstream);

        int extra = 0;
        for(int x = 0; x < size; x++) {
            for(int y = 0; y < size; y++) {
                if(matrix[x][y] != 0) {
                    extra++;
                }
            }
        }
        // 9 7209
        // 16 78976
        // 25 495625

        out.write("p cnf "+(size*size*size) +" "+(78976+extra) + "\n");

        for(int x = 0; x < size; x++) {
            for(int y = 0; y < size; y++) {
                if(matrix[x][y] != 0) {
                    int f =  ((size*size)*x) + (size*y) + matrix[x][y];
                    out.write(f + " 0\n");
                }
            }
        }

        for(int x = 0; x < size; x++) {
            for(int y = 0; y < size; y++) {
                for(int z = 0; z < size; z++) {
                    int f =  ((size*size)*x) + (size*y) + z + 1;
                    out.write(f + " ");
                }
                out.write("0\n");
            }
        }

        for(int x = 0; x < size; x++) {
            for(int z = 0; z < size; z++) {
                for(int y = 0; y < (size-1); y++) {
                    for(int i = y+1; i < size; i++) {
                        int s1 = -((size*size)*x+size*y+z+1);
                        int s2 = -((size*size)*x+size*i+z+1);
                        out.write(s1 + " " + s2 + " 0\n");
```

```
                }
            }
        }
    }

    for(int y = 0; y < size; y++) {
        for(int z = 0; z < size; z++) {
            for(int x = 0; x < (size-1); x++) {
                for(int i = x+1; i < size; i++) {
                    int s1 = -((size*size)*x+size*y+z+1);
                    int s2 = -((size*size)*i+size*y+z+1);
                    out.write(s1 + " " + s2 + "  0\n");
                }
            }
        }
    }

    for(int z = 0; z < size; z ++) {
        for(int i = 0; i < (sqrtSize-1); i++) {
            for(int j = 0; j < (sqrtSize-1); j++) {
                for(int x = 0; x < sqrtSize; x++) {
                    for(int y = 0; y < sqrtSize; y++) {
                        for(int k = y+1; k < sqrtSize; k++) {
                            int s1 = -((size*size)*(sqrtSize*i+x)+size
                                *(sqrtSize*j+y)+z+1);
                            int s2 = -((size*size)*(sqrtSize*i+x)+size
                                *(sqrtSize*j+k)+z+1);
                            out.write(s1 + " " + s2 + "  0\n");
                        }
                    }
                }
            }
        }
    }

    for(int z = 0; z < size; z ++) {
        for(int i = 0; i < (sqrtSize-1); i++) {
            for(int j = 0; j < (sqrtSize-1); j++) {
                for(int x = 0; x < sqrtSize; x++) {
                    for(int y = 0; y < sqrtSize; y++) {
                        for(int k = x+1; k < sqrtSize; k++) {
                            for(int l = 0; l < sqrtSize; l++) {
                                int s1 = -((size*size)*(sqrtSize*i+x)+
                                    size*(sqrtSize*j+y)+z+1);
                                int s2 = -((size*size)*(sqrtSize*i+k)+
                                    size*(sqrtSize*j+l)+z+1);
                                out.write(s1 + " " + s2 + "  0\n");
                            }
                        }
                    }
                }
            }
        }
    }
```

```
        }

        out.close();
    }catch (Exception e){
        System.err.println("Error: " + e.getMessage());
    }
}

public int[][] readResult() {
    try {
        FileReader fstream = new FileReader("utfil.txt");
        BufferedReader in = new BufferedReader(fstream);
        in.readLine();
        //System.out.println(in.readLine());
        String line = in.readLine();
        String[] numbers = line.split(" ");

        int size = 16;
        int[][] matrix = new int[size][size];
        for(int i = 0; i < numbers.length-1; i++) {
            int a = Integer.parseInt(numbers[i]);
            if(a > 0) {
                int b = a-1;
                int x = b/(size*size);
                int c = b-(x*(size*size));
                int y = c/size;
                int z = c-(y*size);
                matrix[x][y] = (z+1);
            }
        }
        return matrix;

    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
}
```

B.3 Modifierad Brute-force

```
import java.util.ArrayList;
import java.util.HashMap;

public class ImprovedBruteForce {
public int size = 0;
/**
* Brute force solver with a twist
* @param matrix
* @return
```

```
*/
public boolean solve(int[][] matrix, HashMap<String, ArrayList<Integer>>
    matrixList) {
    size = matrix.length;
    if(matrixList == null) {
        matrixList = new HashMap<String, ArrayList<Integer>>();
        constructMatrixList(matrix, matrixList);
    }

    boolean running = true;
    while(running) {
        if(rule_one(matrix, matrixList)) {
            continue;
        } else if(rule_two(matrix, matrixList)) {
            continue;
        } else if(rule_three(matrix, matrixList)) {
            continue;
        } else {
            running = false;
        }
    }

    int x=0,y=0;
    boolean found = false;

    for(x = 0;x < size; x ++) {
        for(y = 0;y < size; y++) {
            if(matrix[x][y] == 0) {
                found = true;
                break;
            }
        }
        if( found ) break;
    }

    if(!found) {
        return true;
    }

    for(int i = 0; i < matrixList.get(x+"␣"+y).size(); i++) {
        int number = matrixList.get(x+"␣"+y).get(i);

        int[][] tmp = new int[size][size];
        HashMap<String, ArrayList<Integer>> tmpM = new HashMap<String,
            ArrayList<Integer>>();
        for(int q = 0; q < size; q++) {
            for(int w = 0; w < size; w++) {
                String pos = q+"␣"+w;

                if(matrixList.get(pos) != null) {
                    ArrayList<Integer> t = new ArrayList<Integer>();
                    for(int u = 0; u < matrixList.get(pos).size(); u++) {
                        t.add(matrixList.get(pos).get(u));
```

```
                    }
                    tmpM.put(pos, t);
                } else {
                    tmpM.put(pos, null);
                }
                tmp[q][w] = matrix[q][w];
            }
        }

        tmp[x][y] = number;
        tmpM.put(x+" "+y, null);
        remove(x,y,number, tmpM);

        if(solve(tmp, tmpM)) {
            for(int q = 0; q < size; q++) {
                for(int w = 0; w < size; w++) {
                    matrix[q][w] = tmp[q][w];
                }
            }
            return true;
        }
    }
    return false;
}

/**
* Construct hashmap with each positions list of possible values
* @param matrix
* @param matrixList
*/
public void constructMatrixList(int[][] matrix, HashMap<String, ArrayList<
    Integer>> matrixList) {
    for(int x = 0; x < size; x++) {
        for(int y = 0; y < size; y++) {
            if(matrix[x][y] == 0) {
                boolean digits[] = new boolean[size+1];

                for(int i = 0; i < size; i++) {
                    digits[matrix[x][i]] = true;
                    digits[matrix[i][y]] = true;
                }

                int s = (int)Math.sqrt(size);
                int bx = s * (x/s), by = s * (y/s);
                for(int i =0;i<s;i++)
                    for(int j = 0; j < s; j++)
                        digits[matrix[bx+i][by+j]] = true;

                ArrayList<Integer> availibleDigits = new ArrayList<Integer
                    >();
                for(int i = 1; i < (size+1); i++) {
                    if(digits[i] == false) {
                        availibleDigits.add(i);
```

```
                    }
                }
                matrixList.put(x+" "+y, availibleDigits);
            } else {
                matrixList.put(x+" "+y, null);
            }
        }
    }
}

/**
* Check for singles, if found add them to the matrix
* @param matrix
* @param matrixList
* @return
*/
public boolean rule_one(int[][] matrix, HashMap<String, ArrayList<Integer>>
     matrixList) {
    for(int x = 0; x < size; x++) {
        for(int y = 0; y < size; y++) {
            String pos = x+" "+y;
            if(matrixList.get(pos) != null && matrixList.get(pos).size() ==
                 1) {
                matrix[x][y] = matrixList.get(pos).get(0);
                matrixList.put(pos, null);

                remove(x, y, matrix[x][y], matrixList);

                return true;
            }
        }
    }
    return false;
}

/**
* Find "squeezed" numbers, if a region, row or column has a position with
    an unique
* value, remove all the other possible values for that same position
* @param matrix
* @param matrixList
* @return
*/
public boolean rule_two(int[][] matrix, HashMap<String, ArrayList<Integer>>
     matrixList) {
    for(int x = 0; x < size; x++) {
        HashMap<Integer, ArrayList<String>> row = new HashMap<Integer,
            ArrayList<String>>();
        HashMap<Integer, ArrayList<String>> col = new HashMap<Integer,
            ArrayList<String>>();
        for(int i = 1; i < (size+1); i++) {
            row.put(i, new ArrayList<String>());
            col.put(i, new ArrayList<String>());
```

```
        }

        for(int y = 0; y < size; y++) {
            if(matrixList.get(x+" "+y) != null) {
                ArrayList<Integer> rowList = matrixList.get(x+" "+y);
                for(int i = 0; i < rowList.size(); i++) {
                    row.get(rowList.get(i)).add(x+" "+y);
                }
            }
            if(matrixList.get(y+" "+x) != null) {
                ArrayList<Integer> colList = matrixList.get(y+" "+x);
                for(int i = 0; i < colList.size(); i++) {
                    col.get(colList.get(i)).add(y+" "+x);
                }
            }
        }

        for(int i = 1; i < (size+1); i++) {
            ArrayList<String> r = row.get(i);
            ArrayList<String> c = col.get(i);

            if(r.size() == 1) {
                matrixList.get(r.get(0)).clear();
                matrixList.get(r.get(0)).add(i);
                return true;
            }
            if(c.size() == 1) {
                matrixList.get(c.get(0)).clear();
                matrixList.get(c.get(0)).add(i);
                return true;
            }
        }
    }
    for(int x = 0; x < size; x = x+3) {
        for(int y = 0; y < size; y = y+3) {

            HashMap<Integer, ArrayList<String>> box = new HashMap<Integer,
                ArrayList<String>>();

            for(int i = 1; i < (size+1); i++) {
                box.put(i, new ArrayList<String>());
            }

            int s = (int)Math.sqrt(size);
            int bx = s * (x/s), by = s * (y/s);
            for(int i =0;i<s;i++) {
                for(int j = 0; j < s; j++) {
                    String pos = (bx+i)+" "+(by+j);
                    if(matrixList.get(pos) != null) {
                        ArrayList<Integer> boxList = matrixList.get(pos);
                        for(int p = 0; p < boxList.size(); p++) {
                            box.get(boxList.get(p)).add(pos);
                        }
```

```
                    }
                }
            }
            for(int i = 1; i < (size+1); i++) {
                ArrayList<String> b = box.get(i);

                if(b.size() == 1) {
                    matrixList.get(b.get(0)).clear();
                    matrixList.get(b.get(0)).add(i);
                    return true;
                }
            }
        }
    }
    return false;
}

/**
* Search for "bounded pairs" which is a set of tho possitions which
    contains the exact same two possibilites
* Remove these to possibilites from the rest of the positions in the same
    region, row or column
* @param matrix
* @param matrixList
* @return
*/
public boolean rule_three(int[][][] matrix, HashMap<String, ArrayList<Integer
    >> matrixList) {
    for(int x = 0; x < size; x++) {
        HashMap<Integer, ArrayList<String>> row = new HashMap<Integer,
            ArrayList<String>>();
        HashMap<Integer, ArrayList<String>> col = new HashMap<Integer,
            ArrayList<String>>();
        for(int i = 1; i < (size+1); i++) {
            row.put(i, new ArrayList<String>());
            col.put(i, new ArrayList<String>());
        }

        for(int y = 0; y < size; y++) {
            if(matrixList.get(x+" "+y) != null) {
                ArrayList<Integer> rowList = matrixList.get(x+" "+y);
                for(int i = 0; i < rowList.size(); i++) {
                    row.get(rowList.get(i)).add(x+" "+y);
                }
            }
            if(matrixList.get(y+" "+x) != null) {
                ArrayList<Integer> colList = matrixList.get(y+" "+x);
                for(int i = 0; i < colList.size(); i++) {
                    col.get(colList.get(i)).add(y+" "+x);
                }
            }
        }
```

```
for(int i = 1; i < (size+1); i++) {
    ArrayList<String> r1 = row.get(i);
    ArrayList<String> c1 = col.get(i);

    for(int j = i+1; j < (size+1); j++) {
        ArrayList<String> r2 = row.get(j);
        ArrayList<String> c2 = col.get(j);

        if(r1.size() == 2 && r2.size() == 2) {
            if(r1.contains(r2.get(0)) && r1.contains(r2.get(1))) {
                String[] pos1 = r1.get(0).split("␣");
                String[] pos2 = r1.get(1).split("␣");
                int r = Integer.parseInt(pos1[0]);

                for(int c = 0; c < size; c++) {
                    if(matrixList.get(r+"␣"+c) != null && c !=
                        Integer.parseInt(pos1[1]) && c != Integer.
                        parseInt(pos2[1])) {
                        if(matrixList.get(r+"␣"+c).contains((
                            Integer)i)) {
                            matrixList.get(r+"␣"+c).remove((Integer
                                )i);
                        } else if(matrixList.get(r+"␣"+c).contains
                            ((Integer)j)) {
                            matrixList.get(r+"␣"+c).remove((Integer
                                )j);
                        }
                    }
                }
            }
        }
        if(c1.size() == 2 && c2.size() == 2) {
            if(c1.contains(c2.get(0)) && c1.contains(c2.get(1))) {
                String[] pos1 = c1.get(0).split("␣");
                String[] pos2 = c1.get(1).split("␣");
                int c = Integer.parseInt(pos1[1]);

                for(int r = 0; r < size; r++) {
                    if(matrixList.get(r+"␣"+c) != null && r !=
                        Integer.parseInt(pos1[0]) && r != Integer.
                        parseInt(pos2[0])) {
                        if(matrixList.get(r+"␣"+c).contains((
                            Integer)i)) {
                            matrixList.get(r+"␣"+c).remove((Integer
                                )i);
                        } else if(matrixList.get(r+"␣"+c).contains
                            ((Integer)j)) {
                            matrixList.get(r+"␣"+c).remove((Integer
                                )j);
                        }
                    }
                }
            }
```

```
                }
            }
        }
    }
    return false;
}

/**
* Remove a number from the row, column and region of a position
* @param x
* @param y
* @param number
* @param matrixList
*/
public void remove(int x, int y, int number, HashMap<String, ArrayList<
    Integer>> matrixList) {
    for(int i = 0; i < size ; i++) {
        if(matrixList.get(x+" "+i) != null && matrixList.get(x+" "+i).
            contains((Integer)number)) {
            matrixList.get(x+" "+i).remove((Integer)number);
        }
        if(matrixList.get(i+" "+y) != null && matrixList.get(i+" "+y).
            contains((Integer)number)) {
            matrixList.get(i+" "+y).remove((Integer)number);
        }
    }

    int s = (int)Math.sqrt(size);
    int bx = s * (x/s), by = s * (y/s);
    for(int i =0;i<s;i++) {
        for(int j = 0; j < s; j++) {
            if(matrixList.get((bx+i)+" "+(by+j)) != null && matrixList.get
                ((bx+i)+" "+(by+j)).contains((Integer)number)) {
                matrixList.get((bx+i)+" "+(by+j)).remove((Integer)number);
            }
        }
    }
}
}
```