KTH Computer Science
and Communication

Lösning av Sudoku med mänskliga strategier

DD143X, Examensarbete i datalogi om 15 högskolepoäng vid
Programmet för datateknik 300 högskolepoäng

JOHAN BRODIN, JBRODI@KTH.SE
JONATHAN PELLBY, JONATHP@KTH.SE

Examenrapport vid CSC
Handledare: Pawel Herman
Examinator: Mårten Björkman

...

Abstract

Sudoku is a puzzle that has become popular during the last decade. A great number of algorithms have been implemented to solve the problem but many use approaches that do not correspond to how a human solves a sudoku puzzle. The purpose of this project was to evaluate how a strategy based algorithm which utilizes a more human approach compares to a so called Dancing Links-algorithm and an exhaustive search algorithm with respect to solving time and correctness.

A qualitative study was conducted, with four respondents, which together with a literature study lay the foundation for the strategies in the implemented algorithm.

Furthermore, a comparison was performed with 725 sudoku puzzles of which the strategy-based algorithm only managed to solve sudoku puzzles of a limited difficulty where the others solved all. The result showed though, that the strategy based algorithm was the fastest for the difficulties that it solved, but at the same time it showed a trend hinting at increased solving time with increasing difficulty.

Finally, it could be determined that Dancing Links is a generally faster algorithm than the others. A strategy based algorithm could however be a fast alternative when solving sudoku puzzles of the difficulties that often appear in daily papers.

Referat

Sudoku är ett pussel som har blivit populärt det senaste decenniet. Ett stort antal algoritmer har implementerats för att lösa problemet men många använder angreppssätt som inte motsvarar hur en människa går tillväga för att lösa sudokupussel. Syftet med projektet var utvärdera hur en strategibaserad algoritm som använder ett mer mänskligt angreppsätt står sig mot en sk. Dancing Links-algoritm och en totalsökningsalgoritm med avseende på tid och korrekthet.

I projektet genomfördes en kvalitativ undersökning, med fyra respondenter, som tillsammans med en litteraturstudie lade grunden för strategierna i den implementerade algoritmen.

Vidare gjordes en jämförelse med 725 sudokupussel av vilka den strategibaserade algoritmen endast löste sudokupussel av en begränsad svårighetsgrad medan de övriga löste samtliga. Resultat visade däremot att den strategibaserade algoritmen var snabbast för de svårighetgrader den löste men samtidigt också en trend om förlängd lösningstid vid ökande svårighetgrad.

Slutligen kunde det konstateras att Dancing Links är en generellt snabbare algoritm än de övriga. En strategibaserad algoritm kan dock vara ett snabbt alternativ vid lösning av sudokupussel av de svårighetsgrader som ofta förekommer i dagstidningar.

Samarbete inom gruppen

Detta projekt har genomförts av två personer. Initialt gjorde Jonathan Pellby merparten av kapitel 1 och 2, introduktion och bakgrund, medan Johan Brodin skrev stora delar av kapitel 4, resultat. Kapitel 3, metod, påbörjades av Johan Brodin och avslutades av Jonathan Pellby. Kapitel 5 och 6, diskussion och slutsats, är skrivna i nära samarbete. Samtliga kapitel i rapporten har därefter reviderats av båda projektdeltagarna.

Innehåll

Innehåll

1 Introduktion **1**
- 1.1 Problemförklaring . . . 2
- 1.2 Syfte . . . 2

2 Bakgrund **3**
- 2.1 Grunderna i Sudoku . . . 3
- 2.2 Logiska strategier för lösning av sudokupussel . . . 4
  - 2.2.1 Nakna tupler . . . 4
  - 2.2.2 Dolda tupler . . . 5
- 2.3 Befintliga algoritmer för lösning av sudokupussel . . . 7
  - 2.3.1 Totalsökning . . . 7
  - 2.3.2 Dancing Links . . . 8

3 Metod **9**
- 3.1 Svårighet av sudokupussel . . . 9
- 3.2 Undersökning av strategier . . . 10
  - 3.2.1 Undersökningens syfte . . . 10
  - 3.2.2 Beskrivning av undersökningsförfarande . . . 10
  - 3.2.3 Undersökningsmaterialets utformning . . . 10
  - 3.2.4 Undersökningens omfattning . . . 11
- 3.3 Implementation av algoritmer . . . 11
  - 3.3.1 Totalsökning . . . 11
  - 3.3.2 Dancing Links . . . 11
  - 3.3.3 Strategibaserad algoritm . . . 12
- 3.4 Kriterium för att bedöma algoritm . . . 13
  - 3.4.1 Hastighet . . . 13
  - 3.4.2 Korrekthet . . . 13
  - 3.4.3 Komplexitet . . . 13
- 3.5 Jämförelse av algoritmer . . . 14
  - 3.5.1 Mätningen . . . 14
  - 3.5.2 Jämförelser . . . 14

3.5.3 Val av sudokupussel för tester 15

4 Resultat **17**
4.1 Undersökningens resultat 17
4.2 Jämförelser 19
4.2.1 Korrekthet 19
4.2.2 Lösningstid 20

5 Diskussion **23**
5.1 Diskussion av resultat 23
5.1.1 Korrekthet 23
5.1.2 Lösningstid 24
5.1.3 Korrekthet kontra lösningstid 25
5.2 Undersökningen 25
5.3 Svårighetsgrad 26
5.3.1 Tid det tar att lösa pusslet 26
5.3.2 Antal ledtrådar 26
5.3.3 Sudoku Explainers definition 26
5.4 Vid återupprepning av projektet 27

6 Slutsatser **29**

Litteraturförteckning **31**

Bilagor **31**

A Sudokupussel som använts i undersökningen **33**

B Källkod för Strategibaserad algoritm **35**

Kapitel 1

Introduktion

Sudoku är ett pussel i vilket målet är att placera siffror i ett rutnät under några givna restriktioner. En grundpelare är att den som löser pusslet utnyttjar logiska strategier för att lista ut var siffrorna ska placeras. Pusslet har under det senaste decenniet blivit väldigt populärt och det finns mycket litteratur som förklarar vilka strategier som kan användas för att med logik lösa det.

Problemet har också funnit intresse bland programmerare och en stor mängd algoritmer har implementerats de senaste åren. Dessa algoritmer baseras på mer datoranpassade metoder att lösa problemet. Totalsökning är en vanlig metod men det förekommer också varianter som abstraherar pusslet till det mer generella datalogiska problemet exakt mängdtäckning. En sådan algoritm är Donald Knuths så kallade Dancing Links algoritm[1]. Inget av dessa angreppssätt motsvarar dock hur en människa går till väga för att lösa ett sudokupussel.

I detta projekt implementeras därför en algoritm som i större utsträckning liknar det angreppsätt som människor använder sig av vid lösning av sudokupussel, med logiska strategier som grund. Detta för att undersöka hur en sådan algoritm kan mäta sig med de mer datoranpassade algoritmerna vad gäller exekveringstid och andel lösta sudokupussel. Konstruktionen av algoritmen baseras dels på en litteraturstudie men också på en undersökning där personer observeras medan de löser sudokupussel.

1.1 Problemförklaring

Nedan följer en detaljerad sammanställning av de delmoment som ingår i projektet.

- Undersöka några strategier som människor använder för att lösa sudokupussel genom litteraturstudie och intervjubaserad undersökning.
- Konstruera och implementera en algoritm baserat på ovanstående punkt.
- Implementera en totalsökningsalgoritm som löser sudokuproblemet.
- Implementera en variant av Donald Knuths Dancing Links algoritm som löser sudokuproblemet.
- Jämföra den, i projektet, konstruerade algoritmen med de två övriga algoritmernas effektivitet vid lösning av sudokupussel med avseende på exekveringstid och korrekthet.

Inom ramen för detta projekt åsyftas endast den klassiska varianten av sudokupussel med kravet att de har en unik lösning. Vidare kommer projektet inte behandla hur konstruktionen av de sudokupussel som används vid jämförelse av algoritmerna går till.

1.2 Syfte

Syftet med detta projekt är att undersöka logiska strategier som människor använder sig av för att lösa sudokupussel och implementera en algoritm baserat på detta. Detta uppnås genom att jämföra en sådan algoritm med andra algoritmer som utnyttjar angreppssätt som är mer anpassade för en dator. Utifrån jämförelsen är förväntan att få en större inblick i hur de logiska strategierna kan omsättas i kod och hur en sådan algoritm presterar med avseende på tid och korrekthet.

Kapitel 2

Bakgrund

I detta kapitel ges en överblick av grunderna i sudoku och en genomgång av några vanliga strategier för logisk lösning av Sudoku. Avslutningsvis ges en kort beskrivning av två befintliga algoritmer för att lösa sudokupussel med datorer.

2.1 Grunderna i Sudoku

Ett generellt sudokupussel består av rutnät med $n^2 \times n^2$ celler där n är ett heltal större än 1. Förutom att vara indelad i rader och kolumner, innehåller rutnätet också n^2 stycken delrutnät, så kallade lådor, av storlek $n \times n$. En zon är ett samlingsnamn och innebär en rad, kolumn eller låda.

Målet med pusslet är att placera ut siffrorna $1-n^2$ utan att bryta de restriktioner som gäller för Sudoku, nämligen att varje cell måste innehålla exakt en siffra och att varje zon måste innehålla varje siffra exakt en gång[2]. De siffror som kan placeras i en viss cell, givet restriktionerna, kallas för cellens kandidater.

I den klassiska varianten av sudokupussel är talet n lika med 3. Spelbrädet är då av storlek 9×9 rutor och är indelad i nio stycken lådor. Dessa är av storlek 3×3 rutor och siffrorna 1-9 ska placeras ut, se Figur 2.1. I figuren är även kandidaterna till varje olöst cell markerade.

Varje sudokupussel börjar med ett antal placerade siffror, hädanefter benämnda som ledtrådar, och det är dessa ledtrådar som skiljer två sudokupussel från varandra. Ett sudokupussel som har en unik lösning kallas för ett giltigt sudokupussel.

Det har bevisats genom totalsökning att det minsta antalet givna siffror som krävs för ett giltigt sudokupussel är 17 siffror[3]. För att ett sudokupussel ska vara giltigt måste dessutom minst åtta av de nio siffrorna vara representerade bland ledtrådarna. Detta är uppenbart eftersom alla förekomster av de två siffrorna som saknas då skulle kunna byta plats. Sudokupusslet får då två olika lösningar.

En grundpelare i sudokupussel är att de ska gå att lösa med hjälp av logiska resonemang utifrån de givna ledtrådarna. Den som löser ett pussel ska inte behöva gissa sig till en siffras placering [4].

Figur 2.1. En bild av ett klassiskt sudokupussel. De mindre siffrorna är kandidaterna till den cell de står i.

2.2 Logiska strategier för lösning av sudokupussel

Det finns flertalet logiska strategier som människor använder sig av vid lösning av sudokupussel. I denna sektion beskrivs några vanliga strategier som används. Gemensamt för de strategier som beskrivs är att de minskar antalet kandidater för en cell i syfte att finna vilken siffra som måste placeras i cell.

2.2.1 Nakna tupler

Nakna tupler hittas genom att studera kandidater för ett antal celler och dra slutsatser utifrån dem för att kunna eliminera kandidater från rutornas gemensamma zon.

Naken singel

En naken singel är en av de simplaste strategierna för att lösa sudokupussel. Den uppstår när en cell endast har en kandidat, vilket medför att denna måste placeras där[2]. I Figur 2.2 är siffran 5 den enda kandidaten till den grönmarkerade cellen och är således en naken singel.

Naket par

Ett naket par uppstår när två stycken celler som ligger i samma zon har exakt två kandidater och där kandidaterna är samma för båda cellerna[2]. Då måste de två kandidaterna vara i någon av de två cellerna och siffrorna kan således elimineras som kandidater för resterande celler i zonen.

Figur 2.2. Den grönmarkerade cellen innehåller en naken singel eftersom 5 är dess enda kandidat.

Naken trippel, kvadrupel och kvintupel

En naken trippel liknar ett naket par men involverar tre celler och tre siffror. Varje siffra behöver dock inte vara kandidat för alla tre celler. Kvadrupler och kvintupler fungerar på samma sätt fast med fyra respektive fem celler och siffror[2]. I sudokupusslet i Figur 2.3 finns det en naken trippel i kolumn två i de tre grönmarkerade cellerna. I cellerna kan endast 1, 2 och 5 placeras och eftersom det är tre celler och tre siffror kan siffrorna inte placeras på något annat ställe i kolumnen. De kan således elimineras från kandidaterna till övriga celler i kolumn två.

2.2.2 Dolda tupler

Dolda tupler liknar nakna tupler i det att de försöker eliminera kandidater. Till skillnad från nakna tupler behöver dock inte siffrorna som hör till en dold tupel vara de enda kandidaterna till cellerna.

Dold singel

Att använda dolda singlar vid lösning av sudokupussel kan liknas vid enkel uteslutning. En dold singel uppstår när en given siffra endast är kandidat till en cell i en zon[2]. Siffran måste således placeras i cellen. Detta kan jämföras med en naken singel, som uppstår när en cell endast har en kandidat. I Figur 2.4 finns det en dold singel i den grönmarkerade cellen. Det är den enda cellen i den lådan som har siffran 8 som kandidat vilket medför att 8 måste placeras där.

Figur 2.3. En naken trippel i kolumn två. Den består av de tre grönmarkerade cellerna och siffrorna 1, 2 och 5.

Figur 2.4. En dold singel i den grönmarkerade cellen. Det är den enda cellen i lådan som siffran 8 kan placeras i.

Dolt par, dold trippel och dold kvadrupel

Ett dolt par uppstår när två siffror endast är kandidaterna till två celler i en zon. Det följer att siffrorna måste placeras i de cellerna och cellernas övriga kandidater kan därmed strykas. Dolda tripplar och kvadruplar fungerar på samma sätt, men då med tre respektive fyra celler och siffror. På samma sätt som med nakna tupler av högre ordning behöver inte alla siffror återfinnas i varje cell[2]. Ett dolt par kan

observeras i den sjunde raden av sudokupusslet i Figur 2.5. De enda cellerna i raden som siffrorna 1 och 9 kan placeras i är de grönmarkerade. Det följer att siffrorna måste vara där och att resterande siffror kan elimineras från de grönmarkerade cellernas kandidater.

Figur 2.5. Ett dolt par i rad sju. Siffrorna 1 och 9 kan endast placeras i de två grönmarkerade cellerna i raden.

2.3 Befintliga algoritmer för lösning av sudokupussel

I följande sektion behandlas ett antal befintliga algoritmer för lösning av sudokupussel. Algoritmerna bygger på logik som kräver kapacitet att lagra mycket information i minnet samtidigt och möjlighet att snabbt återställa ett sudokupussel till ett tidigare tillstånd. Således är algoritmerna bättre anpassade efter en dators prestanda än en människas förmåga.

2.3.1 Totalsökning

En totalsökningsalgoritm utnyttjar ett relativt simpelt angreppssätt för att lösa ett problem. Den testar alla möjliga varianter av det givna problemet tills dess att den finner en lösning. För klassiska sudokupussel kan en totalsökningsalgoritm lösa alla indata på relativt kort tid (mindre än en sekund), på grund av den begränsade storleken. Eftersom totalsökningsalgoritmer testar tills den hittar rätt lösning kan den lösa alla giltiga sudokupussel.

2.3.2 Dancing Links

Sudoku är en variant av problemet exakt mängdtäckning och att lösa sudokupussel av generell storlek har bevisats vara NP-fullständigt[5, 6]. Detta innebär i praktiken att det saknas en rimligt tidseffektiv algoritm för att lösa problemet[7]. Problemet exakt mängdtäckning kan ses som att ur en matris fylld med ettor och nollor välja rader i matrisen så att kolumnsumman av de valda raderna blir 1. En mindre matris (2.1) illustrerar detta där raderna 1, 4 och 5 ger en lösning.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \tag{2.1}$$

Dancing Links är en rekursiv, icke-deterministisk, djupet-först, backtracking algoritm utvecklad av Donald Knuth som används för att lösa sådana problem[1]. Om matrisen konstrueras för att representera ett sudokupussel kan Dancing Links genom att lösa problemet exakt mängdtäckning också lösa sudokupusslet.

Kapitel 3

Metod

I detta kapitel beskrivs hur projektet har genomförts. Det inleds med att behandla hur svårigheten av ett sudokupussel ska bedömas. Därefter beskrivs hur undersökningen av strategier genomförts. Sedan ges en beskrivning av de implementerade algoritmerna, med tyngdpunkt på den algoritm som utifrån undersökningen har konstruerats. Slutligen beskrivs hur jämförelsen av algoritmerna har genomförts med tillhörande definition av de kriterier som den gjorts utifrån.

3.1 Svårighet av sudokupussel

I projektet jämförs de implementerade algoritmernas prestanda med olika sudokupussel som indata. I teorin kan en algoritm prestera olika väl beroende på strukturen av det sudokupussel den ska lösa. För att kunna göra rättvis jämförelser är det därför viktigt att definiera riktlinjer för hur svårighetsgraden av ett sudokupussel ska bedömas. Det tycks idag däremot inte finnas någon standard för att gradera svårigheten av ett sudokupussel[4].

Som tidigare beskrivet finns det ett antal logiska strategier som används för att lösa Sudoku. Ett möjligt sätt att bedöma svårigheten av ett sudokupussel är att utgå från vilka strategier som krävs för att lösa sudokupusslet[4]. Strategierna måste då viktas efter hur avancerade de är. Ett sudokupussel som kräver mer avancerade strategier blir således svårare. Då ett sudokupussel ofta kan lösas på mer än ett sätt och bör graderingen baseras på det sätt som kräver minst avancerade strategier[4]. Att gradera svårighetsgraden utifrån detta är rimligt, då den blir direkt baserad på den logiska förmågan hos den som löser sudokupusslet. Att avväga hur användbara och avancerade strategier är kräver dock omfattande analys som inte ryms inom ramen för detta projekt. Därför används ett verktyg som heter Sudoku Explainer (SE). SE graderar sudokupussels svårighetsgrad baserat på ovannämnda resonemang på en skala från 1.0 till 10.0, där högre siffra innebär högre svårighetsgrad. Sudokupussel mellan 1.0–1.2 bedöms som lätta, 1.5 anses vara medelsvåra och 1.7 till 2.5 bedöms som svåra[8].

3.2 Undersökning av strategier

Projektet inleddes med en litteraturstudie av vilka strategier som används av människor för att lösa sudokupussel. I samband med detta utformades en undersökning för att få mer konkret information om hur människor går tillväga i lösningen av ett sudokupussel. I denna sektion beskrivs hur undersökningen utformades och genomfördes.

3.2.1 Undersökningens syfte

Syftet med undersökningen var att samla information om vilka strategier som människor använder sig av för lösa sudokupussel. Fokus låg på att finna användbara strategier, både bland de funna i litteraturstudien och bland övriga, som skulle kunna inkluderas i en algoritm för en dator. Med det i åtanke utformades därför undersökningen för att ge kvalitativa resultat snarare än kvantitativa.

3.2.2 Beskrivning av undersökningsförfarande

Som undersökningsförfarande användes ett intervjubaserat upplägg. En genomförd intervju bestod av en intervjuare, en respondent samt undersökningsmaterial som innefattade ett tidtagarur, sudokupussel på A4-papper, en mall för att översätta cellerna i sudokupusslen till ett koordinatsystem och ett undersökningsprotokoll.

Intervjun började med att intervjuaren förklarade syftet med undersökningen för att sedan instruera respondenten om att denna skulle lösa ett, eller flera, sudokupussel och att medverkan var anonym. Respondenten ombads ständigt beskriva sin tankegång och vara beredd på att svara på frågor.

När respondenten placerade en siffra i sudokupusslet noterades vilken siffra det var, i vilken cell den placerades, varför respondenten placerade den där samt tiden för placeringen. När sudokupusslet löstes uppmanades respondenten att gradera svårighetsgraden för sudokupusslet och fick chans att ge övriga kommentarer om själva pusslet och summera de strategier som denne använder. Eventuella frågetecken som uppstått under själva lösandet reddes ut av intervjuaren. Intervjun avslutades med att respondenten fick frågan om denna ville ha en kopia av rapporten.

3.2.3 Undersökningsmaterialets utformning

Två sudokupussel användes vid intervjuerna och var samma för varje intervju. Vid valet av sudokupussel lades vikt vid att finna sudokupussel av rimlig svårighetsgrad. Ett sudokupussel med allt för låg svårighetsgrad skulle kunna lösas med relativt simpla metoder och endast ge triviala resultat beträffande strategier. Om pusslet å andra sidan var för svårt skulle det ha kunnat resultera i att respondenten fastnade och inte kunde komma vidare. Med detta i åtanke genererades med hjälp av SE två sudokupussel med svårighetsgrad medelsvår och svår[8]. Se Bilaga A.

3.2.4 Undersökningens omfattning

Totalt genomfördes fyra intervjuer och totalt löstes de två sudokpusslen tre gånger vardera. I vissa fall hann endast respondent med att lösa ett sudokupussel. Omständigheterna och lokalerna varierade. En av respondenterna blev intervjuad på KTH och de andra i sina hem. Respondenterna blev personligt tillfrågade av intervjuaren och urvalet var personer som ansåg sig ha god erfarenhet av att lösa sudokupussel av högre svårighetsgrad.

3.3 Implementation av algoritmer

I projektet har tre algoritmer implementerats. Samtliga algoritmer är skrivna i Java, som valts på grund av god erfarenhet och tillgänglighet till befintlig kod för Dancing Links-algoritmen. Nedan följer beskrivningar av implementation av dessa.

3.3.1 Totalsökning

Den totalsökningsalgoritm som har implementerats baseras på pseudokod, Se Kod 3.1, från en föreläsning vid Stanford University[9]. Det är en rekursiv, backtrackande algoritm som i varje steg går igenom pusslet, finner en tom cell och försöker placera en siffra i cellen så att inga konflikter uppstår. När den inte längre kan placera någon siffra i en cell så backtrackar den till ett tidigare tillstånd och försöker fortsätta därifrån.

Kod 3.1. Pseudokod för en totalsökningsalgoritm för lösning av sudokupussel av storlek 9 x 9 från Stanford University

```
bool SolveSudoku(Grid<int> &grid){
    int row, col;
    if (!FindUnassignedLocation(grid, row, col))
        return true; // all locations successfully assigned!
    for (int num = 1; num <= 9; num++) { // options are 1-9
        if (NoConflicts(grid, row, col, num)) { // if # looks ok
            grid(row, col) = num; // try assign #
            if (SolveSudoku(grid))
                return true; // recur if succeed stop
            grid(row, col) = UNASSIGNED; // undo & try again
        }
    }
    return false; // this triggers backtracking from early decisions
}
```

3.3.2 Dancing Links

En matris för problemet exakt mängdtäckning kan konstrueras för att representera ett sudokupussel. Matrisen får 729 rader ($9 \times 9 \times 9$) där varje rad motsvarar placeringen av en viss siffra i en viss cell. De 324 kolumnerna motsvarar de restriktioner

som fås ur reglerna som definierar Sudoku[5]. För varje rad placeras en etta i de kolumner vars restriktioner radens sifferplacering uppfyller.

Implementationen av Dancing Links-algoritmen baseras på kod skriven av Daniel Seiler[10]. Den använder sig av dubbellänkade listor för att representera matrisen. En nod håller referenser till de noder som ligger ovanför, nedanför samt till vänster och höger om noden[1]. Algoritmen väljer i varje steg ut en kolumn och en nod i kolumnen som får uppfylla kolumnens restriktion. Detta motsvarar att välja en rad i matrisen. Resterande noder i kolumnen, och deras tillhörande rader tas då bort eftersom de inte kan tillhöra samma lösning. Om algoritmen kommer till en punkt då den inte kan fortsätta återställer den rader och kolumner till ett tidigare tillstånd. Grunden för att göra detta effektivt är de dubbellänkade listorna, i vilka insättning och borttagning kan göras enkelt[1].

Valet att implementera Dancing Links baseras dels på att algoritmen utnyttjar en abstraktion av sudokupussel till ett mer generellt problem, men också att det är en dokumenterat tidseffektiv algoritm för att lösa det generella problemet exakt mängdtäckning[5, 1]. Detta öppnar upp för intressanta resultat vid jämförelse med de andra implementerade algoritmerna.

3.3.3 Strategibaserad algoritm

Utifrån resultatet från litteraturstudien och undersökningen har en algoritm konstruerats som baseras på de logiska strategier som människor kan använda sig av för att lösa sudokupussel. Strategierna som implementerades har sin grund i vilka strategier respondenterna i undersökningen använde. Implementationen inkluderar även strategierna naken trippel, kvadrupel och dolt par. Dessa inkluderades på grund av deras likhet med de strategier som respondenterna använde. En grundpelare i implementationen är användningen av cellers kandidater. Dessa genereras direkt när algoritmen startar och utnyttjas i flertalet av strategierna.

Algoritmens angreppssätt baseras i stort sett på det angreppsätt som observerades hos respondenterna i undersökningen. Den utgår hela tiden från att försöka hitta nakna singlar och när det misslyckas försöker den använda sig av strategierna i tur och ordning. När en strategi lyckas återgår den till att leta nakna singlar igen. Om ingen strategi lyckas terminerar algoritmen och resultatet blir ett ofullständigt sudokupussel. Algoritmen placerar aldrig en siffra utan att vara säker på dess placering.

Nedan följer de strategier som implementerats i den prioritetsordning utifrån vilken de körs i algoritmen.

1. Naken singel
2. Dold singel
3. Pekande par
4. Naket par

5. Dolt par
6. Naken trippel
7. Naken kvadrupel

Strategierna är ordnade efter hur avancerade de anses vara och implementerades efter beskrivningen i sektion 2.2 och 4.1. Se Bilaga B för fullständig kod.

3.4 Kriterium för att bedöma algoritm

Inom projektet har tre stycken algoritmer implementerats och jämförts. För att kunna göra jämförelser är det dock viktigt att definiera utifrån vilka kriterier de ska utföras. I följande sektion beskrivs de kriterium som har använts i projektet.

3.4.1 Hastighet

Ett vanligt sätt att bedöma en algoritm är utifrån dess hastighet. Det kan göras genom att undersöka hur lång tid den tar för att lösa ett givet problem. En tidsmätning görs rimligtvis mellan det att algoritmen startar tills dess att den terminerar. Exekveringstiden kan dock skilja sig åt mellan olika exekveringar, trots att indata är densamma[11]. Det är därför viktigt att konstruera tidsmätningen på ett sådant sätt att den ger en så korrekt bild av exekveringstiden som möjligt. Se sektion 3.5.1 för hur detta hanteras i projektet.

3.4.2 Korrekthet

En totalsökningsalgoritm som används för att lösa sudokupussel hittar alltid en korrekt lösning, förutsatt att det givna pusslet och algoritmen är korrekt. Att andra algoritmer som inte baseras på totalsökning ger en korrekt lösning är dock inte givet. Korrekthet blir därmed ett intressant kriterium att bedöma en algoritm utifrån. En möjlighet till definition av korrekthet är antalet felplacerade siffror i den lösning som algoritmen ger. Ett svar till ett sudokupussel där någon av reglerna i Sudoku bryts är dock ointressant. Därför definieras kriteriet för en algoritms korrekthet istället utifrån andelen sudokupussel som algoritmen ger korrekta lösningar till. För att kunna göra jämförelser och dra välgrundade slutsatser från detta kriterium är det viktigt att algoritmerna körs med en stor mängd indata.

3.4.3 Komplexitet

En analys av tidskomplexiteten för en algoritm visar hur tiden för exekvering ökar när storleken på indata ökar. Eftersom storleken för indata är konstant i detta projekt är en komplexitetsanalys dock av mindre intresse. Istället kan det vara intressant att undersöka hur exekveringstiden och andelen korrekta lösningar förändras när svårighetsgraden på sudokupusslen ökar.

3.5 Jämförelse av algoritmer

För att utvärdera den algoritm som konstruerats i projektet jämförs den med Dancing Links och Totalsökning. De tre lösningsalgoritmer har testas på en mängd sudokupussel utifrån exekveringstid och andel korrekta sudokupussel. Jämförelserna görs utifrån hur dessa förändras när svårighetsgraden av sudokupusslen ökar.

Ett test består utav en lösningsalgoritm och ett sudokupussel. Varje lösningsalgoritm implementerar ett gränssnitt med en funktion - *solve*. Funktionen anropas med en två-dimensionell vektor som representerar sudokupusslet den ska lösa och returnerar en annan två-dimensionell vektor som innehåller det lösta pusslet. Algoritmernas interna struktur skiljer sig och detta upplägg har därför valts för att tiden det tar att översätta ett sudokupussel från ett standardiserat format ska tillräknas tidmätningen.

3.5.1 Mätningen

Innan testet börjar körs algoritmen med sudokupusslet ett antal gånger utan att spara data om resultatet. Det beror på att de första gångerna koden körs i Java har kompilatorn inte effektiviserat den.

Mätningen av tid börjar från det att funktionen *solve* anropas, och avslutas när denna returnerar lösningen. Efter det noteras tiden och huruvida lösningen är korrekt. Korrektheten bekräftas genom att kontrollera att inga restriktioner för Sudoku bryts. Inga externa program används för tidsmätning, endast Javas inbyggda metoder. Mätningen upprepas ett antal gånger för att samla tillräckligt mycket data för att utesluta enstaka extremvärden. Dessa kan uppkomma av flera anledningar, t ex. kan datorns processor bli avbruten av en annan process.

När testet är klart skrivs resultatet, dvs. huruvida algoritmen löste pusslet, pusslets namn och svårighetsgrad samt en lista med tidsåtgång, till en Excel-fil. I denna fil kan sedan statistiska operationer utföras och grafer genereras.

3.5.2 Jämförelser

För att få en rättvis bedömning av mätdata för ett test har median valts över medelvärde då mätdatat är snedfördelat, dvs. det förekommer förhöjda värden som inte bör få påverka det samlade resultatet. Lösningstiden för sudokupussel av en viss svårighetsgrad beräknas som medelvärdet av de uppmätta medianvärdena för den svårighetsgraden. Det används som ett jämförbart värde för hur de tre algoritmerna presterar.

Tider jämförs endast på sudokupussel som samtliga algoritmer klarar av att lösa. Detta för att undvika behovet att definiera hur lång tid ett misslyckat försök tar och för att få en rättvis bild av algoritmernas tider. En algoritm som klarar av att lösa ett mindre antal sudokupussel ska inte straffas i tidsavseende. Det tas istället upp vid jämförelse av korrekthet och blir relevant vid en total jämförelse av algoritmerna över samtliga kriterium.

3.5.3 Val av sudokupussel för tester

För att generera testdata har programmet Sudoku Explainer (SE) använts. SE är ett open-source projekt som både kan generera och gradera sudokupussel. Genom modifiering av koden till SE kunde programmets inbyggda generator användas för att skriva sudokupussel inom ett visst svårighetsgradsintervall till fil.

Generatorn slumpar fram ett sudokupussel, kontrollerar med hjälp av en inbyggd lösare om det är korrekt och beräknar svårighetsgraden av den. Om svårighetsgraden ligger i det fördefinierade intervallet så returneras den. Sudokupusslen lagras på fil med punkter som markerar de tomma cellerna. I Figur 3.1(a) illustreras hur sudokupusslet i Figur 3.1(b) lagras på fil.

```
53..7....
6..195...
.98....6.
8...6...3
4..8.3..1
7...2...6
.6....28.
...419..5
....8..79
```

(a) Sudokupusslet lagrat på fil.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(b) Sudokupusslet.

Figur 3.1. Illustrerar hur (b) lagras som fil i (a)

Kapitel 4

Resultat

I detta kapitel beskrivs de resultat som erhållits under projektet. Den första sektionen innehåller resultaten från den undersökning som genomförts. Den andra beskriver resultatet från jämförelsen av de tre implementerade algoritmerna.

4.1 Undersökningens resultat

I denna sektion sammanfattas resultatet från undersökningen. Den inleds med en genomgång av hur respondenterna upplevde svårighetsgraden av de sudokupussel de löste och fortsätter sedan med en sammanfattning av olika tillvägagångssätt som de använde för att lösa sudokupusslet.

Svårighetsgrad

Det medelsvåra sudokupusslet löstes på 15, 23 respektive 31 minuter. Samtliga respondenter bedömde svårighetsgraden av pusslet som medelsvårt. Bland annat motiverades detta med att det hela tiden fanns tydliga strategier att använda.

Det svåra sudokupusslet löstes på 24, 26 respektive 34 minuter. Två av tre av respondenterna ansåg att pusslet var av svårighetsgrad svår medan den sista tyckte att det låg på en svårighetsgrad mellan medel och svår.

Tillvägagångssätt

Samtliga respondenter arbetade utifrån fullständig korrekthet, dvs. de satte bara ut siffror när de var övertygade om att deras placering var korrekt. Det hände vid ungefär tre tillfällen att en respondent placerade en siffra fel. Detta upptäcktes dock tämligen fort och genom att följa sin egen arbetsgång baklänges kunde respondenten korrigera felet.

Kandidater

Tre av fyra respondenter antecknade ibland kandidater till en cell genom att göra anteckningar i just den cellen i sudokupusslet. Den fjärde använde sig uppenbart av kandidater, men höll dem istället i minnet. En av respondenterna markerade sporadiskt ut kandidater i pusslet, medan andra resonerade att det skulle undvikas i möjligaste mån då det tog onödig tid och förstörde överblicken.

Fokus

Samtliga respondenter använde snarlika strategier för välja var de skulle fokusera sitt sökande. Det vanligaste var att de fokuserade på zoner där det fanns många ledtrådar med motiveringen att det där existerade färre alternativ. Två av respondenterna prioriterade aktivt de zoner eller siffror som hade 2-3 siffror kvar att placera. En respondent markerade dessa i pusslet för att underlätta upptäckandet av en naken singel vid förändringar i pusslet. En annan respondent räknade ledtrådar av samma siffra, om tillräckligt många fanns så uppsöktes de lådor där siffran saknades. Om ovanstående inte upptäcktes fokuserade somliga av respondenterna på att hitta celler vars zoner hade många placerade siffror.

När en siffra väl placerats fanns olika idéer om fortskridande. Vissa fokuserade på närliggande zoner för att se effekten av den nyss placerade siffran. Några av respondenterna följde detta väldigt metodiskt och resultatet blev att pusslet löstes zon för zon dvs. vissa delar av sudokupusslet kunde vara fullständigt lösta och andra helt orörda. Vissa letade i andra lådor efter möjlighet att placera ut samma siffra igen.

Strategier

En majoritet av respondenterna var väldigt strategiska i sitt sökande efter möjliga placeringar. Tre av fyra höll sig, så långt som möjligt, till den redan definierade strategin: naken singel. När denna inte längre upplevdes som applicerbar övergavs den till förmån för mer avancerade strategier. När en mer avancerad strategi väl var använd var det flera som återgick till att leta nakna singlar.

De strategier som användes kan härledas till de strategier som beskrivits tidigare i sektion 2.2. Både dold singel samt dold och naket par användes vid flertalet tillfällen. Dessa upptäcktes genom analys av kandidater.

Vid ett tillfälle användes en strategi som, vid närmare efterforskning, visade sig ha namnet pekande par. Ett pekande par uppstår om en siffra endast kan placeras i två celler i en låda och om cellerna ligger i samma rad eller kolumn. Eftersom siffran måste placeras någonstans i lådan följer det att den inte kan placeras i någon annan cell i den rad eller kolumn som paret delar. Därmed kan siffran tas bort från kandidaterna till övriga celler i raden eller kolumnen.

4.2 Jämförelser

Vid utvärdering användes en av datorerna på KTH Campus i terminalsal Grå. Datorns specifikationer var Intel® Core™2 Quad CPU Q9550 @ 2.83GHz x 4, 3.8 GiB RAM och 64-bitars version av Ubuntu 12.04 LTS.

De tre algoritmer som testades var, som tidigare beskrivna, Totalsökningsalgoritmen (TS), Dancing Links (DL) och Strategibaserade algoritmen (SB).

Innan tidsmätningen kördes testet 10 gånger och varje test upprepades 40 gånger. Lösningstiden för ett sudokupussel beräknas som medianen av dessa 40 värden. Totalt testades algoritmerna mot 725 unika sudokupussel av svårighetsgrad mellan 1.2 och 6.9.

4.2.1 Korrekthet

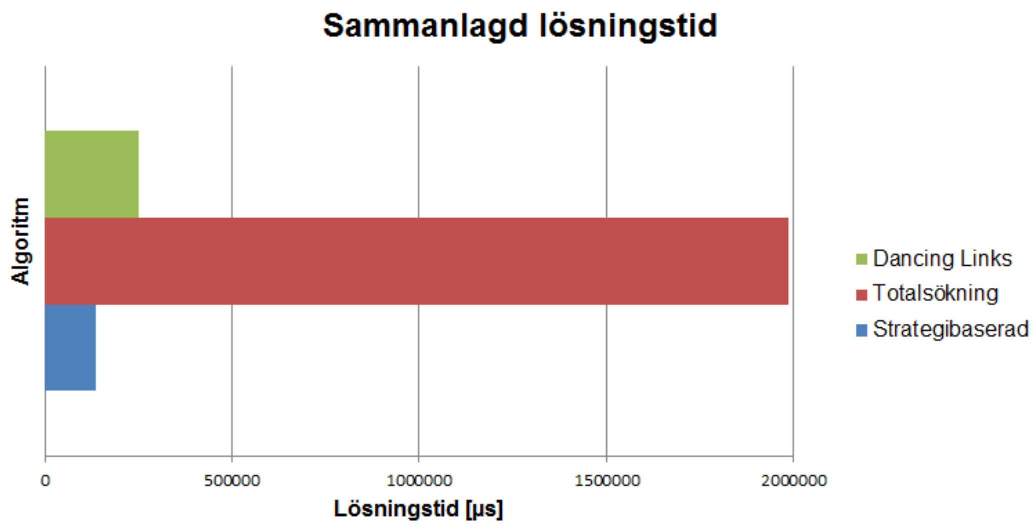
Av de 725 sudokupussel som testades löste både TS och DL samtliga av dem. SB löste 337 av dem med fördelning enligt Tabell 4.1 nedan. Från och med svårighetsgrad 5.6 löste SB inget av sudokupusslen.

Svårighetsgrad	**Antal sudokupussel av denna typ**	**Andel lösta av SB**
1.2-1.7	144	100 %
2.0-2.5	117	100 %
2.6	20	100 %
2.8	10	50 %
3.0	34	100 %
3.2	19	15.79%
3.4	12	91.67%
3.6	2	100 %
4.2-4.7	170	0 %
5.0	2	50 %
> 5.0	195	0 %

Tabell 4.1. Andelen lösta sudokupussel för SB samt totala antalet sudokupussel per svårighetsgrad.

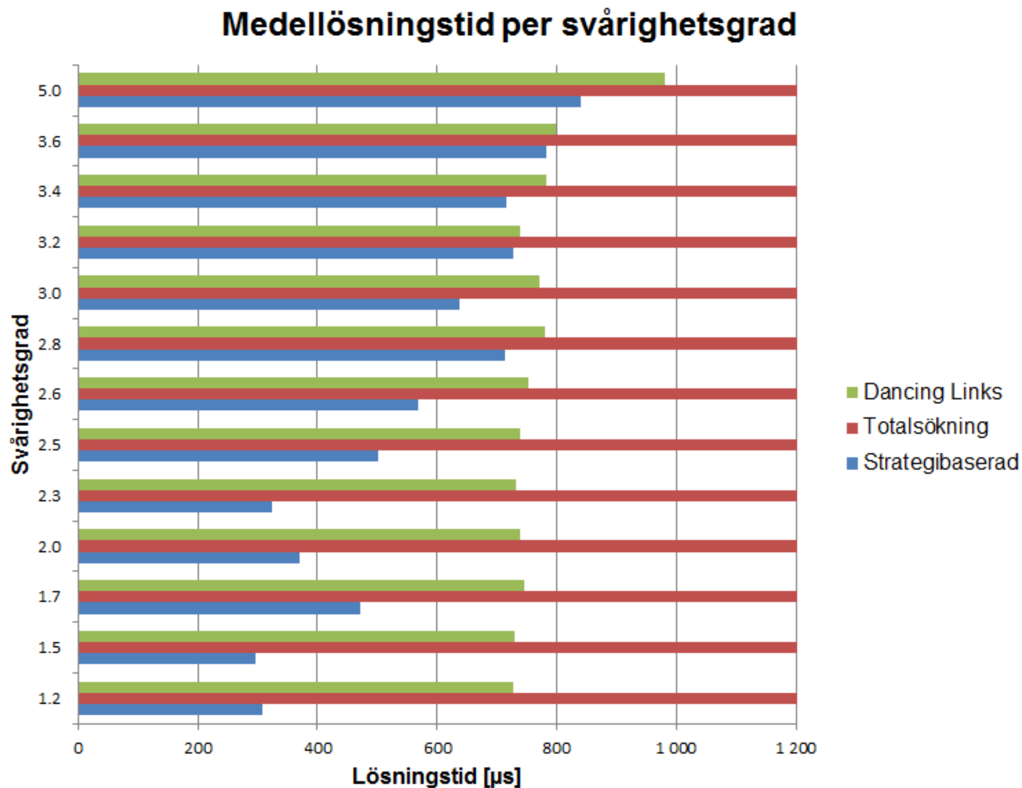
4.2.2 Lösningstid

Figur 4.1 visar den sammanlagda lösningstiden per algoritm för de 337 sudokupussel som samtliga algoritmer lyckades lösa. Tiden mäts i mikrosekunder.



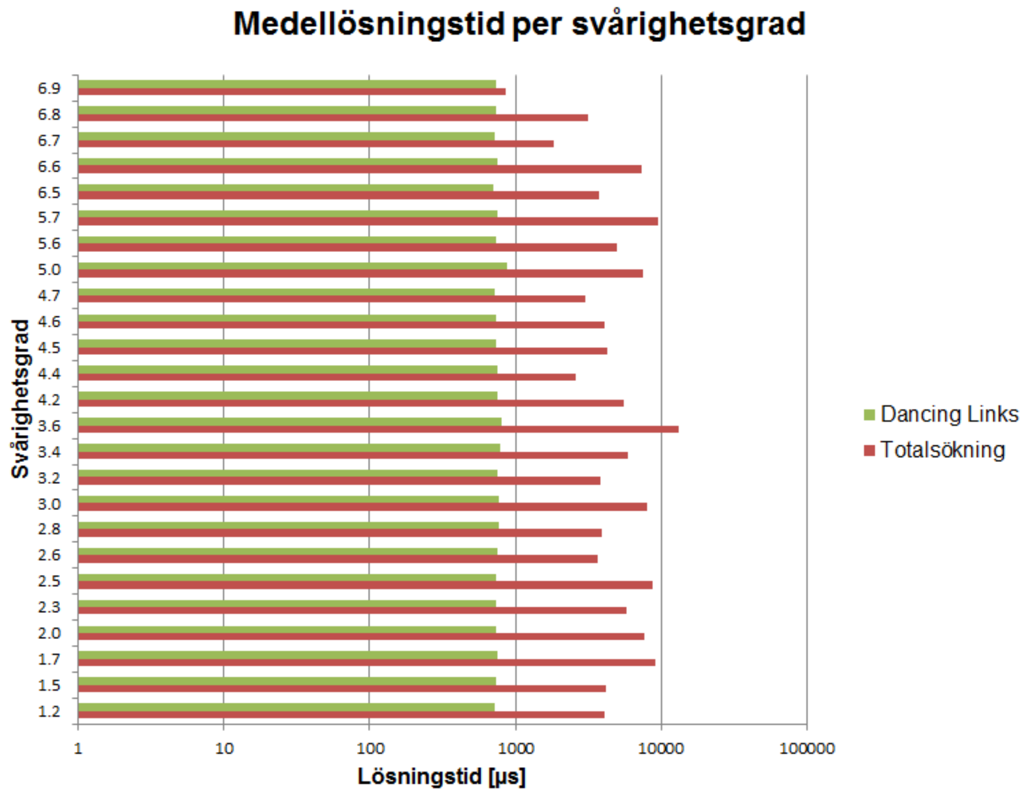
Figur 4.1. Total lösningstid för samtliga sudokupussel per algoritm.

Varje stapel i Figur 4.2 motsvarar medelvärdet av samtliga medianlösningstider för sudokupussel av en given svårighetsgrad och är begränsad till de sudokupussel som samtliga algoritmer lyckades lösa. Figuren visar inte värden större än 1200 mikrosekunder för att tydligare kunna urskilja lägre resultat. För fullständigt resultat för TS och DL se Figur 4.3.



Figur 4.2. Medelvärdet av samtliga medianlösningstider för sudokupussel av en given svårighetsgrad för en viss algoritm.

I Figur 4.3 visas medelvärdet av samtliga medianlösningstider för sudokupussel av en given svårighetsgrad. Resultatet visas för alla 725 sudokupussel i testet för både TS och DL. Tiden är mätt i mikrosekunder och skalan är logaritmisk.



Figur 4.3. Medelvärdet av samtliga medianlösningstider för sudokupussel av en given svårighetsgrad för TS och DL.

Kapitel 5

Diskussion

Detta kapitel inleds med en diskussion kring de resultat som erhållits vid exekveringen av totalsökningsalgoritmen (TS), Dancing Links (DL) och den strategibaserade algoritmen (SB). Sedan diskuteras den undersökning som utfördes följt av resonemang kring valet av svårighetsgradsdefinition. Kapitlet avslutas med en diskussion kring förbättringar till SB som hade varit intressanta att implementera men som inte hunnits med.

5.1 Diskussion av resultat

I denna sektion diskuteras algoritmernas korrekthet och lösningstider. Sektionen avslutas med ett resonemang kring sambandet dem emellan.

5.1.1 Korrekthet

TS och DL löste samtliga sudokupussel, precis som förväntat. SB däremot klarar mindre än hälften av de sudokupussel som användes vid testning. I Tabell 4.1 går det att observera att för svårighetsgrader mindre eller lika med 2.6 klarar den samtliga sudokupussel. En stor andel av testfallen ligger i detta intervall och det är därför rimligt att anta att SB implementerar samtliga strategier som krävs för dessa svårighetsgrader.

För svårighetsgrader mellan 2.8 och 3.6 varierar andelen sudokupussel som SB klarar av att lösa. Det går däremot inte att observera ett direkt samband i intervallet mellan ökande svårighetsgrad och minskande andel lösta sudokupussel. Till exempel klarar SB 91.67 % av de 12 sudokupussel av svårighetsgrad 3.4 men endast 15.79 % av de 19 sudokupussel som har svårighetsgrad 3.2. Det kan dock bero på hur graderingen av sudokupussel genomförs. Den baseras på den svåraste strategin som behövs och tar inte hänsyn till vilka ytterligare strategier som krävs för att lösa pusslet. Det medför att två sudokupussel av samma svårighetsgrad inte nödvändigtvis kräver samma strategier. Om någon strategi som krävs specifikt för det ena pusslet saknas i SB kommer det att fallera medan det andra kan lösas. Fortsätt-

ningsvis innebär det att misslyckade lösningsförsök av svårighetsgrad högre än 2.8 inte nödvändigtvis fallerar på grund av kravet på en ny strategi utan av frånvaron av en strategi som krävdes för 2.8. På detta sätt kan alltså avsaknad av strategier för lägre svårighetsgrader få effekter på resultatet för högre svårighetsgrader. Den låga andelen (15.79 %) lösta sudokupussel av svårighetsgrad 3.2 antyder att en strategi för denna nivå, eller en lägre nivå, saknas. Att ett antal sudokupussel ändå löstes antyder att avsaknaden av en strategi ibland kan ersättas av en mer avancerad sådan.

För svårighetsgrader högre eller lika med 4.2 klarar SB inget av pusslen med undantag för ett sudokupussel av svårighetsgrad 5.0. Det är tydligt att SB saknar strategier för dessa svårighetsgrader. Det är svårt att dra några välgrundade slutsatser utifrån undantaget 5.0 då endast två sudokupussel av denna svårighetsgrad testades. Framgången beror troligtvis på att det sudokupussel SB klarade av krävde utnyttjande av naken kvadrupel och ingen oimplementerad strategi som kan krävas för svårighetsgrad lägre än 5.0. Detta antagande baseras på att naken kvadrupel är den mest avancerade strategin av de implementerade, vilket är rimligt då den involverar flest kombinationer av celler.

Avslutningsvis kan det konstateras att TS och DL är betydligt bättre än SB sett utifrån ett korrekthetsperspektiv samtidigt som SB kräver mer implementerad logik.

5.1.2 Lösningstid

I Figur 4.1 visas totala lösningstiden för de tre algoritmerna för de sudokupussel som samtliga algoritmer löste. Ur denna framgår det tydligt att TS är klart långsammare än både DL och SB. Totaltiden för SB är hälften (54 %) av den för DL. För de sudokupussel som SB klarar av att lösa tycks dess angreppsätt vara gynnsamt för att minimera den totala lösningstiden.

Figur 4.2 visar medelvärdet av lösningstiderna för sudokupussel av en viss svårighetsgrad för varje algoritm. Den visualiserar stora skillnader i hur de tre algoritmernas lösningstid påverkas när svårighetsgraden ökar. Det går att observera en trend på att SB:s lösningstid ökar med svårighetsgraden. TS tar längre tid än både SB och DL för samtliga svårighetsgrader. Tiderna för TS varierar kraftigt mellan olika svårighetsgrader och det tycks inte finnas något direkt samband mellan svårighetsgrad och lösningstid. Tiderna för DL tycks däremot vara relativt konstanta oavsett svårighetsgrad. Dessa observationer förstärks ytterligare av Figur 4.3.

SB utnyttjar endast en strategi i taget och byter till en mer avancerad strategi först när den tidigare inte längre är användbar. Det innebär att om en mer avancerade strategi ska utnyttjas måste samtliga mindre avancerade strategier ha körts och misslyckats. Det är därför ett rimligt resultat att sudokupussel med högre svårighetsgrad tar längre tid att lösa. Det är värt att notera att för svårighetsgrader över 3.0 är skillnaden mellan lösningstiderna för SB och DL mycket liten. Det skulle behövas ytterligare testfall för att kunna uttala sig definitivt men det tycks finnas en brytpunkt i intervallet 3.2–5.0 där DL och SB är lika snabba.

Vid en jämförelse mellan SB och TS är det tydligt att SB är markant snabbare. För de allra svåraste pusslen kan det tänkas att TS skulle kunna vara generellt snabbare än SB eftersom SB skulle behöva iterera över både många och avancerade strategier. Detta antagande är dock svårt att bekräfta med den begränsade mängden sudokupussel av högre svårighetsgrad som SB för närvarande testats på samt klarar av att lösa.

Sammanfattningsvis är SB den klart snabbaste algoritmen, av de tre som testats, för de sudokupussel SB kan lösa. DL har däremot visat upp en tendens av konstant lösningstid och det rimligt att anta att generellt sett är det den snabbaste algoritmen. SB är däremot ett tidseffektivt alternativ för sudokupussel av lägre svårighetsgrad.

5.1.3 Korrekthet kontra lösningstid

Givet att SB, till skillnad från TS och DL, inte kan lösa alla sudokupussel är det intressant att diskutera om och också hur lösningstiden för SB skulle förändras om korrektheten höjs. För att öka andelen lösta sudokupussel skulle nya strategier behöva implementerats. Förutsatt att de nya strategierna läggs till i slutet av den befintliga ordningen kommer lösningstiden för de sudokupussel som den i dagsläget klarar av att lösa inte att förändras. Detta eftersom de nya strategierna aldrig kommer nås. Varje sudokupussel som kräver de nya strategierna kommer däremot ta generellt sett längre tid att lösa då de tidigare strategierna måste testats innan de nya kan nås. Ovanstående förutsätter att nya strategier inte kräver förändringar av datastrukturen som kan påverka samtliga resultat.

Det är därför högst troligt att om nya strategier implementeras kommer SB förbli snabbare än både DL och TS för de sudokupussel där den redan noterats vara snabbare. För sudokupussel med högre svårighetsgrad är det dock, som redan konstaterats, högst troligt att DL är snabbare än SB.

5.2 Undersökningen

Undersökningen bidrog till ökade insikter om vilket angreppssätt människor använder vid lösande av sudokupussel. Projektet hade dock gynnats av att fler interjuver genomförts och på en mer heterogen grupp människor. Antalet personer som intervjuades var få vilket gjorde det svårt att göra några djupare och mer systematiska analyser. Respondenterna tillhörde dessutom en relativt homogen grupp, där samtliga respondenter löste sudokupussel ofta och ansåg sig vara bra på det. Det hade varit intressant att observera hur en som aldrig, eller sällan, löser sudokupussel skulle ta sig an uppgiften. Det hade även varit av intresse att genomföra intervjuer med sudokupussel av högre svårighetsgrad. Med högre svårighetsgrad är det möjligt att fler strategier hade kunnat observeras.

Trots det begränsande underlaget var det möjligt att se vissa trender. Ett exempel på en sådan trend var att samtliga respondenter, efter att ha fastnat och använt en mer avancerad strategi, återgick till att leta nakna singlar. Denna ob-

servation resulterade i att angreppssättet implementerades i den strategibaserade algoritmen. Undersökningens utformning gjorde det lätt att observera strategier och angreppsätt och får därför anses lyckad. Vid ett upprepat försök skulle endast små justeringar göras till upplägget, genom att till exempel minska storleken på sudokupusslen. Intervjuformerna skulle kunna ha varit mer likvärdiga vad gäller miljön i samband med själva lösandet.

5.3 Svårighetsgrad

Från resultaten kan observeras att algoritmerna uppvisade helt olika tendenser när svårighetsgraden förändrades. Dessa tendenser är givetvis direkt beroende på den valda definitionen av svårighetsgrad. I denna sektion diskuteras därför huruvida den valda definitionen verkligen är optimal.

5.3.1 Tid det tar att lösa pusslet

Att tiden det tog att lösa ett sudokupussel skulle vara relaterat till svårighetsgraden kan tyckas vara ett rimligt antagande. I undersökningen kunde observeras att det tog längre tid för samtliga respondenter att lösa sudokupusslet av högre svårighetsgrad. Då skulle det behövas en algoritm som agerar norm för vad som är lång exekveringstid och därigenom vad som är svårt. En sådan definition av svårighetsgrad skulle dock kunna gynna algoritmer som liknar den normsättande algoritmen. Det blir dessutom svårt att bedöma och jämföra algoritmer utifrån exekveringstid och svårighetsgrad om svårighetsgraden är direkt beroende av exekveringstiden.

5.3.2 Antal ledtrådar

Ett enkelt mått för svårighetsgrad är att utgå från antalet givna ledtrådar. Det är dock ett vanligt misstag att anta att ett sudokupussel med färre ledtrådar skulle vara svårare att lösa. Det finns flera exempel på sudokupussel som bara har 17 ledtrådar som att går att lösa med enkel uteslutning. Att använda antalet ledtrådar som mått anses vara ett av de sämsta sätten att klassificera svårighetsgraden på ett sudokupussel[8].

5.3.3 Sudoku Explainers definition

Det kan argumenteras för att den valda defintionen av svårighetsgrad (SE:s) är anpassad efter det angreppssätt som SB använder och att den skulle vara fördelaktig för SB. Då Sudoku är ett pussel utformat för människor och denna defintion på många sätt baseras på den logiska förmågan hos den som löser sudokupusslet måste den dock anses vara bra.

Däremot är ett potentiellt problem med den svårighetsgradsbedömning som SE använder att den inte tar hänsyn till hur många gånger en avancerad strategi behöver användas. Två sudokupussel ges samma svårighetsgrad, trots att den ena kräver

flertalet användningar av en avancerad strategi medan den andra endast kräver att strategin används en gång. En bättre gradering skulle möjligtvis ta hänsyn till detta. En bättre gradering skulle möjligtvis ta hänsyn till detta. På grund av projektets begränsade omfång valdes ändå SE:s gradering och den får, som motiverats tidigare, anses vara den lämpligaste.

5.4 Vid återupprepning av projektet

Det finns ett antal punkter skulle kunna ha gjorts annorlunda i projektet och sådant som fortfarande finns kvar att utreda. Det hade varit intressant att undersöka hur algoritmerna presterar när storleken på sudokupusslet ökar. Hur skulle det till exempel sett ut om ett sudokupussel av storlek 16x16 eller 25x25 använts?

Det finns flertalet utökade aspekter av SB som skulle kunna undersökas. En sådan sak är hur ändringar i ordningen av strategierna i SB hade påverkat dess resultat vad gäller lösningstid. En annan infallsvinkel på detta hade varit att ändra hur algoritmen reagerar när en strategi lyckas. Är det tidsmässigt optimalt att gå tillbaka till den simplaste strategin eller ska algoritmen stega sig nedåt? Det hade också varit intressant att implementera fler strategier för att kunna dra mer välgrundade slutsatser kring SB:s effektivitet relativt de två andra algoritmerna.

I implementationen utelämnades vissa idéer till SB som erhölls från undersökningen. Flera respondenter fokuserade sitt sökande kring de zoner där en siffra precis hade placerats. Samtliga respondenter använde dessutom endast kandidater om de ansåg sig tvungna. SB tar dock fram kandidaterna till alla celler från början. Det vore intressant att se hur resultaten för SB skulle påverkats om dessa två idéer hade implementerats då det hade gett algoritmen ett angreppssätt som ännu mer motsvarade det som respondenterna använde.

Slutligen hade ett annat projekt som varit intressant att genomföra varit att konstruera ett eget program för generering och gradering av sudokupussel. På det sättet skulle graderingen kunna anpassas för att till exempel väga in antalet strategier av varje svårighet som ett sudokupussel kräver.

Kapitel 6

Slutsatser

Ur ett korrekthetsperspektiv har den strategibaserade algoritmen svårt att mäta sig med totalsöknings- och Dancing Links-algoritmerna som båda klarar av att lösa sudokupussel av samtliga svårighetsgrader. Den mängd kod och tid som krävs för att implementera en strategibaserad algoritm som kan lösa sudokupussel av högre svårighetsgrad är dessutom betydligt större än den som krävs för de övriga algoritmerna. Det är även osäkert om de strategier som finns idag räcker till för att lösa de allra svåraste sudokupusslen.

Trots detta är den strategibaserade algoritmen intressant av andra anledningar. Skiljt från de andra två algoritmerna skulle den bland annat kunna användas för att visualisera en relevant lösningsprocess och genom detta, steg för steg lära ut lösningsstrategier till en observatör. Den strategibaserade algoritmen är dessutom snabbare än de två andra algoritmerna för de sudokupussel som den klarade. Den uppvisar dock ett beteende av ökande lösningstider för högre svårighetsgrader och det är därför troligt att Dancing Links är den generellt snabbaste algoritmen då den visar på en konstant lösningstid oberoende av svårighetsgrad.

Avslutningsvis, trots att SB inte kan användas för att lösa samtliga sudokupussel kan den vid lösning av sudokupussel av den svårighetsgrad som vanligtvis förekommer i dagstidningar vara ett tidseffektivt alternativ.

Litteraturförteckning

[1] D. E. Knuth, “Dancing Links,” tech. rep., Stanford University.

[2] M. Boström, Stora Sudokuboken. Känguru.

[3] G. C. Bastian Tugemann, “There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem,” tech. rep., University College Dublin.

[4] A. C. Stuart, “Sudoku Creation and Grading,” tech. rep., Syndicated Puzzles Inc.

[5] J. Chu, “A Sudoku Solver in Java implementing Knuth’s Dancing Links Algorithm,” tech. rep., Berkeley University of California.

[6] T. S. Takayuki Yato, “Complexity and Completeness of Finding Another Solution and Its Application to Puzzles,” tech. rep., The University of Tokyo.

[7] J. Karlander, “Np-problem.” `http://www.csc.kth.se/utbildning/kth/kurser/DD2354/algokomp07/For0707+08.pdf`. Hämtad 2013-04-02.

[8] N. Juillerat, “Sudoku Explainer.” `http://diuf.unifr.ch/people/juillera/Sudoku/Sudoku.html`, december 2007. Hämtad 2013-03-14.

[9] J. Zelenski. `http://see.stanford.edu/materials/icspacs106b/Lecture11.pdf`. Hämtad 2013-04-02.

[10] D. Seiler, “Dancing Sudoku.” `http://dancingsudoku.sourceforge.net/`. Hämtad 2013-04-04.

[11] B. Boyer, “Robust Java benchmarking, Part 1: Issues. Understand the pitfalls of benchmarking Java code,” tech. rep., IBM.

Bilaga A

Sudokupussel som använts i undersökningen

			7			5	8	1
	9						6	7
7			3		5		4	
			8		7			
		3	4			7		8
		4			1		2	
			5	7	8			
				9		8		3
2	5					9		6

Figur A.1. Medelsvårt sudokupussel som använts i undersökningen

5	9					7		
7			1			5		9
	8					3	6	1
				2			9	
9				4	3			
6					9	4		7
				5	1	9		
	4					6	7	
					4		3	

Figur A.2. Svårt sudokupussel som använts i undersökningen

Bilaga B

Källkod för Strategibaserad algoritm

Kod B.1. Solver.java

```
1  package se.pellbybrodin.sudokusolver.models;
2
3  public interface Solver {
4
5    public int[][] solve(int[][] board);
6
7    public String getName();
8
9  }
```

Kod B.2. HumanSolver.java

```
1  package se.pellbybrodin.sudokusolver.solvers.human;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import se.pellbybrodin.sudokusolver.models.Solver;
7  import se.pellbybrodin.sudokusolver.solvers.human.rules.HiddenPairRule;
8  import
       se.pellbybrodin.sudokusolver.solvers.human.rules.HiddenSingleRule;
9  import se.pellbybrodin.sudokusolver.solvers.human.rules.NakedPairRule;
10 import se.pellbybrodin.sudokusolver.solvers.human.rules.NakedQuadRule;
11 import
       se.pellbybrodin.sudokusolver.solvers.human.rules.NakedSingleRule;
12 import
       se.pellbybrodin.sudokusolver.solvers.human.rules.NakedTrippelRule;
13 import se.pellbybrodin.sudokusolver.solvers.human.rules.PointingPair;
14
15 public class HumanSolver implements Solver {
16
17   @Override
18   public int[][] solve(int[][] init) {
19     final List<Rule> rules = new ArrayList<Rule>();
20     final HumanBoard board = new HumanBoard(init);
21
```

```
22      rules.add(new NakedSingleRule());
23      rules.add(new HiddenSingleRule());
24      rules.add(new PointingPair());
25      rules.add(new NakedPairRule());
26      rules.add(new HiddenPairRule());
27      rules.add(new NakedTrippelRule());
28      rules.add(new NakedQuadRule());
29
30      boolean alterd = true;
31      while (!board.isSolved() && alterd) {
32        for (final Rule rule : rules) {
33          if (rule.apply(board)) {
34            alterd = true;
35            break;
36          } else {
37            alterd = false;
38          }
39        }
40      }
41
42      return board.getPrimitiveBoard();
43    }
44
45    @Override
46    public String getName() {
47      return HumanSolver.class.getSimpleName();
48    }
49
50  }
```

Kod B.3. Board.java

```
1   package se.pellbybrodin.sudokusolver.models;
2
3   public abstract class Board {
4
5     protected int size;
6     protected int order;
7
8     public Board(int size) {
9       this.size = size;
10      this.order = (int) Math.sqrt(size);
11    }
12
13    /**
14     * Name of the board
15     */
16    private String boardName;
17
18    /**
19     * Returns the name of the board
20     *
21     * @return name of the board
22     */
```

```
23    public String getBoardName() {
24      return this.boardName;
25    }
26
27    /**
28     * Sets the name of the board to the given string
29     *
30     * @param name
31     *            the new name to be set
32     */
33    public void setBoardName(String name) {
34      this.boardName = name;
35    }
36
37    /**
38     * Returns the value in the cell at specific row and column in the
          board
39     *
40     * @param row
41     *            the cells row (0-8)
42     * @param column
43     *            the cells column (0-8)
44     * @return the value in the cell at a specific row and column on the
          board
45     * @throws IndexOutOfBoundsException
46     */
47    public abstract int get(int row, int column)
48        throws IndexOutOfBoundsException;
49
50    /**
51     * Place a value on the board at position [row, column]
52     *
53     * @param row
54     *            the cells row (0-8)
55     * @param column
56     *            the cells column (0-8)
57     * @param value
58     *            the value which should be placed in the cell
59     */
60    public abstract void place(int row, int column, int value);
61
62    /**
63     * Clear the value from the board at position [row, column]
64     *
65     * @param row
66     *            the cells row (0-8)
67     * @param column
68     *            the cells column (0-8)
69     */
70    public abstract void clear(int row, int column);
71
72    /**
73     * Returns <code>true</code> if the given value can be placed at
          [row,
```

```
74       * column], <code>false</code> otherwise.
75       *
76       * @param row
77       *            the cells row (0-8)
78       * @param column
79       *            the cells column (0-8)
80       * @param value
81       *            the value which should be checked
82       * @return <code>true</code> if the given value can be placed at
              [row,
83       *         column], <code>false</code> otherwise.
84       */
85      public abstract boolean canPlace(int row, int column, int value);
86
87      /**
88       * Return the box index of the cell place at row/column
89       *
90       * @param row
91       * @param column
92       * @return
93       */
94      public int getBox(int row, int column) {
95        return row / this.order * this.order + column / this.order;
96      }
97
98      /**
99       * Returns the board represented as a primitive 2D array.
100      *
101      * @return the board as a 2D array
102      */
103     public abstract int[][] getPrimitiveBoard();
104
105     /**
106      * Returns <code>true</code> if the sudoku is solved, that is, if
             all tiles
107      * are filled and no conflicts exists. Else, it returns
            <code>false</code>.
108      *
109      * @return <code>true</code> if this board is solved, else
110      *         <code>false</code>
111      */
112     public boolean isSolved() {
113       final int ROW = 0, COLUMN = 1, BOX = 2;
114       final boolean[][][] placed = new boolean[3][9][9];
115
116       for (int row = 0; row < 9; row++) {
117         for (int col = 0; col < 9; col++) {
118           final int value = get(row, col);
119           if (value == 0) {
120             return false;
121           }
122           if (!placed[ROW][row][value - 1]
123               && !placed[COLUMN][col][value - 1]
124               && !placed[BOX][getBox(row, col)][value - 1]) {
```

```
125           placed[ROW][row][value - 1] = true;
126           placed[COLUMN][col][value - 1] = true;
127           placed[BOX][getBox(row, col)][value - 1] = true;
128         } else {
129           return false;
130         }
131       }
132     }
133     return true;
134   }
135 
136   @Override
137   public String toString() {
138     final StringBuilder sb = new StringBuilder();
139 
140     for (int i = 0; i < this.size; i++) {
141       sb.append("\n");
142       for (int j = 0; j < this.size; j++) {
143         final int value = get(i, j);
144         if (value == 0) {
145           sb.append(".␣");
146         } else {
147           sb.append(value + "␣");
148         }
149       }
150     }
151     return sb.toString();
152   }
153 
154 }
```

Kod B.4. HumanBoard.java

```
1  package se.pellbybrodin.sudokusolver.solvers.human;
2  
3  import java.util.ArrayList;
4  import java.util.Collection;
5  import java.util.List;
6  
7  import se.pellbybrodin.sudokusolver.models.Board;
8  import se.pellbybrodin.sudokusolver.models.Cell;
9  
10 public class HumanBoard extends Board {
11 
12   /**
13    * A 2-dimensional array holding information on each cell.
14    */
15   private Cell[][] cells;
16 
17   /**
18    * Rows, columns and boxes of the board
19    */
20   private Row[] rows;
21   private Column[] columns;
```

```
22    private Box[] boxes;
23
24    /**
25     * All unsolved cells of the board
26     */
27    private ArrayList<Cell> unsolved;
28
29    /**
30     * Constructor
31     *
32     * @param primitive
33     *            a 2d array representation of the puzzle
34     */
35    public HumanBoard(int[][] primitive) {
36      super(primitive.length);
37      init(primitive);
38    }
39
40    /*
41     * INIT
42     */
43
44    /**
45     * Initializing function
46     *
47     * @param primitive
48     *            a 2d array representation of the puzzle
49     */
50    private void init(int[][] primitive) {
51
52      this.order = (int) Math.sqrt(this.size);
53      this.cells = new Cell[this.size][this.size];
54      this.unsolved = new ArrayList<Cell>();
55
56      // Setup the board
57      for (int row = 0; row < this.size; row++) {
58        for (int col = 0; col < this.size; col++) {
59          this.cells[row][col] = new Cell(row, col, false);
60        }
61      }
62
63      // Setup references to rows, columns and boxes
64      this.rows = new Row[9];
65      this.columns = new Column[9];
66      this.boxes = new Box[9];
67      for (int i = 0; i < 9; i++) {
68        this.rows[i] = new Row(i);
69        this.columns[i] = new Column(i);
70        this.boxes[i] = new Box(i);
71      }
72
73      // Place the values
74      for (int row = 0; row < this.size; row++) {
75        for (int col = 0; col < this.size; col++) {
```

```
76          final int value = primitive[row][col];
77          if (value != 0) {
78            place(row, col, value);
79          }
80        }
81      }
82  
83      // Setup candidates values;
84      setupCandidates();
85  
86      // Add unsolved cells to queue
87      for (final Cell[] array : this.cells) {
88        for (final Cell cell : array) {
89          if (cell.getNumberOfCandidates() != 0) {
90            this.unsolved.add(cell);
91          }
92        }
93      }
94    }
95  
96    private void setupCandidates() {
97      // For each number
98      for (final Row row : this.rows) {
99        for (final Cell cell : row.getUnsolvedCells()) {
100         for (int i = 1; i <= this.size; i++) {
101           this.cells[cell.getRow()][cell.getColumn()].addCandidate(i);
102         }
103       }
104     }
105 
106     cleanupCandidates();
107 
108   }
109 
110   private void cleanupCandidates() {
111     for (int i = 0; i < this.size; i++) {
112       cleanupCandidatesInZone(this.rows[i]);
113       cleanupCandidatesInZone(this.columns[i]);
114       cleanupCandidatesInZone(this.boxes[i]);
115     }
116   }
117 
118   private void cleanupCandidatesInZone(Zone zone) {
119     final List<Integer> placed = new ArrayList<Integer>();
120     // Find all numbers that has been placed in the zone
121     for (final Cell cell : zone.getCells()) {
122       if (cell.isSolved()) {
123         placed.add(cell.getValue());
124       }
125     }
126 
127     // Remove them from all cells in the zone
128     for (final Cell cell : zone.getCells()) {
129       removeCandidates(cell, placed);
```

```
130         }
131     }
132 
133     @Override
134     public int[][] getPrimitiveBoard() {
135         final int[][] primitive = new int[this.size][this.size];
136         for (int row = 0; row < this.size; row++) {
137             for (int col = 0; col < this.size; col++) {
138                 primitive[row][col] = this.cells[row][col].getValue();
139             }
140         }
141         return primitive;
142     }
143 
144     /*
145      * GET CELLS
146      */
147 
148     @Override
149     public int get(int row, int column) {
150         return this.cells[row][column].getValue();
151     }
152 
153     public Cell getCell(int row, int column) {
154         return this.cells[row][column];
155     }
156 
157     public List<Cell> getUnsolvedCells() {
158         return new ArrayList<Cell>(this.unsolved);
159     }
160 
161     /*
162      * ROWS
163      */
164 
165     public Row[] getRows() {
166         return this.rows;
167     }
168 
169     public List<Cell> getCellsInRow(int row) {
170         return this.rows[row].getCells();
171     }
172 
173     public List<Cell> getCellsInSameRow(int row, int column) {
174         final List<Cell> rowCells = getCellsInRow(row);
175         rowCells.remove(column);
176         return rowCells;
177     }
178 
179     public List<Cell> getUnsolvedInRow(int row) {
180         return this.rows[row].getUnsolvedCells();
181     }
182 
183     /*
```

```
184      * COLUMNS
185      */
186
187     public Column[] getColumns() {
188       return this.columns;
189     }
190
191     public List<Cell> getCellsInColumn(int column) {
192       return this.columns[column].getCells();
193     }
194
195     public List<Cell> getCellsInSameColumn(int row, int column) {
196       final List<Cell> colCells = getCellsInColumn(column);
197       colCells.remove(row);
198       return colCells;
199     }
200
201     public List<Cell> getUnsolvedInColumn(int column) {
202       return this.columns[column].getUnsolvedCells();
203     }
204
205     public int getBoxNumber(int row, int column) {
206       return row / 3 * 3 + column / 3;
207     }
208
209     public Box[] getBoxes() {
210       return this.boxes;
211     }
212
213     public List<Cell> getCellsInSameBox(int row, int column) {
214       final int box = getBoxNumber(row, column);
215       final List<Cell> boxCells = getCellsInBox(box);
216       boxCells.remove(this.cells[row][column]);
217       return boxCells;
218     }
219
220     public List<Cell> getCellsInBox(int box) {
221       return this.boxes[box].getCells();
222     }
223
224     public List<Cell> getUnsolvedInBox(int box) {
225       return this.boxes[box].getUnsolvedCells();
226     }
227
228     /*
229      * PLACING
230      */
231
232     @Override
233     public boolean canPlace(int row, int column, int value) {
234       return this.cells[row][column].hasCandidate(value);
235     }
236
237     @Override
```

```
238     public void place(int row, int column, int value) {
239         this.cells[row][column].setValue(value);
240         removeValueFromAdjacentCellCandidates(row, column, value);
241         this.unsolved.remove(this.cells[row][column]);
242     }
243
244     public void place(Cell cell, int value) {
245         place(cell.getRow(), cell.getColumn(), value);
246     }
247
248     @Override
249     public void clear(int row, int column) {
250         this.cells[row][column].setValue(0);
251     }
252
253     /*
254      * CANDIDATES
255      */
256
257     /**
258      * Removes the given value for all cells in the same row, column and
            box as
259      * the cell specified by the parameters
260      *
261      * @param row
262      *            the row of the cell
263      * @param column
264      *            the column of the cell
265      * @param value
266      *            the value to be removed
267      */
268     private void removeValueFromAdjacentCellCandidates(int row, int
            column,
269             int value) {
270         /* Remove all candidates from cell */
271         this.cells[row][column].removeAllCandidates();
272
273         final int box = getBoxNumber(row, column);
274         /* Remove from column cells */
275         for (int i = 0; i < this.size; i++) {
276             this.rows[row].getCell(i).removeCandidate(value);
277             this.columns[column].getCell(i).removeCandidate(value);
278             this.boxes[box].getCell(i).removeCandidate(value);
279         }
280     }
281
282     public boolean removeCandidate(Cell cell, int value) {
283         return this.cells[cell.getRow()][cell.getColumn()]
284                 .removeCandidate(value);
285     }
286
287     public boolean removeCandidates(Cell cell, Collection<Integer>
            candidates) {
288         return this.cells[cell.getRow()][cell.getColumn()]
```

```
289         .removeCandidates(candidates);
290   }
291
292   public boolean retainCandidates(Cell cell, Collection<Integer> pair)
        {
293     return this.cells[cell.getRow()][cell.getColumn()]
294         .retainCandidates(pair);
295   }
296
297   /*
298    * ZONES
299    */
300
301   public abstract class Zone {
302
303     public boolean isSolved(int index) {
304       return getCell(index).isSolved();
305     }
306
307     public abstract Cell getCell(int index);
308
309     public List<Cell> getCells() {
310       final List<Cell> cells = new ArrayList<Cell>();
311       for (int i = 0; i < HumanBoard.this.size; i++) {
312         cells.add(getCell(i));
313       }
314       return cells;
315     }
316
317     public List<Cell> getUnsolvedCells() {
318       final List<Cell> cells = getCells();
319       final List<Cell> unsolved = new ArrayList<Cell>();
320       for (final Cell cell : cells) {
321         if (!cell.isSolved()) {
322           unsolved.add(cell);
323         }
324       }
325       return unsolved;
326     }
327
328     public List<Integer> potentialPositions(int value) {
329       final List<Integer> potential = new ArrayList<Integer>();
330       for (int i = 0; i < HumanBoard.this.size; i++) {
331         if (getCell(i).hasCandidate(value)) {
332           potential.add(i);
333         }
334       }
335       return potential;
336     }
337
338   }
339
340   public class Row extends Zone {
341
```

```
342     int rowNumber;
343
344     public Row(int rowNumber) {
345       this.rowNumber = rowNumber;
346     }
347
348     @Override
349     public Cell getCell(int index) {
350       return HumanBoard.this.cells[this.rowNumber][index];
351     }
352
353   }
354
355   public class Column extends Zone {
356
357     int columnNumber;
358
359     public Column(int columnNumber) {
360       this.columnNumber = columnNumber;
361     }
362
363     @Override
364     public Cell getCell(int index) {
365       return HumanBoard.this.cells[index][this.columnNumber];
366     }
367   }
368
369   public class Box extends Zone {
370
371     int boxNumber;
372     int startX;
373     int startY;
374
375     public Box(int boxNumber) {
376       this.boxNumber = boxNumber;
377       this.startX = boxNumber / HumanBoard.this.order
378           * HumanBoard.this.order;
379       this.startY = boxNumber % HumanBoard.this.order
380           * HumanBoard.this.order;
381     }
382
383     @Override
384     public Cell getCell(int index) {
385       final int x = index / HumanBoard.this.order;
386       final int y = index % HumanBoard.this.order;
387       return HumanBoard.this.cells[this.startX + x][this.startY + y];
388     }
389
390   }
391
392 }
```

Kod B.5. Cell.java

```
1  package se.pellbybrodin.sudokusolver.models;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5  import java.util.HashSet;
6  import java.util.List;
7
8  public class Cell {
9
10   private int row;
11   private int column;
12   private int value;
13   private HashSet<Integer> candidates;
14
15   public Cell(int row, int column) {
16     init(row, column, 0, false);
17   }
18
19   public Cell(int row, int column, boolean fillCandidates) {
20     init(row, column, 0, true);
21   }
22
23   public Cell(int row, int column, int value) {
24     init(row, column, value, true);
25   }
26
27   /**
28    * Initializer
29    *
30    * Sets up the cell and fills the candidates if set.
31    *
32    * @param row
33    *            the row of the cell
34    * @param column
35    *            the column of the cell
36    * @param value
37    *            the value of the cell
38    * @param fillCandidates
39    *            <code>true</code> if the candidates should be filled
40    */
41   private void init(int row, int column, int value, boolean
         fillCandidates) {
42     this.row = row;
43     this.column = column;
44     this.value = value;
45
46     this.candidates = new HashSet<Integer>();
47     if (fillCandidates) {
48       for (int i = 1; i <= 9; i++) {
49         this.candidates.add(i);
50       }
51     }
52
53   }
```

```
54
55    /**
56     * @return the value of the cell
57     */
58    public int getValue() {
59      return this.value;
60    }
61
62    /**
63     * @param value
64     *            the value to set
65     */
66    public void setValue(int value) {
67      this.value = value;
68    }
69
70    public boolean isSolved() {
71      return this.value != 0;
72    }
73
74    /**
75     * @return the row of the cell
76     */
77    public int getRow() {
78      return this.row;
79    }
80
81    public boolean sameRowAs(Cell cell) {
82      return getRow() == cell.getRow();
83    }
84
85    /**
86     * @return the column of the cell
87     */
88    public int getColumn() {
89      return this.column;
90    }
91
92    public boolean sameColumnAs(Cell cell) {
93      return getColumn() == cell.getColumn();
94    }
95
96    public boolean hasCandidate(int value) {
97      return this.candidates.contains(value);
98    }
99
100   public int getNumberOfCandidates() {
101     return this.candidates.size();
102   }
103
104   /**
105    * @return the candidates for this cell as a list
106    */
107   public List<Integer> getCandidates() {
```

```
108       return new ArrayList<Integer>(this.candidates);
109     }
110
111     /**
112      * @return the candidates for this cell as a set
113      */
114     public HashSet<Integer> getCandidatesSet() {
115       return this.candidates;
116     }
117
118     /**
119      * Remove the given number from the candidates of this cell. If the
            number
120      * is not a candidate, it does nothing.
121      *
122      * @param number
123      *            the number to be removed
124      * @return
125      */
126     public boolean removeCandidate(int number) {
127       return this.candidates.remove(number);
128     }
129
130     public void addCandidate(int number) {
131       this.candidates.add(number);
132     }
133
134     /**
135      * Remove the given numbers from the candidates of this cell. If a
            number is
136      * not a candidate, it does nothing.
137      *
138      * @param number
139      *            the number to be removed
140      * @return
141      */
142     public boolean removeCandidates(Collection<Integer> candidates2) {
143       boolean bool = false;
144       for (final Integer number : candidates2) {
145         if (this.candidates.remove(number)) {
146           bool = true;
147         }
148       }
149       return bool;
150     }
151
152     public boolean retainCandidates(Collection<Integer> pair) {
153       return this.candidates.retainAll(pair);
154     }
155
156     /**
157      * Removes all candidates from the cell.
158      */
159     public void removeAllCandidates() {
```

```
160     this.candidates.clear();
161   }
162
163   @Override
164   public String toString() {
165     return "[" + this.row + ", " + this.column + "] value= " +
          this.value
166         + ", candidates=" + this.candidates.toString();
167   }
168
169 }
```

Kod B.6. Rule.java

```
1  package se.pellbybrodin.sudokusolver.solvers.human;
2
3  public interface Rule {
4
5    /**
6     * Tries to apply this rule on the board and returns
           <code>true</code> if it
7     * is successful.
8     *
9     * @param board
10    *            a reference to the sudoku board to be solved
11    * @return <code>true</code> if the rule altered the board in any
           way,
12    *         <code>false</code> otherwise.
13    */
14   public boolean apply(HumanBoard board);
15
16 }
```

Kod B.7. NakedSingleRule.java

```
1  package se.pellbybrodin.sudokusolver.solvers.human.rules;
2
3  import se.pellbybrodin.sudokusolver.models.Cell;
4  import se.pellbybrodin.sudokusolver.solvers.human.HumanBoard;
5  import se.pellbybrodin.sudokusolver.solvers.human.Rule;
6
7  /**
8   * This rule is special - its the only rule that places values on the
         board.
9   */
10 public class NakedSingleRule implements Rule {
11
12   @Override
13   public boolean apply(HumanBoard board) {
14     boolean altered = false;
15     // For each unsolved cell
16     for (final Cell cell : board.getUnsolvedCells()) {
17       // If the cell has only one unsolved candidate, it's a naked
             single
```

```
18       if (cell.getNumberOfCandidates() == 1) {
19         board.place(cell, cell.getCandidates().get(0));
20         altered = true;
21       }
22     }
23     return altered;
24   }
25 }
```

Kod B.8. NakedPairRule.java

```
1  package se.pellbybrodin.sudokusolver.solvers.human.rules;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import se.pellbybrodin.sudokusolver.models.Cell;
7  import se.pellbybrodin.sudokusolver.solvers.human.HumanBoard;
8  import se.pellbybrodin.sudokusolver.solvers.human.HumanBoard.Zone;
9  import se.pellbybrodin.sudokusolver.solvers.human.Rule;
10
11 /**
12  * Removes candidates in the board bases on the naked pair rule.
13  */
14 public class NakedPairRule implements Rule {
15
16   @Override
17   public boolean apply(HumanBoard board) {
18
19     if (checkZones(board, board.getRows())) {
20       return true;
21     }
22     if (checkZones(board, board.getColumns())) {
23       return true;
24     }
25     if (checkZones(board, board.getBoxes())) {
26       return true;
27     }
28
29     return false;
30
31   }
32
33   public boolean checkZones(HumanBoard board, Zone[] zones) {
34     for (final HumanBoard.Zone zone : zones) {
35       if (checkCells(board, zone.getUnsolvedCells())) {
36         return true;
37       }
38     }
39     return false;
40
41   }
42
43   public boolean checkCells(HumanBoard board, List<Cell> cells) {
```

```
44      final List<Cell> pairs = new ArrayList<Cell>();
45
46      // Find all pairs
47      for (final Cell cell : cells) {
48        if (cell.getNumberOfCandidates() == 2) {
49          pairs.add(cell);
50        }
51      }
52
53      // For each unique pair
54      for (int i = 0; i < pairs.size(); i++) {
55        for (int j = i + 1; j < pairs.size(); j++) {
56          final Cell first = pairs.get(i);
57          final Cell second = pairs.get(j);
58          // If they have the same candidates, it's a naked pair
59          if
                (first.getCandidatesSet().equals(second.getCandidatesSet()))
                {
60            final List<Integer> candidates = first.getCandidates();
61            // Remove their candidates from all other cells in the zone
62            boolean altered = false;
63            for (final Cell cell : cells) {
64              if (cell.equals(first) || cell.equals(second)) {
65                continue;
66              }
67              if (board.removeCandidates(cell, candidates)) {
68                altered = true;
69              }
70            }
71            return altered;
72          }
73        }
74      }
75      return false;
76    }
77  }
```

Kod B.9. NakedTrippelRule.java

```
1   package se.pellbybrodin.sudokusolver.solvers.human.rules;
2
3   import java.util.ArrayList;
4   import java.util.HashSet;
5   import java.util.List;
6
7   import se.pellbybrodin.sudokusolver.models.Cell;
8   import se.pellbybrodin.sudokusolver.solvers.human.HumanBoard;
9   import se.pellbybrodin.sudokusolver.solvers.human.Rule;
10
11  public class NakedTrippelRule implements Rule {
12
13    @Override
14    public boolean apply(HumanBoard board) {
15      return applyTupleSize(board, 3);
```

```
16    }
17
18    public boolean applyTupleSize(HumanBoard board, int tupleSize) {
19      for (int nr = 0; nr < 9; nr++) {
20        if (help(board, board.getCellsInRow(nr), tupleSize)) {
21          return true;
22        }
23        if (help(board, board.getCellsInColumn(nr), tupleSize)) {
24          return true;
25        }
26        if (help(board, board.getCellsInBox(nr), tupleSize)) {
27          return true;
28        }
29      }
30      return false;
31    }
32
33    private boolean help(HumanBoard board, List<Cell> zone, int
          tupleSize) {
34      final List<Cell> potentialTuple = new ArrayList<Cell>();
35      // Find all non-empty cells that has number of candidates <= to
36      // tupleSize
37      for (final Cell cell : zone) {
38        if (!(cell.getNumberOfCandidates() == 0)
39            && cell.getCandidatesSet().size() <= tupleSize) {
40          potentialTuple.add(cell);
41        }
42      }
43
44      // Use every found cell as starting point for finding a triple.
45      for (final Cell originCell : potentialTuple) {
46        final HashSet<Integer> candidates = new HashSet<Integer>(
47            originCell.getCandidatesSet());
48        final HashSet<Cell> tuples = new HashSet<Cell>(); // Keep track
            of
49                                      // added cells
50        tuples.add(originCell);
51
52        // Add all cells which contains a subset of the originCell
53        for (final Cell cell : potentialTuple) {
54          if (candidates.containsAll(cell.getCandidatesSet())) {
55            tuples.add(cell);
56          }
57        }
58
59        // If we have tupleSize number of cells in our list, we have
            found a
60        // hidden tuple.
61        if (tuples.size() == tupleSize) {
62          boolean altered = false;
63          zone.removeAll(tuples);
64          // Remove the tuples candidates from all other cells in the
65          // zone.
66          for (final Cell cell : zone) {
```

```
67            if (!tuples.contains(cell)) {
68              if (board.removeCandidates(cell, candidates)) {
69                altered = true;
70              }
71            }
72          }
73          if (altered) {
74            return true;
75          }
76        }
77      }
78      return false;
79    }
80
81  }
```

Kod B.10. NakedQuadRule.java

```
1   package se.pellbybrodin.sudokusolver.solvers.human.rules;
2
3   import se.pellbybrodin.sudokusolver.solvers.human.HumanBoard;
4   import se.pellbybrodin.sudokusolver.solvers.human.Rule;
5
6   public class NakedQuadRule implements Rule {
7
8     @Override
9     public boolean apply(HumanBoard board) {
10      return new NakedTrippelRule().applyTupleSize(board, 4);
11    }
12
13  }
```

Kod B.11. HiddenSingleRule.java

```
1   package se.pellbybrodin.sudokusolver.solvers.human.rules;
2
3   import java.util.ArrayList;
4   import java.util.List;
5
6   import se.pellbybrodin.sudokusolver.models.Cell;
7   import se.pellbybrodin.sudokusolver.solvers.human.HumanBoard;
8   import se.pellbybrodin.sudokusolver.solvers.human.HumanBoard.Zone;
9   import se.pellbybrodin.sudokusolver.solvers.human.Rule;
10
11  public class HiddenSingleRule implements Rule {
12
13    @Override
14    public boolean apply(HumanBoard board) {
15      if (findHiddenSingle(board, board.getBoxes())) {
16        return true;
17      }
18      if (findHiddenSingle(board, board.getRows())) {
19        return true;
20      }
```

```
21      if (findHiddenSingle(board, board.getColumns())) {
22        return true;
23      }
24
25      return false;
26    }
27
28    private boolean findHiddenSingle(HumanBoard board, Zone[] zones) {
29      List<Integer> potential;
30      // For each value
31      for (int value = 1; value <= 9; value++) {
32        // For each zone
33        for (final Zone zone : zones) {
34          // Check the potential positions for the value
35          potential = zone.potentialPositions(value);
36          // If the value only has one position, it's a hidden single.
37          if (potential.size() == 1) {
38            final Cell single = zone.getCell(potential.get(0));
39            final List<Integer> candidates = new ArrayList<Integer>(
40                single.getCandidates());
41            candidates.remove((Integer) value);
42            final boolean altered = board.removeCandidates(single,
43                candidates);
44            if (altered) {
45              return true;
46            }
47          }
48        }
49      }
50      return false;
51    }
52  }
```

Kod B.12. HiddenPairRule.java

```
1   package se.pellbybrodin.sudokusolver.solvers.human.rules;
2
3   import java.util.ArrayList;
4   import java.util.HashSet;
5   import java.util.List;
6
7   import se.pellbybrodin.sudokusolver.models.Cell;
8   import se.pellbybrodin.sudokusolver.solvers.human.HumanBoard;
9   import se.pellbybrodin.sudokusolver.solvers.human.HumanBoard.Zone;
10  import se.pellbybrodin.sudokusolver.solvers.human.Rule;
11
12  public class HiddenPairRule implements Rule {
13
14    @Override
15    public boolean apply(HumanBoard board) {
16      if (checkZones(board, board.getRows())) {
17        return true;
18      }
19      if (checkZones(board, board.getColumns())) {
```

```
20          return true;
21        }
22        if (checkZones(board, board.getBoxes())) {
23          return true;
24        }
25        return false;
26      }
27
28      private boolean checkZones(HumanBoard board, Zone[] zones) {
29        for (final Zone zone : zones) {
30          if (getPairFromCells(board, zone.getUnsolvedCells())) {
31            return true;
32          }
33        }
34        return false;
35      }
36
37      private boolean getPairFromCells(HumanBoard board, List<Cell> cells)
            {
38        if (cells.size() < 3) {
39          return false;
40        }
41
42        final int[] count = new int[10];
43
44        // Count how many cells each cell is value for
45        for (final Cell cell : cells) {
46          for (final int i : cell.getCandidatesSet()) {
47            count[i]++;
48          }
49        }
50
51        // Each value that is candidate for two cells is a possible pair
              cell
52        final List<Integer> possible = new ArrayList<Integer>();
53        for (int i = 1; i <= 9; i++) {
54          if (count[i] == 2) {
55            possible.add(i);
56          }
57        }
58
59        // For each value, make a pair with another value
60        HashSet<Cell> matches = new HashSet<Cell>();
61        HashSet<Integer> pair;
62        for (int i = 0; i < possible.size(); i++) {
63          for (int j = i + 1; j < possible.size(); j++) {
64            matches = new HashSet<Cell>();
65            pair = new HashSet<Integer>();
66            pair.add(possible.get(i));
67            pair.add(possible.get(j));
68
69            // Add each cell that has both numbers as candidates
70            for (final Cell cell : cells) {
71              final HashSet<Integer> cand = new HashSet<Integer>(
```

```
72                 cell.getCandidatesSet());
73             cand.retainAll(pair);
74             if (cand.size() == 2) {
75               matches.add(cell);
76             } else if (cand.size() == 1) {
77               break;
78             }
79           }
80 
81           // If only two cells contain the pair, a hidden pair is found
82           if (matches.size() == 2) {
83             boolean altered = false;
84             // Remove all other candidates from the cell.
85             for (final Cell cell : matches) {
86               if (board.retainCandidates(cell, pair)) {
87                 altered = true;
88               }
89             }
90             return altered;
91           }
92         }
93       }
94       return false;
95     }
96   }
```

Kod B.13. PointingPair.java

```
1  package se.pellbybrodin.sudokusolver.solvers.human.rules;
2 
3  import java.util.List;
4 
5  import se.pellbybrodin.sudokusolver.models.Cell;
6  import se.pellbybrodin.sudokusolver.solvers.human.HumanBoard;
7  import se.pellbybrodin.sudokusolver.solvers.human.HumanBoard.Zone;
8  import se.pellbybrodin.sudokusolver.solvers.human.Rule;
9 
10 public class PointingPair implements Rule {
11 
12   @Override
13   public boolean apply(HumanBoard board) {
14 
15     List<Integer> potential;
16     // For each box
17     for (final Zone zone : board.getBoxes()) {
18       // For each value
19       for (int i = 1; i <= 9; i++) {
20         potential = zone.potentialPositions(i);
21         // If the value can only be placed in two places in the box
22         if (potential.size() == 2) {
23 
24           final Cell first = zone.getCell(potential.get(0));
25           final Cell second = zone.getCell(potential.get(1));
26           List<Cell> same;
```

```
27            // If they share the same row or column, we have found
28            // pointing pair
29            // in that row/column.
30            if (first.sameRowAs(second)) {
31              same = board.getCellsInRow(first.getRow());
32            } else if (first.sameColumnAs(second)) {
33              same = board.getCellsInColumn(first.getColumn());
34            } else {
35              continue;
36            }
37
38            same.remove(first);
39            same.remove(second);
40            boolean altered = false;
41            // Remove the candidates for the pair for all other cells in
42            // the row / column.
43            for (final Cell cell : same) {
44              if (board.removeCandidate(cell, i)) {
45                altered = true;
46              }
47            }
48            if (altered) {
49              return true;
50            }
51          }
52        }
53      }
54      return false;
55    }
56  }
```