Kungliga Tekniska Högskolan

Examensarbete inom Datalogi,
Grundnivå, DD143X

Att lösa sudoku med SAT-lösare

Författare:
Kristoffer Emanuelsson
kriema@kth.se

Ludvig Stenström
ludste@kth.se

Handledare:
Vahid Mosavat
vahid@nada.kth.se

29 april 2014

Royal Institue of Technology

Degree Project in Computer Science,
First Cycle, DD143X

Solving Sudoku with SAT-solvers

Authors:
Kristoffer Emanuelsson
kriema@kth.se

Ludvig Stenström
ludste@kth.se

Supervisor:
Vahid Mosavat
vahid@nada.kth.se

April 29, 2014

Sammanfattning

I den här rapporten undersöktes hur effektiva och användarvänliga moderna SAT-lösare har blivit, och ifall de lämpar sig för att lösa vardagliga problem. SAT syftar på det första bevisade beräkningssvåra NP-fullständiga problemet satisfierbarhetsproblemet. Två sådana lösare testades genom att det välkända problemet sudoku reducerades till SAT och löstes. Resultaten visar att dessa lösare är ungefär en faktor 15 långsammare än en regelbaserad sudoku-lösare. Jämförelsevis löser en SAT-lösare ett sudokupussel på ca 10 ms. Dessa resultat tyder på att SAT-lösare är tillräckligt snabba för att lösa denna typ av problem, och skulle kunna lösa ännu svårare problem om effektivare reduktioner görs.

Abstract

This report examined how efficient and user friendly modern SAT solvers have become, and if they are suitable for solving everyday problems. SAT refers to the first proven NP-complete problem "the satisfiability problem". The term NP-complete refers to a problem that is hard to compute. Two such solvers were tested by reducing the well-known problem sudoku to SAT and then solving it. The results showed that these solvers are about a factor 15 slower than a rule-based Sudoku solver, but comparatively, a SAT solver solves a Sudoku puzzle in about 10 ms. These results suggest that the SAT solver is fast enough to solve this kind of problem, and could solve even more difficult problems if efficient reductions were made.

Innehåll

Tabeller

Kapitel 1

Inledning

Att schemalägga klasser på en högskola, planera den optimala rutten för en handelsresa, skapa en bordsplacering där gästerna är kräsna med vem de får sitta bredvid eller det klassiska spelet sudoku är alla exempel på NP-fullständiga problem. Denna grupp av problem karaktäriseras av att det ännu inte finns en algoritm som löser dem i polynomisk tid. Ifall en sådan algoritm över huvud taget existerar är idag ett stort forskningsområde. Mycket tid läggs istället på att hitta andra sätt att lösa problemen så fort som möjligt. Satisfierbarhetsproblemet, ofta förkortat SAT, är det första problem som bevisades vara NP-fullständigt, och det finns därför många program som försöker hitta en effektiv lösning på det. Det är därför av intresse att se ifall det är effektivt att ta omvägen via SAT och en SAT-lösare för att lösa ett vardagligt NP-fullständigt problem, såsom sudoku, istället för att försöka finna en lösning direkt. Är det effektivt med avseende på tid, med avseende på hur komplicerat det är att skriva en reduktion till SAT jämfört med en egen lösningsalgoritm, och med avseende på hur lätta SAT-lösarna är att använda?

Resten av det här kapitlet kommer att ta upp problemspecifikationen, syftet med rapporten och lite definitioner om sudoku. I kapitel 2 beskrivs bakgrunden till problemet och uttryck som NP-fullständighet och SAT kommer att förklaras i mer detalj. Kapitel 3 kommer att gå igenom den metod som är använd och kapitel 4 visar upp resultaten. I kapitel 5 diskuteras resultaten och slutsatser dras sedan i kapitel 6. Rapporten avslutas med ett Appendix i kapitel 8 där man kan följa en mer genomgående reduktion i logisk notation. Där finns även skripten och koden som används bifogad.

1.1 Problemspecifikation

Frågeställningen rapporten ska försöka besvara är:

> *Har SAT-lösare blivit så pass snabba och användarvänliga att de kan utnyttjas för att lösa vardagliga NP-fullständiga problem såsom sudoku?*

1.2 Begränsningar

Det finns en mängd SAT-lösare som är mer eller mindre snabba. Att testa alla dessa vore en omöjlighet, därför lades istället fokus på ett fåtal av dessa för att erhålla mer kvalitativa resultat.

I undersökningen valdes två SAT-lösare ut för att genomföra testerna. Den första var Glucose[9] som vunnit flera av kategorierna för 2013 års upplaga av de Internationella SAT-mästerskapen[1]. Som nummer två valdes miniSAT som är en populär och välanvänd SAT-lösare på marknaden, med svenskt ursprung[14].

För att SAT-lösarna skulle fungera bra valdes att bara använda UNIX-baserade system som testplattform, men resultaten bör vara likvärdiga även på andra operativsystem.

Som testdata användes alla de sudokupussel som består av 17 ledtrådar, då dessa är ändliga och skapar en möjlighet till verifikation och jämförelse vid andra undersökningar. Antalet ledtrådar är dessutom svagt korrelerade till svårigheten av sudokuproblemet, men det finns svårare problem med fler ledtrådar som även de skulle kunna vara intressanta att använda som testdata.

1.3 Syfte

Syftet med denna rapport är att undersöka och jämföra hur snabbt det NP-fullständiga problemet sudoku går att lösa genom att ta omvägen att använda sig av SAT och en SAT-lösare.

De senaste åren har mycket fokus lagts på NP-fullständiga problem och att försöka optimera de lösare som existerar när beräkningskapaciteten hos datorer har ökat signifikant. Det är därför av intresse att applicera lösarna på

vardagliga NP-fullständiga problem för att påvisa deras värde och undersöka hur bra de faktiskt är.

1.4 Definitioner

För att kunna diskutera sudoku följer här definitioner för de vanligaste termerna som kommer användas i rapporten.

1.4.1 Ruta

Plats där siffrorna $1 - n$ ska skrivas i pusslet. Kan också benämnas cell.

1.4.2 Region

Ett rutnät bestående av n rutor. För en region gäller samma regler som för en rad eller kolumn, dvs att talen $1 - n$ bara får, och måste, existera exakt en gång.

1.4.3 Kandidat

För varje ruta som är tom i ett sudokupussel finns ett begränsat antal siffror som är möjliga för att fylla rutan utan att bryta mot de regler som finns för rad, kolumn och region, dessa kallas kandidater.

1.4.4 Ledtråd

En ledtråd är en från start redan ifylld ruta som används för att lista ut rätt nummer för resterande rutor. Rutor som fylls i under pusslets gång med hjälp av algoritmer anses inte vara ledtrådar.

Kapitel 2

Bakgrund

Kapitlet kommer att börja med att beskriva NP-fullständighet och fortsätta med att fördjupa läsaren i sudoku, både som ett pussel och som ett matematiskt problem, men också olika sätt att angripa problemet. Avsnitt 2.5 och 2.6 handlar om SAT och SAT-lösare.

2.1 NP-fullständighet

NP-fullständighet syftar till att tala om hur beräkningstungt ett problem är. Det är alltid enkelt att verifiera att en lösning faktiskt är korrekt om en sådan ges, men alla NP-fullständiga problem saknar idag en känd algoritm för att kunna hitta en lösning till problemet i polynomisk tid. Det är dock inte bevisat att en sådan inte existerar. Detta är ett stort, öppet forkningsområde och mycket tid läggs på att ta reda på om en sådan algoritm går eller inte går att finna[4, 7].

Flera kända problem och pussel är NP-fullständiga, och genom att hitta ett snabbt beräkningssätt till ett av dessa, kan man även applicera detta på alla andra NP-fullständiga problem genom reduktion[7]. Några kända NP-fullständiga problem är Handelsresandeproblemet, Kappsäcksproblemet och Sittningsproblemet, men också pussel och spel så som sudoku, schack, MS röj, Go och Mastermind[5, 8].

2.2 Sudoku

Den mest populära varianten av spelet går ut på att fylla ett 9∗9 stort rutnät med siffrorna 1 till och med 9, där varje rad och kolumn enbart får innehålla varje siffra en gång. Dessutom finns 9 stycken $3 * 3$ kvadrater (regioner), som även de endast får innehålla alla siffror högst en gång. I startläget är vissa siffror redan förskrivna i rutorna, och dess antal och placering avgör svårigheten på sudokupusslet.

Sudoku är en ihopskrivning av *suuji wa dokushin ni kagiru*, vilket betyder *numret ska förbli ensamt*. Det hittades i sin pusselform första gången i en New York-baserad pusseltidning från 1979. Det var dock först 1985, då spelet återigen gjorde entré i Japan, och då med dagens namn, som spelet där blev populärt[16]. Det dröjde dock till cirka 2005 innan det slog igenom och blev ett internationellt fenomen, efter att Wayne Gould tog hem en sudokubok från Japan och bestämde sig för att göra ett program för att generera egna sudokupussel[16]. Idag återfinns sudoku som ett vanligt pussel i dagstidningar, såväl som inom tävlingssammanhang. Det finns därför ett intresse för att studera hur dessa pussel kan konstrueras, lösas och analyseras med hjälp av algoritmer som går att implementera med hjälp av datorer.

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

Figur 2.1: Ett sudokupussel med 17 ledtrådar

2.3 Matematiskt problem

År 2005 publicerade Bertram Felgenhauer och Frazer Jarvis en artikel som beräknade antalet möjliga sudokuplaner till

$$6\,670\,903\,752\,021\,072\,936\,960 \approx 6,671 * 10^{21}$$

stycken[6], och vidare beräknades 2006 att antalet signifikant olika sudokuplaner till

$$5\,472\,730\,538 \approx 5,473 * 10^{9}$$

stycken[13]. Att testa alla kombinationer för att undersöka om en lösning fungerar på ett givet pussel, på en modern dator, där antagandet gjorts att 1 miljard lösningar kan verifieras per sekund, skulle ta cirka 190 000 år.

Det är bevisat av Gary McGuire att det inte finns några sudokupussel som endast har 16 ledtrådar med en unik lösning[11]. Hans resultat begränsar antalet ledtrådar till ett minimum av 17, och refererar även till det tidigare kända faktumet att dessa är 49 151 st[12].

Standardsudoku som är uppbyggt av $3 * 3$ st $3 * 3$ kvadrater, eller mer generellt $n^2 * n^2$ rutor är ett NP-svårt kalkyleringsproblem[15]. Detta kan enkelt bevisas genom en reduktion från latinsquares till sudoku[3]. Att bevisa att det dessutom är NP-fullständigt görs enklast genom att verifiera en lösningsinstans i polynomisk tid.

2.4 Sudokualgoritmer

Det finns många olika tillvägagångssätt för att försöka lösa ett sudokupussel. Vissa av dessa metoder och algoritmer lämpar sig bättre för människor att använda, och vissa är det bara realistiskt att en dator försöker sig på. Här följer en kort presentation av två sudokualgortimer.

2.4.1 Brute-force algorithm med Backtracking

Algoritmen börjar med att försöka placera ut talet 1 i första lediga ruta sett från övre vänstra hörnet. Efter att siffran placerats ut undersöks att alla regler håller för hela pusslet. Om detta är fallet, går algoritmen vidare till nästa ruta till höger och upprepar proceduren att placera ut talen från $1 - 9$, och om ett tal kan placeras, går den vidare. I det fall att en regel

bryts försöker den först med nästa tal i ordningen. Den undersöker att alla regler håller. Om detta har gjorts med alla tal upp till och med 9 utan framgång, backar den till föregående ruta och ökar dess värde med 1. Denna procedur upprepas till pusslet är löst. Det går enkelt att inse att algoritmen alltid kommer att ge ett korrekt svar, men det är ingen effektiv algoritm, då logiken för att välja nästa tal inte är särskilt avancerad.

2.4.2 Regelbaserad

En regelbaserad algoritm liknar mer det tillvägagångssätt som människor skulle använda för att lösa en sudoku. De grundläggande reglerna är givna, men till dessa kan fler regler härledas som gör lösningen snabbare. Några vanliga regler som brukar finnas med är[10]:

Ensam Kandidat

Alla siffror har blivit uteslutna från en ruta och det kvarstår endast en, därför kan denna placeras.

Nakna Par/Tripplar

Om rutorna x och y ligger på samma rad, kolumn eller region och har samma kandidater a och b kvar, måste a och b placeras på något sätt i x och y, vilket följer att a och b kan uteslutas från de andra rutorna på samma rad, kolumn eller region. Samma regel kan utökas till tre rutor med tre lika ensamma kandidater, fyra rutor med fyra lika ensamma kandidater, och så vidare.

Gömda Par/Tripplar

Om rutorna x och y ligger på samma rad, kolumn eller region och är de enda rutorna som innehåller kandidaterna a och b, måste a och b placeras på något sätt i x och y, vilket följer att resterande kandidater kan uteslutas från x och y. Samma regel kan utökas till tre rutor som är de enda att innehålla tre specifika kandidater, fyra rutor som är de enda att innehålla fyra specifika kandidater, och så vidare.

2.5 SAT

Satisfieringsproblemet, vidare benämnt SAT, består i att bestämma om det är möjligt att sätta binära litteraler till antingen SANT eller FALSKT så att en serie uttryck valideras till SANT.

2.5.1 Litteraler

Varje litteral, eller variabel, kan förekomma som x eller $\neg x$ (utläses ”inte x”). Litteralen x validerar till SANT om x sätts lika med SANT, och $\neg x$ validerar till SANT om x sätts lika med FALSK.

2.5.2 Klausul

En klausul är en mängd litteraler med $\vee$ (eller) operatorn mellan sig. En klausul validerar till SANT om minst en av litteralerna i klausulen validerar till SANT.

2.5.3 Formel

En formel är en mängd klausuler med $\wedge$ (och) operatorn mellan sig. En formel validerar till SANT om varje klausul validerar till SANT.

2.5.4 Exempel

$\phi = (x_1 \vee x_2) \wedge (\neg x_1 \wedge \neg x_3)$ Formeln består av två klausuler och tre litteraler. Formeln kan satisfieras genom att till exempel sätta x_1=SANT, x_2=SANT och x_3=FALSKT, men den skulle inte bli satisfierad ifall x_3 och x_1 sätts till SANT, eftersom den andra klausulen då kommer att valideras till falskt.

Att lösa SAT-problemet är ett viktigt och välkänt NP-fullständigt problem.

2.6 SAT-lösare

Eftersom SAT är ett viktigt problem inom algoritmteori finns många försök att skapa program som trots allt löser SAT-problemet. Med dagens snabba datorer har dessa blivit allt effektivare och mer lättanvända för gemene man. Det är möjligt att lösa problem av begränsad storlek väldigt fort med dagens teknik. Vartannat år utförs The International SAT Competition, där olika

SAT-lösare tävlar mot varandra i att lösa och verifiera olika SAT-problem så snabbt och så korrekt som möjligt. Vinnaren av denna tävling 2013 var i flera kategorier Glucose. Med anledning av detta fokuserar denna rapport på en tillämpning av lösaren Glucose för sina tester[1, 9].

Kapitel 3

Metod

Det här kapitlet börjar med att beskriva ett sätt att representera en sudoku och fortätter med att beskriva hur reduktionen från sudoku till SAT går till. Avsnitt 3.3 beskriver formatet DIMACS som är en representation av SAT och 3.4 och 3.5 går igenom lösning av SAT respektive utförandet av testen.

3.1 Respresentation av sudoku

Första steget kommer bestå av att konstruera en reduktion av sudokuproblemet till SAT-problemet. Givet indata på form

```
000000081
230000000
040700000
000600370
005300000
100000000
400000200
000058000
000010000
```

där de första 9 siffrorna är rad 1 i pusslet, nästkommande 9 är rad 2 och så vidare, ska en reduktion byggas som översätter denna data till indata för SAT-problemet. En nolla representerar en tom ruta.

3.2 Sudoku till SAT

Varje ruta i ett pussel kan representeras av sin koordinat i x- respektive y-led, c_{11} representerar alltså den översta vänstra rutan. I SAT kan varje litteral endast ha två värden, SANT eller FALSKT. För att kunna representera vilket nummer som ska vara i varje enskild ruta, representeras varje ruta av 9 stycken litteraler s_{xyz} där $z \in \{1..9\}$. Om litteralen s_{xyz} är sann för något $x, y, z \in \{1..9\}$ innebär det att nummer z ska stå på ruta c_{xy}. För ett löst sudoku ska endast en av s_{xy1} till s_{xy9} vara sann samtidigt för varje $x, y \in \{1..9\}$.

Reglerna kan då definieras genom fyra omskrivningar (reduktionen som logisk notation går att hitta i Appendix):

3.2.1 Minst ett nummer i varje ruta

Någon av s_{xy1} till s_{xy9} måste vara sant för varje $x, y \in \{1..9\}$

Matematisk notation

$$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigvee_{z=1}^{9} s_{xyz} \tag{3.1}$$

3.2.2 Varje nummer existerar max en gång per rad

Om s_{xyz} är sant för någon $x, y, z \in \{1..9\}$ får ingen annan av s_{iyz} vara sann för något $i \in \{1..9\}, i \neq x$

Matematisk notation

$$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{x=1}^{8} \bigwedge_{i=x+1}^{9} (\neg s_{xyz} \vee \neg s_{iyz}) \tag{3.2}$$

3.2.3 Varje nummer existerar max en gång per kolumn

Om s_{xyz} är sant för någon $x, y, z \in \{1..9\}$ får ingen annan av s_{xiz} vara sann för något $i \in \{1..9\}, i \neq y$

Matematisk notation

$$\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{y=1}^{8} \bigwedge_{i=y+1}^{9} (\neg s_{xyz} \vee \neg s_{xiz}) \tag{3.3}$$

3.2.4 Varje nummer existerar max en gång per region

Om s_{xyz} är sant för någon $x, y, z \in \{1..9\}$ får ingen annan av s_{ijz} vara sann för något $i, j \in \{$ samma region $\}$

Matematisk notation

$$\bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{2}\bigwedge_{k=y+1}^{3}(\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+x)(3j+k)z}) \tag{3.4}$$

$$\bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{2}\bigwedge_{y=1}^{3}\bigwedge_{k=x+1}^{3}\bigwedge_{l=1}^{3}(\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+k)(3j+l)z}) \tag{3.5}$$

3.2.5 Ledtrådar

Ledtrådarna är de givna siffrorna i ett sudokupussel. De säger att på c_{xy} ska siffra z stå. Denna kunskap kan enkelt läggas till i form av klausuler med endast en litteral i.

Om z är en ledtråd som ska stå på c_{xy} för något $x, y, z \in \{1..9\}$ läggs denna klausul till:

$$(s_{xyz}) \tag{3.6}$$

3.2.6 Konstant reduktion

Reduktionen när det gäller reglerna är alltid densamma för alla pussel, oavsett vilka ledtrådar pusslet har, vilket gör att denna bit av reduktionen är konstant. Kvar är sedan bara ledtrådarna som går i linjär tid att reducera. Detta gör att reduktionen kan göras väldigt snabbt och mycket kan förberäknas.

3.3 DIMACS

Den färdiga reduktionen ska presentera indata till SAT-lösaren på DIMACS-formatet[2]. Exempel på DIMACS:

```
c
c start with comments
c
```

```
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

Filen har följande utseende:

- Filen kan starta med kommentarer. Dessa indikeras med ett `c` som första tecken på raden
- Direkt efter kommentarerna följer raden `p cnf nbvar nbclauses`
  - `nbvar` totalt antal variabler
  - `nbclauses` totalt antal klausuler
- Efter detta kommer `nbclauses` st rader
  - Varje rad består av en mellanrum-separerad lista med tal $x \in \{-\texttt{nbclauses} \ldots -1, 1 \ldots \texttt{nbclauses}\}$. Där
    - litteral $x > 0$ är SANT om x är SANT
    - litteral $x < 0$ är SANT om $|x|$ är FALSK

3.4 Lösning av SAT

För att kunna besvara frågeställningen kommer två delar att behöva testas. Steg ett är att ta reda på hur lätt det är att installera och använda en SAT-lösare. Två olika SAT-lösare har därför valts ut och kommer att installeras för att sedan testköras. Steg två är att låta lösarna försöka lösa instanser av SAT. Lösarnas prestanda mäts i första hand genom hur många av probleminstanserna lösarna klarade av att lösa. Om de presterade lika mäts genomsnittstiden det tog för dem att lösa samtliga problem.

3.4.1 Installation av SAT-lösare

Både Glucose och miniSAT är skrivna i C och distribueras okompilerade. Det innebär att för att kunna använda sig av dessa SAT-lösare krävs en C-kompilator på sin maskin, samt ett specifikt kodbibliotek installerat. Detta står dock beskrivet på Glucose installationssida. Installation testades på operativsystemen Windows 7 64-bit, Mac OSX 64-bit samt Ubuntu 32/64-bit.

På både Mac och Windows uppstod problem med det nödvändiga tredjepartsbiblioteket, men på Ubuntu gick detta utan problem och installationen tog fem minuter från nedladdning till körning av första testet. Det steg som troligen är svårast för användare utan förkunskaper inom skapandet av egna program är hur ett program kompileras och att det därefter kräver att en kommandotolk används. Dock är de steg som behövs relativt enkla att utföra om man tar hjälp av det stora utbud av guider som finns tillgängligt online. Dessutom är kompilering något som endast behöver utföras en gång.

3.4.2 Utförande av test

Testet delas upp i två delar på grund av testdatas storlek samt för att se om resultaten varierar mellan olika system. Hårdvaran för system finns specificerad i Tabell 3.1. Resultatet fås genom att sex separata körningar av ett bash-script körs på två olika Unixsystem. Det första resultatet kastas då det kan variera mer innan programmet har optimerats med avseende på hårdvaran. För båda lösarna mäts två saker: antalet lösta sudokus, samt genomsnittstiden för en lösning.

	System 1	**System 2**
OS	Ubuntu 12.04.4 LTS - x86_64	Ubuntu 12.04.4 LTS - x86_32
CPU	Intel Xeon(R) X3470 @ 2.93GHz	Intel i5-4670K @ 3.40GHz
RAM	15 GiB	4096 MiB

Tabell 3.1: Hårdvara på testsytemet

Testdata

Testdata till lösarna är de till idag kända sudokupusslen med 17 ledtrådar[12]. Detta gav lösarna nästan 50 000 pussel att lösa.
För System 1 körs bara en del av testfilerna på grund av platsbrist.

3.5 Regelbaserad lösare

För att jämföra hur väl de valda SAT-lösarna presterar körs även tester på en regelbaserad sudokulösare, som då löser sudoku direkt. Lösaren valdes på grund av att den liksom SAT-lösarna är skriven i C, samt att den använder sig

av regler för att lösa pusslet och inte enbart testar alla möjliga lösningar[17]. De regler som använts är bland annat det tre beskrivna i avsnitt 2.4.2.

Kapitel 4

Resultat

Här presenteras resultaten från de gjorda testerna i separata tabeller för Glucose, miniSAT och den regelbaserade lösaren som använts som jämförelse. Kapitlet avslutas med en sammanställning av de olika tabellerna.

4.1 Glucose

4.1.1 System 1

Nr	**Antal filer**	**Antal lösta**	**Genomsnittlig tid (ms)**
Test 1	2000	2000	8,039
Test 2	2000	2000	8,132
Test 3	2000	2000	8,064
Test 4	2000	2000	8,064
Test 5	2000	2000	8,097
Total genomsnittlig tid:		8,079 ms	

Tabell 4.1: Resultat av Glucose efter test på System 1

4.1.2 System 2

Nr	Antal filer	Antal lösta	Genomsnittlig tid (ms)
Test 1	49151	49151	19,158
Test 2	49151	49151	19,106
Test 3	49151	49151	19,356
Test 4	49151	49151	19,241
Test 5	49151	49151	19,394
Total genomsnittlig tid:		19,251 ms	

Tabell 4.2: Resultat av Glucose efter test på System 2

4.2 MiniSAT

4.2.1 System 1

Nr	Antal filer	Antal lösta	Genomsnittlig tid (ms)
Test 1	2000	2000	7,846
Test 2	2000	2000	7,803
Test 3	2000	2000	7,846
Test 4	2000	2000	7,793
Test 5	2000	2000	7,848
Total genomsnittlig tid:		7,827 ms	

Tabell 4.3: Resultat av miniSAT efter test på System 1

4.2.2 System 2

Nr	Antal filer	Antal lösta	Genomsnittlig tid (ms)
Test 1	49151	49151	20,005
Test 2	49151	49151	20,201
Test 3	49151	49151	19,842
Test 4	49151	49151	20,138
Test 5	49151	49151	19,986
Total genomsnittlig tid:		20,034 ms	

Tabell 4.4: Resultat av miniSAT efter test på System 2

4.3 Regelbaserad lösare

4.3.1 System 1

Nr	Antal filer	Antal lösta	Genomsnittlig tid (ms)
Test 1	2000	2000	0,390
Test 2	2000	2000	0,388
Test 3	2000	2000	0,387
Test 4	2000	2000	0,386
Test 5	2000	2000	0,387
Total genomsnittlig tid:		0,388 ms	

Tabell 4.5: Resultat av regelbaserad lösare efter test på System 1

4.3.2 System 2

Nr	Antal filer	Antal lösta	Genomsnittlig tid (ms)
Test 1	49151	49151	0,161
Test 2	49151	49151	0,161
Test 3	49151	49151	0,161
Test 4	49151	49151	0,161
Test 5	49151	49151	0,161
Total genomsnittlig tid:		0,161 ms	

Tabell 4.6: Resultat av regelbaserad lösare efter test på System 2

4.4 Sammanställning

System	Glucose (ms)	miniSAT (ms)	Regelbaserad (ms)
1	8,079	7,827	0,388
2	19,251	20,034	0,161

Tabell 4.7: Sammanställning av testresultat

Kapitel 5

Diskussion

Resultaten visar att miniSAT gör ett bättre jobb med att lösa sudokuproblemet på system 1, och Glucose gör det snabbare på system 2. Resultat är intressant då Glucose bygger på miniSAT och därför förväntades vara snabbare. Vidare går det tydligt att avläsa att den regelbaserade sudokulösaren som testats är överlägset snabbare än SAT-lösarna. Detta kan bero på flera saker:

Reduktionen som genomförts i den här rapporten är en minimalistisk sådan, vilket gör att SAT-lösarna blir tvungna att börja om många gånger. Detta skulle kunna förbättras genom att lägga till fler redundanta regler som tvingar ner frihetsgraderna på variablerna något.

En annan tanke är att det här är ett väldigt specifikt problem som löses på ett väldigt generellt vis. Ett alternativ skulle vara att skapa en lösare som kombinerar en regelbaserad lösare och en SAT-lösare. Den regelbaserade lösaren skulle vara enkel och bara implementera ett fåtal regler, och när dessa inte kunde fylla fler rutor, skulle SAT-lösaren ta vid för att lösa resten av problemet. Detta skulle markant minska frihetsgraderna hos SAT-problemet och lösningen skulle troligen gå snabbare.

Utläsas kan också det faktum att SAT-lösarna klarade av att lösa 100 % av problemen, vilket ger en positiv tanke om att SAT är en möjlig fungerande väg att gå för att lösa vardagliga problem såsom sudoku. Reduktionen är dessutom väldigt enkel att ta till sig och göra, och resultatet från lösarna är även sedan lätt att reducera tillbaka, vilket gör tillvägagångssättet föredömligt i det här fallet.

Det var slutligen överraskande hur lätt det var att komma igång med

att använda de SAT-lösarna som testades, vilket öppnar upp för funderingar kring andra möjligheter att utnyttja dessa till vardagsproblem.

Det är dock möjligt att vissa vardagsproblem skulle kunna vara mycket svårare att reducera till SAT än vad sudoku har varit. Det skulle göra arbetet väldigt komplicerat om man vill använda en SAT-lösare vid dessa tillfällen, men om reduktionen är enkel och intuitiv kan detta vara ett bra tillvägagångssätt.

Kapitel 6

Slutsatser

I denna rapport har undersökts huruvida SAT-lösare har utvecklats i både användarvänlighet och effektivitet tillräckligt för vara intressanta att använda för att lösa vardagsproblem. Efter att ha studerat resultatet anses så vara fallet. De må inte alltid vara snabbast, men det var förhållandevis enkelt att översätta det vanliga problemet sudoku till ett SAT-problem och lösa det med hjälp av en SAT-lösare.

En fördel med att använda SAT-lösare istället för att implementera en egen algoritm för ett givet problem skulle kunna vara att reduktionen alltid kommer vara densamma, och förbättras i takt med att bättre lösare tas fram.

En naturlig fortsättning för projektet skulle vara att utvidga reduktionen av SAT för att troligen erhålla snabbare tider. Om SAT-lösarna hade kunnat närma sig tiden för en regelbaserad sudokulösare skulle intresset att gå över till att använda SAT-lösare för sina vardagliga problem antagligen öka ännu mer.

Kapitel 7

Litteraturförteckning

[1] *The international SAT Competitions web page*. Anton Belov, m.fl. Hämtad den 11 april 2014, från http://www.satcompetition.org/

[2] *SAT Competition 2009: Benchmark Submission Guidelines*. Anton Belov, m.fl. Hämtad den 11 april 2014, från http://www.satcompetition.org/2009/format-benchmarks2009.html

[3] *The complexity of completing partial Latin squares*. Charles J. Colbourn. Publicerad april 1984. Hämtad från http://www.sciencedirect.com/science/article/pii/0166218X84900751

[4] *The Complexity of Theorem-Proving Procedures*. Stephen A. Cook. Hämtad den 25 april 2014, från http://dl.acm.org/citation.cfm?id=805047

[5] *Computational Complexity of Games and Puzzles*. David Eppstein. Hämtad den 25 april 2014, från http://www.ics.uci.edu/~eppstein/cgt/hard.html

[6] *Enumerating possible Sudoku grids*. Frazer Jarvis Publicerad 20 juni 2005. Hämtad från http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf

[7] *Reducibility among combinatorial problems*. Richard M. Karp. Hämtad den 25 april 2014, från http://cgi.di.uoa.gr/~sgk/teaching/grad/handouts/karp.pdf

[8] *A Survey of NP-complete puzzles*. Graham Kendall, m.fl. Hämtad den 25 april 2014, från https://www.cs.wmich.edu/~elise/courses/cs431/icga2008.pdf

[9] *The Glucose SAT Solver*. Prof. Don Knuth. Hämtad den 11 april 2014, från http://www.labri.fr/perso/lsimon/glucose/

[10] *A Guess-Free Sudoku Solver*. Glen Mailer. Publicerad 6 maj 2008. Hämtad från http://www.dcs.shef.ac.uk/intranet/teaching/public/projects/archive/ug2008/pdf/aca05gam.pdf

[11] *There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem 2012*. Gary McGuire. Hämtad den 25 februari 2014, från http://www.math.ie/McGuire_V1.pdf

[12] *Minimum Sudoku*. Gordon Royle. Hämtad den 14 april 2014, från http://school.maths.uwa.edu.au/~gordon/sudokumin.php

[13] *Mathematics of Sudoku II*. Ed Russell and Frazer Jarvis Publicerad 25 januari 2006. Hämtad från http://www.afjarvis.staff.shef.ac.uk/sudoku/russell_jarvis_spec2.pdf

[14] *The minisat page*. Niklas Eén, Niklas Sörensson. Hämtad den 11 april 2014, från http://minisat.se/

[15] *Complexity and Completeness of Finding Another Solution and Its Application to Puzzles*. Takayuki Yato and Takahiro Seta. Publicerad 2003. Hämtad från http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL87-2.pdf

[16] *Sudoku 2014. Encyclopædia Britannica Online*. Hämtad den 25 februari 2014, från http://www.britannica.com/EBchecked/topic/1214841/sudoku

[17] *A Su Doku Solver in C*. Hämtad den 28 april 2014, från http://www.techfinesse.com/game/sudoku_solver.php

Kapitel 8

Appendix

8.1 Logisk reduktion

8.1.1 Minst ett nummer i varje ruta

Logisk notation

$$
\begin{aligned}
&(s_{111} \vee s_{112} \vee s_{113} \vee ... \vee s_{118} \vee s_{119}) \wedge \\
&(s_{121} \vee s_{122} \vee s_{123} \vee ... \vee s_{128} \vee s_{129}) \wedge \\
&\quad \vdots \\
&(s_{191} \vee s_{192} \vee s_{193} \vee ... \vee s_{198} \vee s_{199}) \wedge \\
&(s_{211} \vee s_{212} \vee s_{213} \vee ... \vee s_{218} \vee s_{219}) \wedge \\
&\quad \vdots \\
&(s_{991} \vee s_{992} \vee s_{993} \vee ... \vee s_{998} \vee s_{999})
\end{aligned}
$$

8.1.2 Varje nummer existerar max en gång per rad

Logisk notation

$$
\begin{aligned}
&(\neg s_{111} \vee \neg s_{211}) \wedge (\neg s_{111} \vee \neg s_{311}) \wedge ... \wedge (\neg s_{111} \vee \neg s_{911}) \wedge \\
&(\neg s_{211} \vee \neg s_{311}) \wedge (\neg s_{211} \vee \neg s_{411}) \wedge ... \wedge (\neg s_{211} \vee \neg s_{911}) \wedge \\
&\quad \vdots \\
&(\neg s_{811} \vee \neg s_{911}) \wedge \\
&(\neg s_{121} \vee \neg s_{221}) \wedge (\neg s_{121} \vee \neg s_{321}) \wedge ... \wedge (\neg s_{121} \vee \neg s_{921}) \wedge \\
&\quad \vdots \\
&(\neg s_{898} \vee \neg s_{998}) \wedge \\
&(\neg s_{199} \vee \neg s_{299}) \wedge (\neg s_{199} \vee \neg s_{399}) \wedge ... \wedge (\neg s_{199} \vee \neg s_{999}) \wedge
\end{aligned}
$$

$$\vdots$$
$$(\neg s_{899} \vee \neg s_{999})$$

8.1.3 Varje nummer existerar max en gång per kolumn

Logisk notation

$$(\neg s_{111} \vee \neg s_{121}) \wedge (\neg s_{111} \vee \neg s_{121}) \wedge ... \wedge (\neg s_{111} \vee \neg s_{191}) \wedge$$
$$(\neg s_{121} \vee \neg s_{131}) \wedge (\neg s_{121} \vee \neg s_{141}) \wedge ... \wedge (\neg s_{121} \vee \neg s_{191}) \wedge$$
$$\vdots$$
$$(\neg s_{181} \vee \neg s_{191}) \wedge$$
$$(\neg s_{211} \vee \neg s_{221}) \wedge (\neg s_{211} \vee \neg s_{231}) \wedge ... \wedge (\neg s_{211} \vee \neg s_{291}) \wedge$$
$$\vdots$$
$$(\neg s_{988} \vee \neg s_{998}) \wedge$$
$$(\neg s_{919} \vee \neg s_{929}) \wedge (\neg s_{919} \vee \neg s_{939}) \wedge ... \wedge (\neg s_{919} \vee \neg s_{999}) \wedge$$
$$\vdots$$
$$(\neg s_{989} \vee \neg s_{999})$$

8.1.4 Varje nummer existerar max en gång per region

Logisk notation

$$(\neg s_{111} \vee \neg s_{121}) \wedge (\neg s_{111} \vee \neg s_{131}) \wedge (\neg s_{121} \vee \neg s_{131}) \wedge$$
$$(\neg s_{211} \vee \neg s_{221}) \wedge (\neg s_{211} \vee \neg s_{231}) \wedge (\neg s_{221} \vee \neg s_{231}) \wedge$$
$$(\neg s_{311} \vee \neg s_{321}) \wedge (\neg s_{311} \vee \neg s_{331}) \wedge (\neg s_{321} \vee \neg s_{331}) \wedge$$
$$(\neg s_{141} \vee \neg s_{151}) \wedge (\neg s_{141} \vee \neg s_{161}) \wedge (\neg s_{151} \vee \neg s_{161}) \wedge$$
$$\vdots$$
$$(\neg s_{171} \vee \neg s_{181}) \wedge (\neg s_{171} \vee \neg s_{191}) \wedge (\neg s_{181} \vee \neg s_{191}) \wedge$$
$$(\neg s_{411} \vee \neg s_{421}) \wedge (\neg s_{411} \vee \neg s_{431}) \wedge (\neg s_{421} \vee \neg s_{431}) \wedge$$
$$\vdots$$
$$(\neg s_{971} \vee \neg s_{981}) \wedge (\neg s_{971} \vee \neg s_{991}) \wedge (\neg s_{981} \vee \neg s_{991}) \wedge$$
$$(\neg s_{112} \vee \neg s_{122}) \wedge (\neg s_{112} \vee \neg s_{132}) \wedge (\neg s_{122} \vee \neg s_{132}) \wedge$$
$$\vdots$$
$$(\neg s_{979} \vee \neg s_{989}) \wedge (\neg s_{979} \vee \neg s_{999}) \wedge (\neg s_{989} \vee \neg s_{999}) \wedge$$

$$(\neg s_{111} \vee \neg s_{211}) \wedge (\neg s_{111} \vee \neg s_{221}) \wedge (\neg s_{111} \vee \neg s_{231}) \wedge$$
$$(\neg s_{111} \vee \neg s_{311}) \wedge (\neg s_{111} \vee \neg s_{321}) \wedge (\neg s_{111} \vee \neg s_{331}) \wedge$$

$$
\begin{aligned}
&(\neg s_{121} \vee \neg s_{211}) \wedge (\neg s_{121} \vee \neg s_{221}) \wedge (\neg s_{121} \vee \neg s_{231}) \wedge \\
&\quad \vdots \\
&(\neg s_{131} \vee \neg s_{311}) \wedge (\neg s_{131} \vee \neg s_{321}) \wedge (\neg s_{131} \vee \neg s_{331}) \wedge \\
&(\neg s_{211} \vee \neg s_{311}) \wedge (\neg s_{211} \vee \neg s_{321}) \wedge (\neg s_{211} \vee \neg s_{331}) \wedge \\
&(\neg s_{221} \vee \neg s_{311}) \wedge (\neg s_{221} \vee \neg s_{321}) \wedge (\neg s_{221} \vee \neg s_{331}) \wedge \\
&(\neg s_{141} \vee \neg s_{241}) \wedge (\neg s_{141} \vee \neg s_{251}) \wedge (\neg s_{141} \vee \neg s_{261}) \wedge \\
&(\neg s_{141} \vee \neg s_{341}) \wedge (\neg s_{141} \vee \neg s_{351}) \wedge (\neg s_{141} \vee \neg s_{361}) \wedge \\
&(\neg s_{151} \vee \neg s_{241}) \wedge (\neg s_{151} \vee \neg s_{251}) \wedge (\neg s_{151} \vee \neg s_{261}) \wedge \\
&\quad \vdots \\
&(\neg s_{291} \vee \neg s_{371}) \wedge (\neg s_{291} \vee \neg s_{381}) \wedge (\neg s_{291} \vee \neg s_{391}) \wedge \\
&(\neg s_{411} \vee \neg s_{511}) \wedge (\neg s_{411} \vee \neg s_{521}) \wedge (\neg s_{411} \vee \neg s_{531}) \wedge \\
&\quad \vdots \\
&(\neg s_{891} \vee \neg s_{971}) \wedge (\neg s_{891} \vee \neg s_{981}) \wedge (\neg s_{891} \vee \neg s_{991}) \wedge \\
&(\neg s_{112} \vee \neg s_{212}) \wedge (\neg s_{112} \vee \neg s_{222}) \wedge (\neg s_{112} \vee \neg s_{232}) \wedge \\
&\quad \vdots \\
&(\neg s_{899} \vee \neg s_{979}) \wedge (\neg s_{899} \vee \neg s_{989}) \wedge (\neg s_{899} \vee \neg s_{999})
\end{aligned}
$$

8.2 Glucose script

solveSudokuGlucose.sh

```
1  #!/bin/bash
2  #Setup logfile
3
4  LOGFILE="./Logs/glucose.log"
5  if [ -e $LOGFILE ]
6  then
7      echo "" > $LOGFILE
8  else
9      touch $LOGFILE
10 fi
11
12 #Setup variables
13 NUM_FILES=‘ls -l ./Problems/ | grep txt | wc -l‘
14 NUM_SOLVED=0
15 TOTAL_TIME=0
```

```
16 LOOP_COUNTER=1
17 #   LOOP over all problem files and solve them
18 for file in $(ls ./Problems/); do
19 # UNIX timestamp concatenated with nanoseconds
20 T=$(date +%s%N)
21 #Solve the Sudoku
22 ./glucose-3.0/core/glucose  ./Problems/$file
      ./Solutions/sg$file > /dev/null
23 #Messure the time
24 TIME_TO_SOLVE=$[ $(date +%s%N) - $T ]
25 TOTAL_TIME="$(($TOTAL_TIME + $TIME_TO_SOLVE))"
26
27 clear
28 echo " $LOOP_COUNTER / $NUM_FILES problems processed"
29 LOOP_COUNTER=$(($LOOP_COUNTER +1))
30
31 #Save data to log file
32 SOLVED=""
33 SOLUTION=""
34 i=1
35 while read line; do
36     if [ $i == 1 ]; then
37         SOLVED=$line
38         if [ "$line" == "SAT" ]; then
39             NUM_SOLVED=$(($NUM_SOLVED + 1))
40         fi
41     else
42     SOLUTION=$line
43     fi
44     ((i++))
45 done < ./Solutions/sg$file
46 #Write data to log about current solution
47 printf "File: $file Time: $TIME_TO_SOLVE Solved:
      $SOLVED Solution: $SOLUTION \n" >>
      ./Logs/glucose.log
```

```
48 printf "Num files: $NUM_FILES Num solved:
       $NUM_SOLVED \n" >> ./Logs/glucose.log
49 done
50 # All sudokus solved, write summary data to logfile
51 # Microseconds Total
52 Micro="$(($TOTAL_TIME/1000))"
53 # Milliseconds Total
54 Milli="$(($TOTAL_TIME/1000000))"
55 # Seconds total
56 Secs="$(($TOTAL_TIME/1000000000))"
57 TOTAL_TIME_FORMATED="Total time: ${Milli}.$((Micro %
       1000)) $((TOTAL_TIME % 1000)) ms "
58 printf "$TOTAL_TIME_FORMATED \n" >>
       ./Logs/glucose.log
59 AVG_TIME=$(($TOTAL_TIME / $NUM_FILES))
60 # Microseconds AVG
61 Micro="$(($AVG_TIME/1000))"
62 # Milliseconds AVG
63 Milli="$(($AVG_TIME/1000000))"
64 # Seconds AVG
65 Secs="$(($AVG_TIME/1000000000))"
66 AVG_TIME_FORMATED="Average time: ${Milli}.$((Micro %
       1000)) $((AVG_TIME % 1000)) ms"
67 printf "$AVG_TIME_FORMATED \n" >> ./Logs/glucose.log
68 echo $TOTAL_TIME_FORMATED
69 echo $AVG_TIME_FORMATED
```

8.3 miniSAT script

solveSudokuMini.sh

```
1 #!/bin/bash
2 #Setup logfile
3
4 LOGFILE="./Logs/mini.log"
5 if [ -e $LOGFILE ]
```

```
6  then
7      echo "" > $LOGFILE
8  else
9      touch $LOGFILE
10 fi
11
12 #Setup variables
13 NUM_FILES=‘ls -l ./Problems/ | grep txt | wc -l‘
14 NUM_SOLVED=0
15 TOTAL_TIME=0
16 LOOP_COUNTER=1
17 #   LOOP over all problem files and solve them
18 for file in $(ls ./Problems/); do
19 # UNIX timestamp concatenated with nanoseconds
20 T=$(date +%s%N)
21 #Solve the Sudoku
22 ./minisat/core/minisat_release  ./Problems/$file
       ./Solutions/sm$file  > /dev/null
23 #Messure the time
24 TIME_TO_SOLVE=$[ $(date +%s%N) - $T ]
25 TOTAL_TIME="$(($TOTAL_TIME + $TIME_TO_SOLVE))"
26
27 clear
28 echo " $LOOP_COUNTER / $NUM_FILES problems processed"
29 LOOP_COUNTER=$(($LOOP_COUNTER +1))
30
31 #Save data to log file
32 SOLVED=""
33 SOLUTION=""
34 i=1
35 while read line; do
36     if [ $i == 1 ]; then
37         SOLVED=$line
38         if [ "$line" == "SAT" ]; then
39             NUM_SOLVED=$(($NUM_SOLVED + 1))
40         fi
```

```
41      else
42      SOLUTION=$line
43      fi
44      ((i++))
45  done < ./Solutions/sg$file
46  #Write data to log about current solution
47  printf "File: $file Time: $TIME_TO_SOLVE Solved:
        $SOLVED Solution: $SOLUTION \n" >> $LOGFILE
48  printf "Num files: $NUM_FILES Num solved:
        $NUM_SOLVED \n" >> $LOGFILE
49  done
50  # All sudokus solved, write summary data to logfile
51  # Microseconds Total
52  Micro="$(($TOTAL_TIME/1000))"
53  # Milliseconds Total
54  Milli="$(($TOTAL_TIME/1000000))"
55  # Seconds total
56  Secs="$(($TOTAL_TIME/1000000000))"
57  TOTAL_TIME_FORMATED="Total time: ${Milli}.$((Micro %
        1000)) $((TOTAL_TIME % 1000)) ms "
58  printf "$TOTAL_TIME_FORMATED \n" >> $LOGFILE
59  AVG_TIME=$(($TOTAL_TIME / $NUM_FILES))
60  # Microseconds AVG
61  Micro="$(($AVG_TIME/1000))"
62  # Milliseconds AVG
63  Milli="$(($AVG_TIME/1000000))"
64  # Seconds AVG
65  Secs="$(($AVG_TIME/1000000000))"
66  AVG_TIME_FORMATED="Average time: ${Milli}.$((Micro %
        1000)) $((AVG_TIME % 1000)) ms"
67  printf "$AVG_TIME_FORMATED \n" >> $LOGFILE
68  echo $TOTAL_TIME_FORMATED
69  echo $AVG_TIME_FORMATED
```

8.4 Regelbaserad solver script

solveSudokuNotSat.sh

```
#!/bin/bash
#Setup logfile

LOGFILE="./Logs/regulatSudokuSolver.log"
if [ -e $LOGFILE ]
then
    echo "" > $LOGFILE
else
    touch $LOGFILE
fi

#Setup variables
NUM_FILES=49151
NUM_SOLVED=0
TOTAL_TIME=0
file="sudoku17-ml"
# UNIX timestamp concatenated with nanoseconds
T=$(date +%s%N)
#Solve the Sudoku
./solver_1.20/a.out -f ./$file
#Messure the time
TIME_TO_SOLVE=$[ $(date +%s%N) - $T ]
#Write data to log about current solution
# Microseconds Total
Micro="$(($TIME_TO-SOLVE/1000))"
# Milliseconds Total
Milli="$(($TIME_TO_SOLVE/1000000))"
# Seconds total
Secs="$(($TIME_TO_SOLVE/1000000000))"
TOTAL_TIME_FORMATED="Total time: ${Milli}.$((Micro %
    1000)) $((TIME_TO_SOLVE % 1000)) ms "
printf "$TOTAL_TIME_FORMATED \n"
AVG_TIME=$(($TIME_TO_SOLVE / $NUM_FILES))
# Microseconds AVG
```

```
34 Micro="$(($AVG_TIME/1000))"
35 # Milliseconds AVG
36 Milli="$(($AVG_TIME/1000000))"
37 # Seconds AVG
38 Secs="$(($AVG_TIME/1000000000))"
39 AVG_TIME_FORMATED="Average time: ${Milli}.$((Micro %
       1000)) $((AVG_TIME % 1000)) ms"
40 printf "$AVG_TIME_FORMATED \n"
```

8.5 Reduktion från sudoku till SAT

reduceSudokuToSAT.cpp

```
1  /*
2   * File:   main.cpp
3   * Author: Krycke
4   *
5   * Created on April 2, 2014, 8:44 AM
6   */
7
8  #include <cstdlib>
9  #include <iostream>
10 #include <fstream>
11 #include <cstring>
12 #include <string>
13 #include <sstream>
14
15 #define N 8829
16
17 using namespace std;
18
19 ifstream fin("test.in");
20 #ifdef LOCAL
21 #define cin fin
22 #endif
23
```

```
24 int getPos(int x, int y) {
25     return (x - 1)*9 + (y - 1);
26 }
27
28 /*
29  *
30  */
31 int main(int argc, char** argv) {
32     ios::sync_with_stdio(false);
33     cin.tie(NULL);
34
35     char clues[82];
36     int file_num = 1;
37     stringstream base;
38
39     //Ett nummer i varje ruta
40     for (int x = 1; x <= 9; x++) {
41         for (int y = 1; y <= 9; y++) {
42             for (int z = 1; z <= 9; z++) {
43                 base << x << y << z << " ";
44             }
45             base << " 0\n";
46         }
47     }
48
49     //Max en per rad
50     for (int y = 1; y <= 9; y++) {
51         for (int z = 1; z <= 9; z++) {
52             for (int x = 1; x <= 9; x++) {
53                 for (int i = x + 1; i <= 9; i++) {
54                     base << "-" << x << y << z << "
                           -" << i << y << z << " 0\n";
55                 }
56             }
57         }
58     }
```

```
59
60      //Max en per kolumn
61      for (int x = 1; x <= 9; x++) {
62          for (int z = 1; z <= 9; z++) {
63              for (int y = 1; y <= 9; y++) {
64                  for (int i = y + 1; i <= 9; i++) {
65                      base << "-" << x << y << z << "
                            -" << x << i << z << " 0\n";
66                  }
67              }
68          }
69      }
70
71      //Max en per region
72      for (int z = 1; z <= 9; z++) {
73          for (int i = 0; i <= 2; i++) {
74              for (int j = 0; j <= 2; j++) {
75                  for (int x = 1; x <= 3; x++) {
76                      for (int y = 1; y <= 2; y++) {
77                          for (int k = y + 1; k <= 3;
                                k++) {
78                              base << "-" << 3 * i + x
                                    << 3 * j + y << z <<
                                    " -" << 3 * i + x <<
                                    3 * j + k << z << "
                                    0\n";
79                          }
80                      }
81                  }
82              }
83          }
84      }
85      for (int z = 1; z <= 9; z++) {
86          for (int i = 0; i <= 2; i++) {
87              for (int j = 0; j <= 2; j++) {
88                  for (int x = 1; x <= 2; x++) {
```

```
89                      for (int y = 1; y <= 3; y++) {
90                          for (int k = x + 1; k <= 3;
                               k++) {
91                              for (int l = 1; l <= 3;
                                   l++) {
92                                  base << "-" << 3 * i
                                       + x << 3 * j + y
                                       << z << " -" << 3
                                       * i + k << 3 * j
                                       + l << z << "
                                       0\n";
93                              }
94                          }
95                      }
96                  }
97              }
98          }
99      }
100
101     while (cin.getline(clues, 82) > 0) {
102         stringstream ss;
103         ss << "dimacs/sudoku" << file_num++ <<
               ".txt";
104         //          string concstr = ss.str();
105         //          ss.str("");
106         ofstream os(ss.str());
107         int number_of_clausuls = N;
108         for (int x = 1; x <= 9; x++) {
109             for (int y = 1; y <= 9; y++) {
110                 if (clues[getPos(x, y)] != '0') {
111                     number_of_clausuls++;
112                 }
113             }
114         }
115         os << "p cnf " << 999 << " " <<
               number_of_clausuls << "\n";
```

```
116         for (int x = 1; x <= 9; x++) {
117             for (int y = 1; y <= 9; y++) {
118                 if (clues[getPos(x, y)] != '0') {
119                     os << x << y << clues[getPos(x,
                           y)] << " 0\n";
120                 }
121             }
122         }
123
124
125
126         os << base.str();
127
128         os.close();
129     }
130
131
132
133     return 0;
134 }
```

8.6 SAT till sudoku

SATtoSudoku.cpp

```
1  /*
2   * File:   main.cpp
3   * Author: Krycke
4   *
5   * Created on February 2, 2014, 1:41 AM
6   */
7
8  #include <cstdlib>
9  #include <iostream>
10 #include <fstream>
11 #include <math.h>
```

```
12
13
14 using namespace std;
15
16 ifstream fin("test.in");
17 #ifdef LOCAL
18 #define cin fin
19 #endif
20
21 typedef unsigned long long ull;
22
23 int main(int argc, char** argv) {
24     ios::sync_with_stdio(false);
25     cin.tie(NULL);
26
27     string sat;
28     getline(cin, sat);
29     cout << sat << endl;
30     int lit;
31     int board[10][10];
32     if( sat == "SAT") {
33         while( cin >> lit ) {
34             if( lit > 99 ) {
35                 int x = lit / 100;
36                 int y = (lit % 100) / 10;
37                 int z = lit % 10;
38                 board[x-1][y-1] = z;
39             }
40         }
41     }
42     for (int x = 0; x < 9; x++) {
43         for (int y = 0; y < 9; y++) {
44             cout << board[x][y];
45         }
46         cout << endl;
47     }
```

```
48      for (int x = 0; x < 9; x++) {
49          for (int y = 0; y < 9; y++) {
50              cout << board[x][y];
51          }
52          cout << endl;
53      }
54
55
56      return 0;
57  }
```