UNIVERSIDAD SIMÓN BOLÍVAR
Ingeniería de la Computación
Diseño de Algoritmos I - CI-5651

Implementación de un SAT-Solver para la resolución de instancias SAT de sudoku

2da. Entrega

Grupo 4
Federico Flaviani 99-31744
Juan García 05-38207

15 de marzo de 2011

Índice

.

1. Introducción

Sudoku es un juego originario de Japón que tiene como objetivo rellenar una cuadrícula de 9 x 9 celdas (81 casillas) dividída en subcuadrículas de 3 x 3 (también llamadas "cajas" o "regiones") con las cifras del 1 al 9 partiendo de algunos números ya dispuestos en algunas de las celdas. La idea es que no se debe repetir ninguna cifra en una misma fila, columna o subcuadrícula. Un sudoku está bien planteado si la solución es única.

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

Figura 1: Ejemplo de un tablero de *Sudoku*

El objetivo del trabajo que se desarrollara es usar los recursos y herramientas computacionales para hallar una solución al juego del sudoku de una manera rápida y eficiente.

1.1. Motivación del proyecto

A partir del análisis de complejidad de algunos algoritmos, específicamente de aquellos que entran en el cunjunto de los NP (Nondeterministic Polynomial time), nos encontramos con el problema del sudoku general, un problema categorizado como NP-Hard, en donde hallar una solución se resume a realizar busquedas profundas dentro del ambiente de estados posibles.

La idea general es aplicar nuevos métodos que permitan hallar la solución de una manera mas eficiente. Hasta ahora hemos aplicado el método simple que es usar *Backtracking*, en donde vamos asignando variables verificando estados con soluciones parciales. Además le agregamos dos técnicas que ayudan a podar un poco el árbol de estados que se va generando, Pure Literal

Elimination en primer lugar y después Unit Propagation. Esto hace que el algoritmo sea mas rápido en encontrar la solución.

Ahora la motivación se centra en querer bajar aun mas esos tiempos y lograr un rendimiento parecido a solvers como Zchaff, y para ello intentaremos implementar la técnica de *Backtraking no cronológico* usando árboles de asignaciones e incidencias.

1.2. Breve descripción del problema

Dada una instancia de Sudoku con solo asignaciones parciales, se plantea buscar una solución, o lo que es igual, hallar el conjunto de asignaciones que deban realizarse para rellenar el tablero, y que satisfagan las condiciones de sudoku. Estas condiciones son las siguientes:

- Cada celda debe contener un número del 1 al 9.
- Una celda no puede tener dos números distintos.
- Dos casillas distintas en una misma fila, columna o subcuadrícula no puede contener el mismo número.

Para representar estas condiciones usaremos la *Forma Normal Conjuntiva (CNF)*, en donde la fórmula del problema se plantea como una conjunción de disjunciones, y el objetivo es encontrar el conjunto de asignaciones a las variables que mantengan la condición de *True* en la fórmula.

Una fórmula CNF consta de una una serie de cláusulas $C_1, C_2, ..., C_n$ agrupadas en una conjuncion $C_1 \wedge C_2 \wedge ... \wedge C_n$, donde cada cláusula es una disjuncion de literales $x_1 \vee x_2 \vee ... \vee x_n$ y cada literal representa una variable booleana X_i que puede tomar valores negados $\overline{x_i}$ o no negados x_i. Entonces un ejemplo de una formula en CNF es la siguiente:

$$(x_1 \vee x_2 \vee x_4) \wedge (\overline{x_3} \vee \overline{x_5}) \wedge (x_7 \vee \overline{x_8} \vee x_9)$$

El objetivo es encontrar asignaciones a las variables X_i tal que la formula siempre sea *True*, y entonces al asignar todas la variables, esa sera una solucion al problema. En cambio si para toda asignacion posible encontramos que la formula siempre es *False*, el problema no tiene solucion.

1.3. Descripción del contenido del informe

En el siguiente informe se dará una breve explicación del diseño e implementación del sat-solver orientado a la resolución del sudoku. Basicamente se explicarán las estructuras de datos usadas para representar el problema y los algoritmos utilizados. Además presentar aquellos problemas encontrados durante el desarrollo, una breve reseña de los elementos implementados y detalles en cuanto a la operatividad del programa. En general estamos presentando una guia para entender la manera en que atacamos la idea y como generamos una solución razonable para el problema del Sudoku.

2. Diseño

2.1. Modelo utilizado para representar el problema

El modelo general para la resolución del problema del *Sudoku* se basa en tres partes. Un proceso de traducción del problema a representación SAT, siguiendo las restricciones las reglas de resolución del juego. Luego la busqueda de la solución mediante el sat-solver, el cual recibe la traducción antes realizada. Por último tomar la salida del solver y convertirla en una solución sudoku mas legible. Por lo tanto nuestra implementación tendra 4 módulos, donde se destacan, un módulo de *Traducción* del problema, uno de *Herramientas auxiliares* a usar en el algoritmo del solver, otro módulo que implementa directamente los algoritmos que llamaremos *Solver* y por último uno de gerencia que permitira correr las diferentes instancias con una sola ejecución, a este lo llamaremos el módulo de *Solución*.

La representación SAT se realizó usando la forma normal conjuntiva (CNF) mencionada anteriormente, en donde cada cláusula que restringe el problema es presentada como una disjuncion de literales, y la fórmula total es una conjunción de cláusulas. Los literales no son mas que asignaciones *True* o *False* de variables en forma positiva (i) o negativa (-i). Fueron implementadas una serie de estructuras para representar a nivel computacional la fórmula CNF, una clausula y un literal, pero adelante se daran mas detalles al respecto.

Se utilizó la representación de un árbol de estados, en donde cada nodo es una fórmula CNF con una asignación que es solución parcial del problema. Una solución parcial del problema es cuando todas las cláusulas del problema se encuentran no asignadas totalmente o ya están satisfechas. En caso de que exista una cláusula no satisfecha ya no es solución parcial. La idea es ir generando sucesores para recorrer el árbol y buscar una solución al problema donde todas las clausulas sean satisfechas.

Un sucesor de un nodo del árbol es aquella fórmula generada a partir de una asignación hecha a una variable, y que sigue siendo una solución parcial. Por lo tanto se generará un arbol binario, ya que solo existen dos posibles sucesores, cuando le asignes *True* o *False* a una variable. Es posible generar un sucesor después de hacer varias asignaciones, en este caso hablamos de aplicar Pure Literal Elimination o Unit Propagation para hacer varias asignaciones en cascada siguiendo una serie de criterios, y esto siempre con

valores *True*. Al final una hoja del árbol sera una fórmula CNF con todas las variables asignadas, siendo esta solución o no.

2.2. Estructuras y algoritmos involucrados en la aplicación

A continuación se explicaran las estruturas y algoritmos utilizados, lo dividiremos en diferentes secciones, segun los módulos explicados anteriormente:

2.2.1. Traducción

El módulo de traducción se basa en leer las instancias parciales de sudoku, para luego mediante cláusulas en CNF representar el problema general en formato SAT-DIMACS, todo esto en conjunto con las asignaciones parciales realizadas. Todos los algoritmos utilizados fueron simples iteraciones que iban escribiendo en el archivo de salida ("out.cnf"), las cláusulas necesarias segun las restricciones del problema.

2.2.2. Herramientas Auxiliares

En este módulo destaca la creación de una lista simple enlazada, que sera la estrucutura que nos permitirá llevar un registro de literales y cláusulas de una fórmula en CNF. Básicamente se implementó el tipo abstracto *List* junto con una serie de operaciones elementales que trabajan sobre el tipo para agregar, eliminar y recorrer elementos de la misma.

2.2.3. SAT-Solver

Para el SAT-solver fue necesario crear una serie de estructuras para representar de manera general una fórmula normal conjuntiva. Se creó un arreglo para almacenar cada uno de los literales en memoria, y los mismos se guardan en una estructura con diferentes campos, específicamente un campo que guarda el valor asignado y una lista de cláusulas asociadas al literal. Para representar una cláusula se crea una estrucutra que guarda contadores sobre los literales satisfechos o no satisfechos, asi como el tamaño de una cláusula, una lista de literales que la componen y un literal que resguarda

cuando una cláusula es unitaria. Para generalizar la fórmula, se crea una estructura que representa una fórmula conjuntiva y guarda el número de literales y cláusulas totales, el arreglo de literales, el número de cláusulas satisfechas y las variables asignadas.

En la versión que implementamos del algoritmo DPLL, lo primero que se realiza es la verificación de que la fórmula de entrada es solución o no. De ser solución se retorna un valor True, si no, se procede con el grueso de la función. Primero se aplica Pure Literal Elimination, en donde aquellos literales que no tengan cláusulas asociadas induciran un valor de True en el valor opuesto de dicho literal. Luego se procede a elegir el primer literal a asignar, si no existe ningún literal para seleccionar y no existe solución, pues se retorna False, y en caso que todas las variables hayan sido asignadas y se encuentra solución, el algoritmo termina entregando la respuesta buscada. En caso de que aun existan variables por asignar, se hace la asignación de esta nueva variable y se ejecuta Unit Propagation. Unit Propagation se encargará de buscar esas cláusulas unitarias en donde se puedan asignar True al literal unitario y satisfacer la cláusula, de esta manera se hacen asignaciones cascada que pueden disminuir el tiempo de busqueda. Luego basta con llamar recursivamente DPLL con la nueva fórmula generada, si esta llamada recursiva retorna False se procede a restaurar las asignaciones hechas en el Unit Propagation y se hace backtracking para asignar el otro valor a la variable primeramente asignada.

La esencia del DPLL es ir asignando variables, y mientras se vayan generando cláusulas unitarias aplicar Unit Propagation. Y ahora gracias al backtracking no cronológico vamos a poder regresarnos a niveles superiores al inmediato anterior, y asi realizar una mejor poda del árbol de posibles soluciones.

Para realizar un backtracking no cronológico es necesario ir generando en el momento de las asignaciones un grafo de implicaciones, que será destruido en el momento del backtracking. Los nodos de este grafo son asignaciones del tipo $x_i valor, nivel, desicion)$, donde $valor$ es el valor de verdad de la variable x_i, $nivel$ es el nivel donde fué asignada la variable y $desicion$ es un booleano que determina si la asignación fué de desici'on o implicada por otras asignaciones, ademas existe un nodo adicional llamado κ que simboliza un conflicto. El grafo de implicación, es un grafo dirigido donde (a, b) es una arista si y sólo si la asignación b es implicada por a.

Una vez que el grafo de implicaciones llega a un conflicto, se procede a

buscar todas las asignaciones de desición, que son alcanzables desde κ en el grafo de implicación anterior pero con el sentido de las aristas cambiadas. Para implementar esa busqueda usamos el algoritmo de DFS recursivo.

La implementación de un algoritmo de DFS es bastante eficiente cuando se usan listas de adyacencias como estructura de datos que representan nuestro grafo. Sin embargo, el grafo al que se le aplica la busqueda en profundidad, no es el grafo de implicaciones, si no, uno igual salvo el sentido de sus aristas. Por lo tanto, tomamos la desición de implementar el grafo de implicaciones con listas de incidencias, ya que estas mismas listas serían las listas de adyacencias, del grafo invertido al que se le aplicara la busqueda en profundidad.

2.2.4. Solución

Por último tenemos el módulo de solución, en donde se recibe la salida de los respectivos solver y se procede a representar la solución obtenida en formato de sudoku general. Luego de esto se genera un archivo en formato Pdf que mostrará la cuadrícula con la solución de la instancia correspondiente.

3. Detalles de implementación

3.1. Reseña de los elementos implementados

La estructura de datos que implementa la formula proposicional en forma normal conjuntiva viene dada por la siguiente estructura:

3.1.1. Objeto Literal

La implementación del objeto literal viene dado por un registro con tres campos: *value*, *asig* y *clauses*. El primero de ellos representa el valor de verdad del literal, el segundo un tipo *Asignation* que se guardara en el árbol de asignaciones para el bactracking no cronol'ogico y el tercero es una lista de de las cláusulas que contienen a dicho literal.

```
struct literal {
  int value;
  Asignation asig;
  List clauses;
}
```

3.1.2. Objeto Cláusula

La implementación del objeto Cláusula viene dado por un registro con seis campos: $numSat$, $numNotSat$ son enteros que determinan cuantos literales están satisfechos o no en la Cláusula en determinado momento, $isSatisf$ nos dice cuando una clausula es satisfecha, *size* es el tamaño de la clusula, *literals* es una lista que contienen todos los literales de la cláusula y $unitLit$ es un apuntador que señala a un literal, en el momento en que para una solución parcial éste sea unitario.

```
struct clause {
  int numSat;
  int numNotSat;
  int isSatisf;
  int size;
  List literals;
```

```
  Literal unitLit;
}
```

3.1.3. Objeto Fórmula

La implementación de la Fórmula viene dada por un registro de 9 campos: $numLiterals$, $numVar$ y $numClauses$ son tres enteros que matienen el número total de literales, número de variables y el número de cláusulas. $literals$ y $clauses$ es un arreglo con todos los literales y una lista con la cláusulas respectivamente. $satClauses$ es el número de cláusulas satisfechas, $numVarAssig$ el número de variables asignadas en un estado dado, $isTrue$ es un booleano que indica si la fómula esta satisfecha y $assigUnitProp$ es una lista de literales asignados en una llamada a Unit Propagation.

```
struct form {
  int numLiterals;
  int numVar;
  int numClauses;
  Literal *literals;
  List clauses;
  int satClauses;
  int numVarAssig;
  int isTrue;
  List assigUnitProp;
}
```

En el arreglo de literales colocamos los $2 * |Var|$ posibles literales que existen sobre un conjunto de variables Var, en las posiciones dadas por la siguiente formula.

$$literales[j] = \begin{cases} x_{(j/2)+1} & si \quad j \ es \ par \\ \\ \overline{x_{(j+1)/2}} & si \quad j \ es \ impar \end{cases}$$

De esta forma j identifica univocamente a cada literal, y por lo tanto basta saber el índice de algún literal en particular, para ubicarlo dentro de una casilla del arreglo $literales$.

3.1.4. Objeto Asignación

Son las estructuras que representan una asignacion en nuestro árbol de asignaciones.

```
struct asignation {
  int desition;
  int variable;
  int value;
  int level;
}


struct asignationTree {
  List incidents;
  List leaf;
  Asignation asigMaxLevel;
}


struct treeVertice {
  Asignation asig;
  List incident;
}
```

3.2. Problemas encontrados

- La estructura de datos que usamos nos permite saber si una cláusula es unitaria fácilmente. Sin embargo, no podemos saber con costo constante cual es el literal unitario de cada cláusula unitaria, ya que tendriamos que recorrer toda la lista de literales para saberlo.
- La implementación de la lista utilizada para almacenar elementos nos genera una serie de memory leaks al momento de agregar un elemento. Esto debido a que la definición recursiva utilizada no soporta el inmenso overhead que se genera al reservar memoria rapidamente en cortos periodos de tiempo. Por lo tanto para instancias que tarden un tiempo considerable en ser resueltas, esto puede representar un problema a nivel de exceso de memoria utilizada.

4. Instrucciones de operación

Para la ejecución del programa se tienen dos opciones, correr un script en Python que permite realizar todas las corridas de las instancias dadas en clase, o correr por separado el traductor con alguna instancia en un formato determinado y luego el solver con el archivo en CNF de un problema en SAT.

4.1. Script en Python

La aplicación en Python llama al compilador para generar los distintos ejecutables, luego procede a corre el traductor para representar la instancia dada en formato SAT, y luego llamar a los respectivos solvers para la resolución del problema. Al obtener la solución, procede a generar los respectivos archivos en .pdf.

Con respecto a paquetes o librerias especiales en caso que quiera correr el script se necesita lo siguiente:

- Compilador Gcc para los programas escritos en C
- Libreria para correr programas en Python
- Libreria PdfLatex para generar los tableros

Y la única linea que debera correr es la siguiente:

```
$> python satSolver.py
```

4.2. Traductor y Solver

Como se mencionó anteriormente se puede compilar y correr por separados el traductor y el solver. Simplemente es necesario tener instalado un compilardor de C, recomdamos el antes mecionado Gcc.

Para compilar los diferentes programas se debe usar el comando:

```
$> make
```

Para compilar el traductor por separado solo debe realizar la llamada:

```
$> make translator
```

Y para realizar la corrida:

```
$> ./translator N_InstanciaParcial out.cnf
```

El segundo argumento es una isntancia parcial del sudoku, en donde el primer numero N es el tamaño del sudoku, luego un underscore (_) y seguido la Instancia con asignación parcial al estilo de las entregadas en clase del archivo "InstanciasSudoku.txt". La traduccion a SAT se entrega en el archivo "out.cnf"

Luego para compilar el solver debe llamar:

```
$> make solver
```

Y para correr el programa:

```
$> ./solver out.cnf
```

Donde out.cnf es la traducción del problema de Sudoku a SAT.

Para compilar el solver de ZChaff:

```
$> cd zchaff64/
/zchaff64 $> make
```

Y debe correrse con la llamada a:

```
$> ./zchaff64/zchaff out.cnf
```

5. Estado actual

5.1. Estado final de la aplicación

El estado final de la aplicación no es operativo al 100 %, específicamente en el módulo del solver. El traductor realiza su función perfectamente, tomando del archivo las instancias de sudoku y entregando la traducción a CNF. De igual manera el módulo de solución, en donde se toma la salida de los solvers y se entrega en formato sudoku pdf también funciona.

5.2. Errores

Aunque fue resuelta la problemática con el algortitmo de DPLL, tenemos algunos memory leaks que hacen que para instancias que tardan mucho tiempo, el consumo de memoria sea muy grande, pudiendo llegar a veces a dejar sin memoria al equipo y entonces el sistema operativo manda una señal de kill al proceso. Para instancias que duran poco tiempo, aproximadamente menos de 20 min, corre perfecto y encuentra la solución.

Otro problema es que no se logro implementar el arbol de asignaciones de manera correcta, por lo tanto el grafo generado no representa la estructura ideal para realziar un backtracking a mayores niveles. Son problemas en la elección del literal a asignar, que esta creando ciclos en el grafo, lo que genera una problemática al momento de buscar una solución.

6. Conclusiones y recomendaciones

Dado que no fue posible cumplir los objetivos planteados en el problema, son pocas las conclusiones que podemos sacar. En general encontramos una serie de problemas para implementar una buena solución al problema, mas que todo por el enfoque que dimos a nivel de programación de los algoritmos. Por lo tanto resulta de vital importancia plantear una solución mas eficiente y concisa de las funciones que se vayan a implementar para desarrollar el solver. Aunque no fue posible implementar la solucion optima usando backtracking no cronológico, se notaron cambios considerables en los tiempos de corrida al resolver el problema, pasando de minutos a pocos segundos. Esto demuestra que la tecnica aplicada realiza una poda significativa en el arbol de estados.

6.1. Resultados obtenidos

A continuación se presentan los tiempos de las corridas del solver hecho en el proyecto y el solver Zchaff sobre las instancias dadas en el archivo "InstanciasSudoku.txt"

Instancia	Solver propio	Solver Zchaff
1	225.671 ms	0.00786 ms
2	$\geq$ 5 min	0.00381 ms
3	$\geq$ 5 min	0.00405 ms
4	80.051 ms	0.00691 ms
5	14872.498 ms	0.00786 ms
6	29.879 ms	0.00715 ms
7	816.949 ms	0.00691 ms
8	135.041 ms	0.00715 ms
9	167.229 ms	0.00786 ms
10	125827.757 ms	0.00810 ms
11	2371.787 ms	0.00691 ms
12	2416.908 ms	0.0128 ms
13	547.145 ms	0.00715 ms
14	56456.277 ms	0.00786 ms
15	$\geq$ 5 min	0.00381 ms
16	137.175 ms	0.00691 ms
17	90.197 ms	0.00691 ms

18	$\geq$ 5 min	0.00405 ms
19	71.579 ms	0.00715 ms
20	46656.309 ms	0.00810 ms
21	54.132 ms	0.00905 ms
22	239.806 ms	0.00786 ms
23	7772.205 ms	0.00691 ms
24	185214.636 ms	0.00691 ms
25	$\geq$ 5 min	0.00286 ms
26	$\geq$ 5 min	0.00309 ms
27	$\geq$ 5 min	0.00405 ms
28	144028.009 ms	0.00715 ms
29	428.553 ms	0.00691 ms
30	2008.273 ms	0.00786 ms
31	391.221 ms	0.00691 ms
32	5074.174 ms	0.00715 ms
33	$\geq$ 5 min	0.00309 ms
34	144701.276 ms	0.00786 ms
35	$\geq$ 5 min	0.00405 ms
36	$\geq$ 5 min	0.00381 ms
37	1203.404 ms	0.00810 ms
38	20366.432 ms	0.00715 ms
39	50972.270 ms	0.00691 ms
40	395472.976 ms	0.00715 ms
41	146.774 ms	0.00691 ms
42	158.635 ms	0.00691 ms
43	$\geq$ 5 min	0.00286 ms
44	17252.228 ms	0.00715 ms
45	524.287 ms	0.00715 ms
46	383.146 ms	0.00715 ms
47	169821.164 ms	0.00691 ms
48	940.442 ms	0.00715 ms
49	674.192 ms	0.00786 ms
50	172.515 ms	0.00691 ms
51	57.373 ms	0.00786 ms
52	84.462 ms	0.00786 ms
53	294.802 ms	0.00810 ms
54	77.813 ms	0.00691 ms

55	163.821 ms	0.00715 ms
56	909.723 ms	0.00810 ms
57	$\geq$ 5 min	0.00309 ms
58	84.602 ms	0.00786 ms
59	46.894 ms	0.00786 ms
60	551.743 ms	0.00786 ms

6.2. Mejoras

- Debemos implementar una estructura de lista más eficiente con relación a la memoria usada, porque dado el overhead de reservar mucha memoria en poco tiempo, cuando el programa tarda demasiado la memoria usada crece demasiado.
- Realizar una mejora en el Backtracking no cronológico y modificar el metodo que agrega la clausula de aprendizaje para que lo haga mas eficiente.

7. Referencias bibliográficas

Referencias

[1] **Efficient Data Structures for Fast SAT Solvers**
LYNCE I., MARQUES-SILVA J.

[2] **Satisfiability Suggested Format**

[3] **Sudoku as SAT Problem**
LYNCE I., OUAKNINE J.

[4] **urlhttp://www.cs.ubc.ca/ hoos/SATLIB/Benchmarks/SAT/satformat.ps**

[5] **GRASP - A New Search Algorithm for Satisfiability**
MARQUES SILVA J., SAKALLAH K.

[6] **BerkMin: a Fast and Robust Sat-Solver**
GOLDBERG E., NOVIKOV Y.