FACULTAD DE INGENIERIA – INSTITUTO DE COMPUTACIÓN

Programación Lógica

OBLIGATORIO Nº2

MATIAS FABRIZIO PÉRES - 4474045
GERMÁN RUIZ - 4317743
PEDRO DANIEL CREMONA - 3991783

JUNIO DEL 2012

1 ÍNDICE

1 Índice 3

2 Introducción 4

3 Estructura de la solución 5

4 Profiler 5

5 Análisis de Eficiencia 8

6 Especificación del Esquema de codificación Sat 10

7 N Values 12

8 Comparacion tiempos de ejecución entre n_values y latin_square_many 14

9 Tiempos de ejecución del latin_square_sat 14

10 Conclusión 16

11 Bibliografía 17

2 INTRODUCCIÓN

En el presente trabajo se describe el proceso de resolución del obligatorio número dos de la materia Programación Lógica del Instituto de Computación

Los objetivos del informe son: enumerar los archivos que componen la solución, describir los análisis de performance solicitados, especificar el esquema de codificación SAT y describir la solución implementada incluyendo las decisiones de diseño relevantes.

El trabajo se encuentra respaldado por material bibliográfico confiable que ha sido seleccionado con atención, a efectos de que la información en la cual nos basemos para la realización del mismo sea certera, lo que brinda constantemente una seguridad en la lectura.

3 ESTRUCTURA DE LA SOLUCIÓN

Nuestra solución consiste de dos archivos, puzle.pl y latin_sat.pl. En el primero se encuentran todos los predicados que resuelven la primer parte del obligatorio. Los predicados principales son:

solve_puzzle(+File,+N,?Solution)
profiler(+Goal,-Profile).

El segundo archivo resuelve la segunda parte del obligatorio sobre cuadrados latinos. Tiene todos los predicados para resolver el problema SAT y además los de conseguir K cuadrados latinos utilizando prolog puro. Los predicados principales de este archivo son:

latin_square_sat(+N, ?M)
latin_square_sat_many(+N,+HowMany,?Bag)
n_values(+Goal,+Template,+K,-Bag)

4 PROFILER

Una vez implementado el resolvedor de puzles, era deseable tener un análisis de eficiencia de la solución implementada y su comportamiento al variar las condiciones del puzle. En particular, variando su tamaño y el número de colores.

Una manera particular de analizar la solución implementada es ejecutar un 'profiler'. Los profilers juegan un rol importante en el desarrollo de programas eficientes. Muy a menudo no es posible anticipar, de ante mano, las sentencias críticas de un programa. En esos casos, es usual utilizar profilers para obtener datos representativos de los datos de entrada, para luego reconocer las porciones críticas del programa y hacerlas más eficientes. En contraste, un programador que intenta reconocer a ciegas las partes críticas, sin saber en qué porciones concentrarse, perderá una gran cantidad de tiempo y esfuerzo optimizando código que tal vez se ejecuta infrecuentemente, resultando en mejoras de performance marginales.

En nuestro caso, siguiendo las especificaciones de las clausulas en el obligatorio, desarrollamos un meta intérprete que ejecuta un 'Goal' y además devuelve una lista de todos los predicados involucrados durante la ejecución y cuántas veces se ejecutó cada uno.

Informalmente, un intérprete es un programa que evalua programas. Desde una perspectiva teórica y práctica, en la ciencia de la computación, la interpretación es invasiva, y muchos programas son intérpretes de un dominio específico de lenguajes. Por ejemplo, un programa que lee un archivo de configuración y se ajusta en forma acorde está interpretando ese 'lenguaje de configuración'.

Un intérprete para un lenguaje similar o idéntico a su propia implementación se llama 'meta-interpreter' (MI). Un intérprete que se puede interpretar a si mismo se llama 'meta-circular'. Prolog propicia la escritura de MIs. Primero y más importante, programas en prolog pueden ser representados, naturalmente como términos de prolog, y son fácilmente inspeccionados y manipulados usando mecanismos 'built-in'. Segundo, la estrategia de computación implícita de prolog y todos los predicados solución pueden ser usados en interpretes, permitiendo especificaciones concisas. Tercero, las variables del nivel-objeto (el programa a ser interpretado) puede ser tratadas como variables en el nivel-meta (el interprete). Por lo tanto, un intérprete puede delegar el manejo del programa interpretado al motor de prolog subyacente.

El meta intérprete más simple es el siguiente programa:

```
solve(Goal):- call(Goal).
```

Sin embargo, no ventaja en usar este intérprete, dado que inmediatamente llama al intérprete de prolog. Un intérprete mucho más popular es el intérprete 'vanilla' que usa la unificación build-in de prolog pero es fácil de modificar y cambiar, por ejemplo, el orden de ejecución.

```
solve(true).
solve((A,B)):-
        solve(A),solve(B).
solve(A):-
        clause(A,B),solve(B).
```

Notar que el meta interprete 'vanilla' utiliza el predicado build-in *clause(H,B)*, que haya una regla en un programa prolog cuya cabeza unifica con H y cuerpo B. Si no hay cuerpo, entonces *cuerpo=true*.

Mantener una cuenta del número de llamadas a un predicado, es conceptualmente, una cuestión simple. En cada llamado, un contador para el

predicado es incrementado. De forma de acceder a la entrada apropiada de una tabla de contadores, primero es necesario extraer el nombre del predicado y la aridad de la llamada. Esto se realiza con el predicado built-in *functor(Goal, Pred, Arity)*. Nuestro meta interprete luce entonces como:

```
solve(true).
solve((A,B)):-
        solve(A),solve(B).
solve(Goal):-
        functor(Goal, Pred, Arity),
        entry_update(Pred,Arity,AuxProf)
        clause(Goal,Body),solve(Body).
```

Notar el llamado a *entry_update*. Dentro de este procedimiento se incrementa el contador para el predicado de nombre *Pred* y aridad *Arity*. Un caso particular que no se muestra en lo anterior es en el que diferenciamos el tipo de llamada. Si se trata de una llamada built-in, entonces no realizamos el llamado a *clause* ni el llamado recursivo sino que simplemente la ejecutamos sin interpretar. De otra forma se incurre en un error de acceso a predicados privados de prolog.

Es importante mencionar una particularidad la solución desarrollada. Esta es, la forma en que se lleva cuenta de los llamados. Los sistemas de prolog típicamente usan las primitivas *assert* y *retract* para manipular y preservar valores durante el back tracking. Por ejemplo, un programador que desea incrementar un contador y preservar su nuevo valor durante el backtracking, podría escribir algo como:

```
..., cnt(C0), C1 is C0+1, retract(cnt(C0)), assert(cnt(C1)), ...
```

Sin embargo, *assert* y *retract* son generalmente rutinas de propósitos generales que pueden incurrir en una significante pérdida de performance. Es por ello que se decidió utilizar los predicados *nb_setarg(+N,+Term,+Value)*, setea el N-ésimo argumento del término Term como Value y *arg(+N,+Term,-Value)*, obtiene el N-ésimo argumento de Term. Estos predicados no son afectados por el back tracking de prolog.

Veamos un trozo de su utilización:

```
entry_update(Pred, Arity, Counter):- arg(1,Counter,L),
delete_member([Pred,Arity,Cont],L,LR), NewCont is Cont+1,
nb_setarg(1,Counter,[[Pred,Arity,NewCont]|LR]), !.
```

entry_update(Pred, Arity, Counter):- arg(1,Counter,L), nb_setarg(1,Counter,[[Pred,Arity,1]|L]).

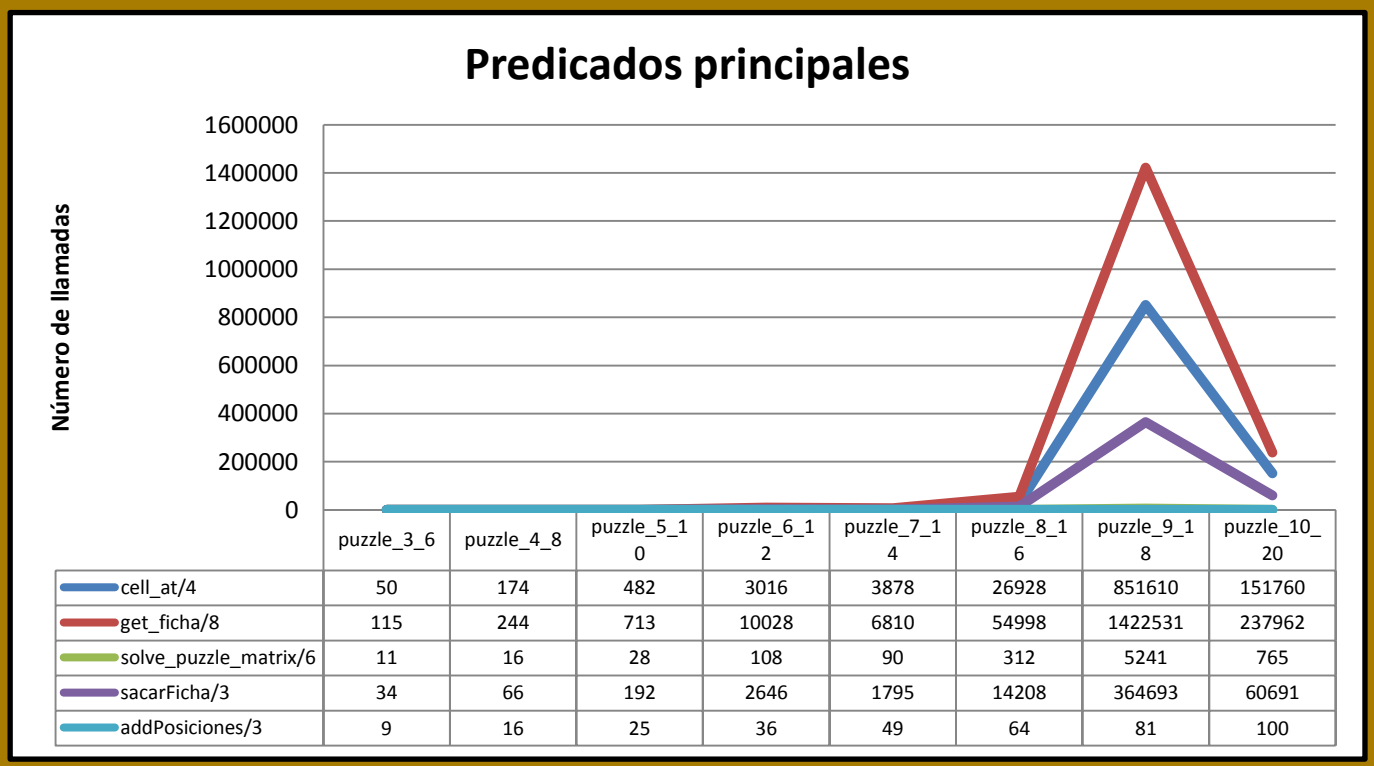
Predicado *Profiler:*

profiler(Call,Profile) :- Counter = counter([]), mi1(Call,Counter),!, makeProfiler(Counter,Profile).

La variable *Counter* nos permite hacer uso de los predicados discutidos. *makeProfiler* simplemente realiza el pasaje de [Pred,Arity,Count] a [Pred/Arity,Count] como se especifica en las clausulas del obligatorio.

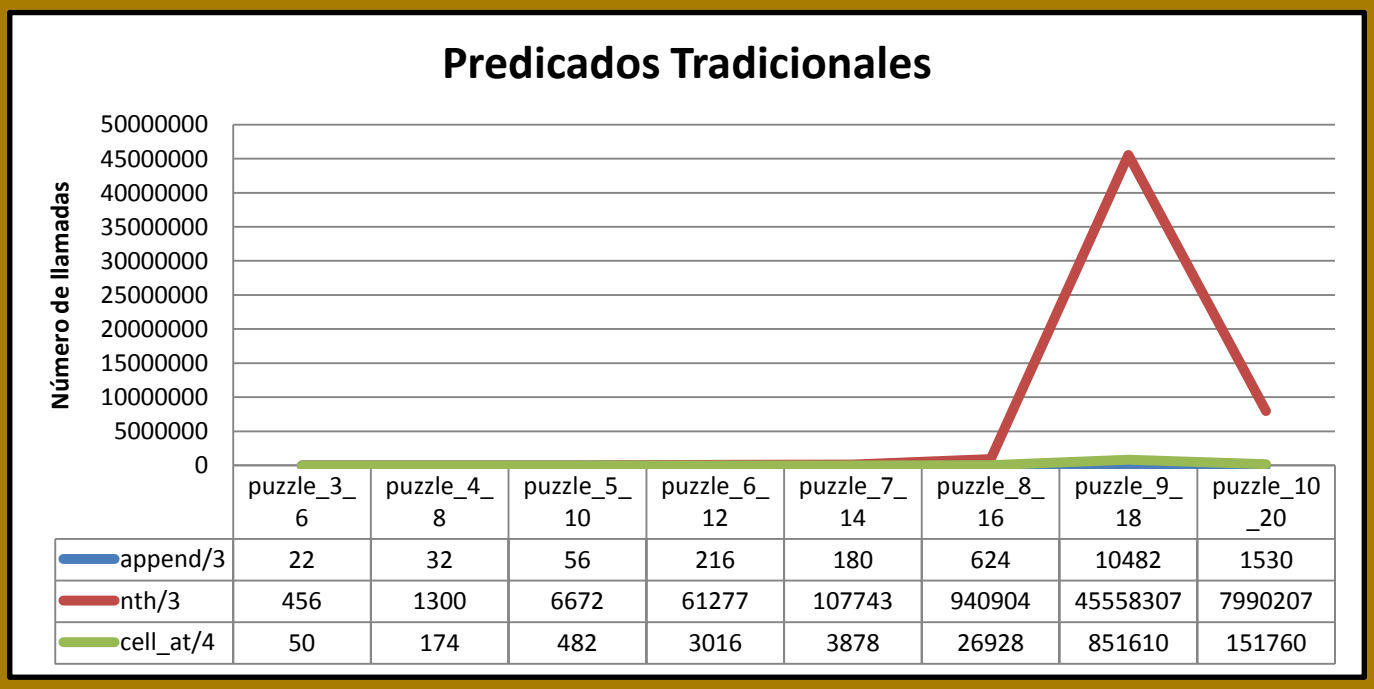
5 ANÁLISIS DE EFICIENCIA

Gracias al meta intérprete desarrollado fue posible observar los siguientes resultados al ejecutar los puzles dados con el obligatorio. Mostramos los predicados más relevantes y cuyo porcentaje de llamadas se mantiene alto



	puzzle_3_6	puzzle_4_8	puzzle_5_10	puzzle_6_12	puzzle_7_14	puzzle_8_16	puzzle_9_18	puzzle_10_20
cell_at/4	50	174	482	3016	3878	26928	851610	151760
get_ficha/8	115	244	713	10028	6810	54998	1422531	237962
solve_puzzle_matrix/6	11	16	28	108	90	312	5241	765
sacarFicha/3	34	66	192	2646	1795	14208	364693	60691
addPosiciones/3	9	16	25	36	49	64	81	100

En primer lugar, en el cuadro Predicados Principales, podemos notar como el número de llamadas, especialmente a los predicados *cell_at/4* y *get_ficha/8,* crece de manera espontánea y en cuantiosa cantidad cuando se ejecuta el puzle *puzzle_9_18*. Este comportamiento desconcierta bastante y debe su causa a la estructura misma del algoritmo desarrollado. En el puzle *puzle_10_20*, a pesar de tener mayor tamaño y más colores, no se realizan tantas llamadas como en el anterior.

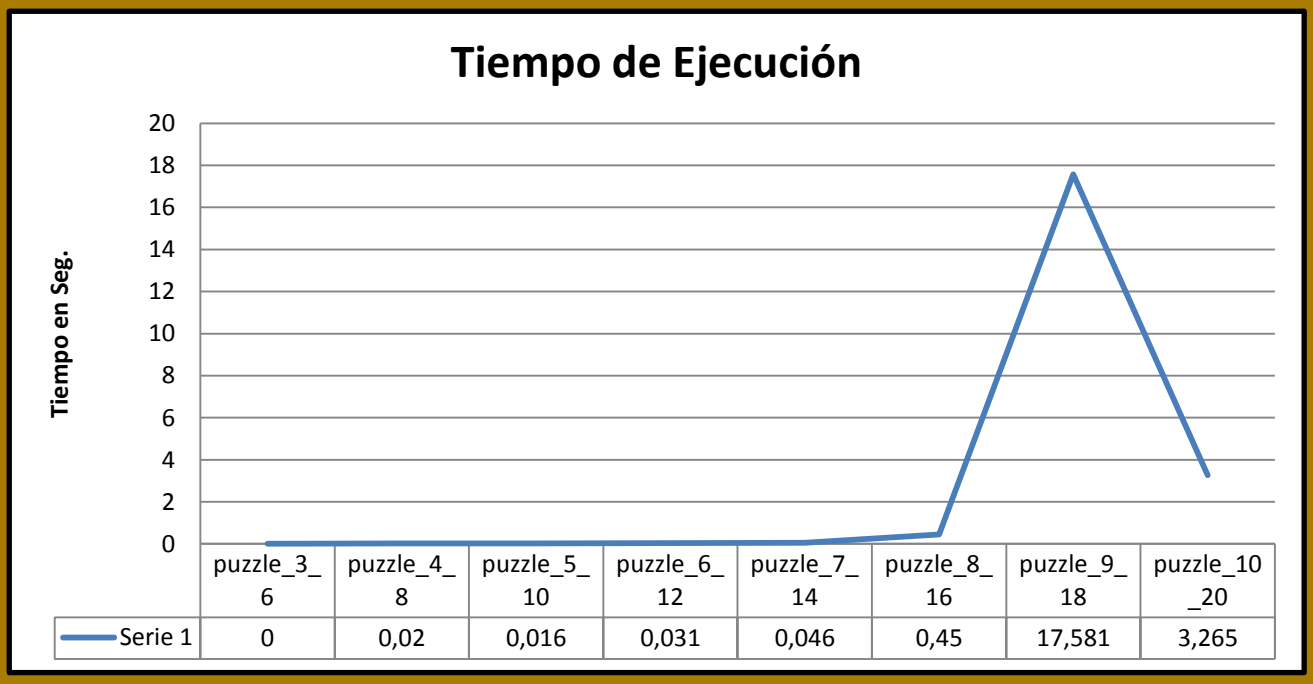


	puzzle_3_6	puzzle_4_8	puzzle_5_10	puzzle_6_12	puzzle_7_14	puzzle_8_16	puzzle_9_18	puzzle_10_20
append/3	22	32	56	216	180	624	10482	1530
nth/3	456	1300	6672	61277	107743	940904	45558307	7990207
cell_at/4	50	174	482	3016	3878	26928	851610	151760

En el segundo recuadro, observamos una desnivelación en la cantidad de llamadas, similar a la observada en la gráfica anterior. En este caso, se trata de

predicados auxiliares vistos y desarrollados en el obligatorio anterior. Aquí, el número de llamadas a *nth/3* se dispara al resolver el puzle de 9 fichas. Esta observación es la principal finalidad de realizar este tipo de análisis, dado que nos permite deducir cuán necesaria es la eficiencia de *nth/3*. Notar que el número de llamadas se eleva al número de 45 millones aprox. También podemos decir lo mismo de los predicados *cell_at/4* y *get_ficha/8*, su eficiencia es importante y si se tuvieran problemas en la resolución de otros puzles más complejos, los primeros predicados en que nos deberíamos concentrar son los mencionados.

Además de utilizar el profiler desarrollado para estudiar la cantidad de llamados, se estudio el tiempo de ejecución del algoritmo para los puzles, obteniéndose el siguiente gráfico:



	puzzle_3_6	puzzle_4_8	puzzle_5_10	puzzle_6_12	puzzle_7_14	puzzle_8_16	puzzle_9_18	puzzle_10_20
Serie 1	0	0,02	0,016	0,031	0,046	0,45	17,581	3,265

Como es de esperarse, el tiempo del puzle de 9 fichas supera por mucho a los demás y decayendo nuevamente en el puzle de 10 fichas.

6 ESPECIFICACIÓN DEL ESQUEMA DE CODIFICACIÓN SAT

Nuestro problema SAT es el de cuadrados latinos de un orden N dado. Como ya sabemos un cuadrado latino de orden N es una matriz compuesta enteramente por enteros entre 1 y N, y cuyas filas y columnas no contienen elementos repetidos.

Entonces lo primero que necesitamos en nuestro esquema de codificación es como representar un cuadrado latino de orden N con variables booleanas. Como cada posición de la matriz debe tener un entero entre 1 y N entonces para cada posición de estas vamos a necesitar N variables booleanas. Por lo tanto la representación de un cuadrado latino de orden N nos quedaría:

Xije donde: i,j,e en [1 .. N]

Xijv modela la celda en la posición (i,j) de la matriz, con valor asignado e. Con esta codificación tendríamos un total de N^3 variables.

A continuación hay que codificar las clausulas correspondientes a nuestro problema SAT. La primer restricción de nuestro problema es que en cada celda hay un solo valor, ya que es una matriz de enteros. Cada celda tiene dominio [1 .. N], por lo tanto hay una clausula para cada celda especificando este dominio.

Para todo i,j (Xij1 v Xij2 v ... v XijN)

Luego la siguiente restricción es que no hallan elementos repetidos en la misma fila. Para cumplir esta restricción utilizamos clausulas binarias. Comparamos de a pares en cada celda de la fila con el valor entero correspondiente:

Para todo i,e (not Xi1e v not Xi2e), (not Xi1e v not Xi3e) , … , (not Xi1e v not XiNe) , (not Xi2e v not Xi3e), … , (not XiN-1e v not XiNe)

La siguiente restricción es igual pero para las columnas por lo tanto:

Para todo j,e (not X1je v not X2je), (not X1je v not X3je) , … , (not X1je v not XNje) , (not X2je v not X3je) , … , (not XN-1je v not XNje)

Observar que con la primer restricción se puede dar que una celda tenga varios valores pero en combinación con las otras dos reglas esta situación no puede ocurrir.

Pasando a la solución en prolog de encontrar un cuadrado latino de orden N mediante MiniSat, el predicado que se ocupa de esto se divide en tres etapas. La primera es la codificación del problema SAT en un archivo de entrada el cual se le pasa al SAT-Solver MiniSat. Utilizando operaciones de escritura se crea este archivo de entrada en el formato requerido por MiniSat. Se le pasa la cantidad de variables y clausulas en la primer línea. Luego se le pasan todas las clausulas mencionadas en nuestra especificación del problema.

Luego en la segunda etapa se ejecuta MiniSat con el archivo de entrada creado. Este interpreta la formula en FNC especificada en el archivo y nos devuleve un archivo conteniendo los valores correspondientes a cada variable que resuelven nuestro problema.

En la última etapa se decodifica la salida y se crea la matriz solución correspondiente al cuadrado latino de orden N ingresado.

Para la siguiente parte donde se pide hacer un predicado donde se devuelvan los primeros K cuadrados latinos encontrados se siguió el procedimiento indicado en la letra. Para el primer cuadrado latino a buscar se crea el archivo de entrada como antes. Cuando se encuentra la solución no se crea de vuelta de

cero el archivo, solo se modifica agregándole como clausula la solución anteriormente hallada. Es decir cada vez q se encuentra una solución se agrega una clausula a la entrada negando la solución para que en la próxima no se encuentre la misma.

7 N VALUES

Como vimos en el primer laboratorio al calcular cuadrados latinos grandes pueden llegar a tomar demasiado tiempo, por lo cual podríamos en vez de calcular todos los cuadrados latinos posibles utilizando findall calcular solamente una cierta cantidad dada.

Para esto deberíamos ejecutar el cuadrado latino y cuando encontró una solución verificar si tenemos que seguir buscando más soluciones hasta ver si encontrando la cantidad de soluciones que queríamos y ahí no continuar con dicha búsqueda

La forma de hacerlo es viendo como se hace el backtracking internamente, para esto ejecutamos el goal, guardamos el valor del template tomado al cumplirse el objetivo en una estructura que no sea "desarmada" por el backtracking(non Backtrackeable) para esto usamos el assertz(queue(Template)) para que agregue en forma de cola a la base de datos de Prolog, luego para el contador de cantidad de soluciones que debe ser decrementado mientras se hace el backtracking también precisamos una estructura similar ya que no se debe perder su valor ya que sino no sabríamos cuantas soluciones tomar, por eso utilizamos los predicados vistos en el curso nb_getval y nb_setval (nb es la sigla de non Backtrackeable) que pasando un nombre identificador setear un valor a la variable con dicho identificador.

En resumen el algoritmo consiste en:

- Buscar una solución.
- Agregarla a la cola de la base de datos de swi prolog.
- Decrementar la variable global que cuenta la cantidad de "backtraking"s.
- Si la variable es 0 entonces agrego el tope a la cola (bottom) y un cut para terminar.
- Por último cargo los valores almacenados en la base de datos en la lista Bag.

Nota: Se asume que el valor de K es mayor a 0 siempre y también se asume que existe esa K cantidad de soluciones.

Se tomaron algunas ideas como dice de letra del predicado findall en particular se tomaron ideas del código dehttp://ai.ia.agh.edu.pl/wiki/prolog:pllib:findall

Algunas pruebas realizadas:

1 ?- n_values(latin_square(3,X),X,4,B).

X = [[3, 1, 2], [1, 2, 3], [2, 3, 1]],

B = [[[3, 2, 1], [2, 1, 3], [1, 3, 2]], [[3, 2, 1], [1, 3, 2], [2, 1, 3]], [[3, 1, 2], [2, 3, 1], [1, 2, 3]], [[3, 1, 2], [1, 2, 3], [2, 3|...]]].

2 ?- n_values(latin_square(3,X),X,2,B).

X = [[3, 2, 1], [1, 3, 2], [2, 1, 3]],

B = [[[3, 2, 1], [2, 1, 3], [1, 3, 2]], [[3, 2, 1], [1, 3, 2], [2, 1, 3]]].

3 ?- n_values(latin_square(2,X),X,1,B).

X = [[2, 1], [1, 2]],

B = [[[2, 1], [1, 2]]].

4 ?- n_values(latin_square(4,X),X,2,B).

X = [[4, 3, 2, 1], [3, 4, 1, 2], [2, 1, 3, 4], [1, 2, 4, 3]],

B = [[[4, 3, 2, 1], [3, 4, 1, 2], [2, 1, 4, 3], [1, 2, 3, 4]], [[4, 3, 2, 1], [3, 4, 1, 2], [2, 1, 3, 4], [1, 2, 4|...]]].

5 ?- n_values(latin_square(4,X),X,1,B).

X = [[4, 3, 2, 1], [3, 4, 1, 2], [2, 1, 4, 3], [1, 2, 3, 4]],

B = [[[4, 3, 2, 1], [3, 4, 1, 2], [2, 1, 4, 3], [1, 2, 3, 4]]].

8 COMPARACIÓN TIEMPOS DE EJECUCIÓN ENTRE N_VALUES Y LATIN_SQUARE_MANY

Para ver cuánto tiempo demora n_values utilizamos la siguiente línea :get_time(X),n_values(latin_square(2,M),M,K,Bag),get_time(Y),T is Y-X.

Para latin_square_many utilizamos: :get_time(X),latin_square_sat_many(N,K,Bag),get_time(Y),T is Y-X.

Los resultados encontrados fueron:

	n_values	latin_square_sat_many
N=2 K=1	0	0,0599
N=3 K=2	0	0,105
N=3 K=8	0,001999	0,4389
N=4 K=2	0,0299	0,1119
N=4 K=10	0,0559	0,5039
N=20 K=1	> 2 min	13,654
N=20 K=10	> 2 min	57,059
N=30 K=30	> 2 min	52,428

Los tiempos se encuentran en segundos.

Claramente el latin_square de prolog puro es más eficiente para valores pequeños de N pero ya para valores mayores a 4 el tiempo crece demasiado y se vuelve impráctico de usar, en cambio latin_square_sat tiene un crecimiento bajo en el tiempo según el tamaño del cuadrado latino. Como se puede ver en la tabla funciona muy bien ya que calcula en menos de un minuto 30 cuadrados latinos de orden 30 algo que demoraría demasiado hasta con una solución muy eficiente en prolog puro.

9 TIEMPOS DE EJECUCIÓN DEL LATIN_SQUARE_SAT

Claramente vemos que la solución utilizando SAT es muy eficiente para calcular cuadrados latinos grandes, ahora analizaremos los tiempos de la

codificación de la entrada que se le pasa al SAT-Solver, de la ejecución por parte de MiniSat, y de la decodificación de la salida dada por el mismo.

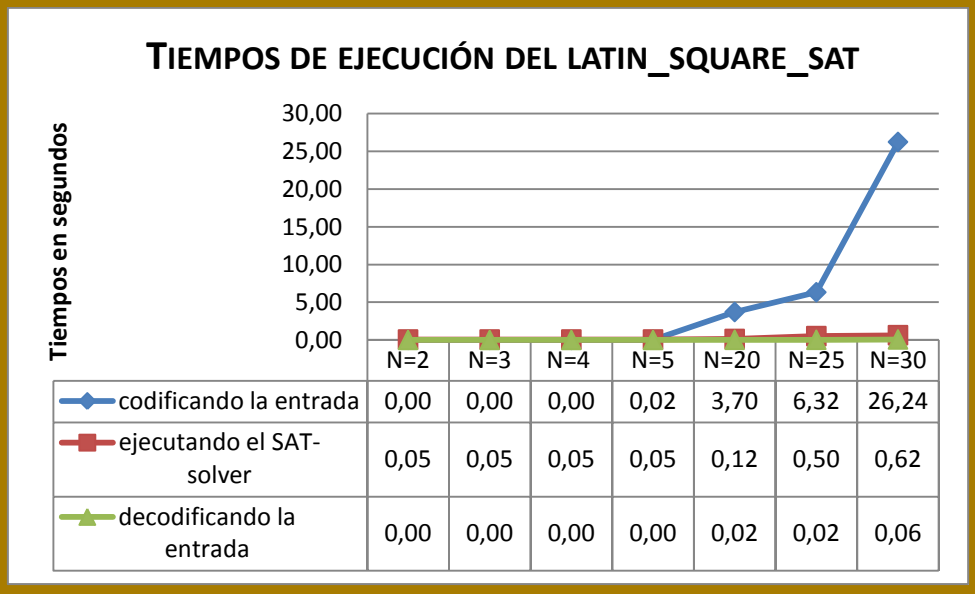
Para realizar esta prueba utilizamos:

```
latin_square_sat_t(N,M,Tcod,Tej,Tdec) :- N>1, get_time(Tuno),
codificarEntrada(N),get_time(Tdos),get_time(Ttres),
ejecutarMiniSat,get_time(Tcuatro),get_time(Tcinco),
decodificarSalida(N,M),get_time(Tseis),
Tcod is Tdos-Tuno,Tej is Tcuatro-Ttres,Tdec is Tseis-Tcinco,!.
```

Tabla con los tiempos en segundos según N:

	codificando la entrada	ejecutando el SAT-solver	decodificando la entrada
N=2	0.00099	0.0520	0.0
N=3	0.0040	0.0509	0.0
N=4	0.0	0.0510	0.0
N=5	0.0189	0.0520	0.0
N=20	3.6979	0.1240	0.0160
N=25	6.3179	0.5000	0.015
N=30	26.239	0.6240	0.0620

Gráfica:



Claramente el mayor tiempo es codificando la entrada. El tiempo de ejecución crece según N pero no mucho, y la decodificación siempre ocupa muy poco tiempo de ejecución. Esto se debe a que en la codificación se hacen muchas operaciones de escritura en el archivo de entrada, y este crece mucho respecto al N ingresado. También se observa que el Sat-Solver es muy eficiente incluso pasándole un archivo de entrada con más de 100 mil variables y 2 millones de

clausulas, mini-sat es capaz de resolver el problema en aproximadamente medio segundo.

10 CONCLUSIÓN

Para finalizar el informe sobre la realización del obligatorio número uno, es importante mencionar los objetivos cumplidos.

El proceso hacia la formalización, ha tenido como propósito, ser coherentes en el desarrollo del informe, a diferencia de las ocasionales contradicciones que pueden encontrarse en piezas de información varias. Se pretendió fijar la atención en aspectos importantes y esenciales para la comprensión del desarrollo.

Debemos mencionar que las decisiones tomadas a lo largo de la implementación surgieron como respuesta a los resultados obtenidos hasta el momento.

11 BIBLIOGRAFÍA

[1] Profiling Prolog Programs - *Saumya K. Debray* - Department of Computer Science - The University of Arizona, Tucson, AZ 85721

[2] http://ktiml.mff.cuni.cz/~bartak/prolog/meta_interpret.html

[3] http://web.student.tuwien.ac.at/~e0225855/acomip/acomip.html

[4] file:///D:/bdi_docs/Desktop/manipterm.html

[5] http://www.ai.uga.edu/ftplib/ai-reports/ai198908.pdf