Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Un *sat-solver* paralelo y distribuido con herencia de cláusulas aprendidas

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Ignacio Vissani

Directores: Nicolás Leandro Rosner, Carlos Gustavo López Pombo

Buenos Aires, 2013

UN *sat-solver* PARALELO Y DISTRIBUIDO CON HERENCIA DE CLÁUSULAS APRENDIDAS

En la actualidad el *software* forma parte integral de nuestra vida. Este interviene en todo tipo de tareas. Desde el envío de un mensaje de texto o la realización de una llamada telefónica hasta el sistema de control de una central nuclear o el sistema de velocidad crucero de un automóvil, utilizan sistemas basados en *hardware* y *software* informático.

El análisis de software es un área en la ingeniería de software que en general se ocupa de la aplicación de métodos formales que permiten garantizar, o al menos ganar confianza en la ausencia de errores en una pieza de *software*. Existe una plétora de herramientas dedicadas a esta tarea, cada una de ellas con sus elecciones particulares de lenguajes y método a aplicar. Entre ellas, una cantidad no menor se ha volcado al uso de *sat-solvers off-the-shelf* como mecanismo de bajo nivel que permite explorar eficientemente un espacio finito de modelos.

En el presente trabajo pretendemos implementar un *sat-solver* paralelo y distribuido que: a) realice una adecuación y balanceo de la carga de procesamiento en *run-time* sobre una cantidad y configuración de equipamiento desconocida, b) realice una adecuación y balanceo del uso de memoria primaria y secundaria en *run-time*, también sobre una cantidad y configuración de equipamiento desconocida, y c) sea extensible y fácil de parametrizar en todos aquellos aspectos que determinen el comportamiento del procedimiento de resolución del problema. Una vez llevada a cabo esta tarea, pretendemos evaluar la viabilidad de implementar una técnica de aprendizaje basada en análisis de cláusulas de conflicto acorde a las necesidades de ambientes paralelos y distribuidos, y su comparación empírica.

Palabras clave: Ingeniería de software, verificación y validación, sat-solver, paralelo, distribuido, aprendizaje basado en conflictos.

A PARALLEL AND DISTRIBUTED *sat-solver* WITH LEARNED CLAUSES INHERITANCE

Today, software is an integral part of our lives, participating in all types of tasks. Text messages, phone calls, the control system of a nuclear power plant or the cruise control setting on a car – all of these things affect our daily lives and all employ systems based on hardware and software.

In general, software analysis is a field within software engineering responsible for the application of formal methods in order to guarantee (or to at least gain some confidence about) the absence of errors in a software artifact. There is a plethora of tools dedicated to this task, each one with its own particular biases for specific languages and the methods to be applied. Many have turned to the use of off-the-shelf *sat-solvers* as low-level mechanisms that allow for efficient exploration of a finite space of models.

The aim of the present work is to implement a parallel and distributed *sat-solver* that: a) provides an adequate processing workload balance at run-time over an *a priori* unknown quantity and configuration of equipment, b) provides an adequate balance of memory use, both primary and secondary, at run-time over an unknown quantity and configuration of equipment, c) is extensible and easy to parameterize in all the relevant aspects that determines the behaviour of the tool in the solution of the problem. Once this task is done, we will evaluate the viability of the implementation of learning techniques based in conflict clause analysis according to the needs of parallel and distributed scenarios, and their empirical comparison.

Keywords: Software engineering, Verification and validation, Sat-solvers, Parallel and distributed, Conflict based learning.

AGRADECIMIENTOS

Intento determinar cuál fue el punto en el tiempo en el que todo empezó. No lo tengo muy claro. Mis recuerdos más viejos son para Nacho Regueira y Alejandro Abramoff. Cada uno con sus respectivas `XT` en las que hacíamos carteles con el Banner o jugábamos al Arkanoid ¿O tal vez fue en lo de Martín González? Un cumpleaños, sí. Una Commodore con un juego ¿Galaga tal vez? Después de eso, laguna, nada. Se me viene la 286, la primera compu "mía". Una flamante 286 con monitor color. Creo que fue un regalo de cumpleaños. Un regalo de mis viejos que era más bien para toda la familia. No recuerdo el día que llegó, pero sí recuerdo el día en que "accidentalmente" la *formatié*. El técnico que vino aquella vez fue el primero que me enseñó algo de *computación*. En aquella aprendimos Quattro Pro y Lotus123 con mi mamá. Después se me viene el lugar de los jueguitos, en el que parábamos con mi papá a veces cuando íbamos o veníamos de lo de la abuela Pocha. Creo que con el de Los Simpson nos agarramos el primer virus. El famoso Michellangelo. Me acuerdo también de cuando intenté, infructuosamente, instalarle Windows 95 a la pobre 286. Un Windows que me copié, diskette por diskette (creo que eran 16) de lo de Nadav Rajzman. Claramente nunca funcionó. En ese entonces ni me imaginaba que era imposible que funcione.

Lo próximo que recuerdo son los primeros años de la secundaria. Ya estaba cansado de la 286, que para ese entonces era más que vieja, así que me iba caminando al colegio para ahorrar hasta el último centavo de la mensualidad que me daba mi vieja. Así logré comprarme la Pentium 233 (¡con MMX!). Y con esa pasé los años en el Acosta, donde conocí a mucha gente que poco tuvo que ver con mi afinidad por la computación, pero mucho tuvo que ver con convertirme en la persona que soy, con todo lo que eso implica. La banda de Bande (en especial Mavi, Chuli, Diego, Juan, Santiago, Fede. Y Hernán, amigo de toda la vida), con la que, entre cafés con leche con dos de azucar y cervezas de $1,60 perdimos el tiempo de la forma más maravillosa posible, entre amigos. Pepe y Manolo que siempre nos saludaban a la voz de "Doctor". Y pensar que algún día eso iba a tener un significado...

Después "-¿Qué vas a estudiar?". La pregunta más temida. "-Ingeniería en informática...", "-Ajá" y eso fue todo lo que dijo mi viejo. Tiempo después me dijo "¿Por qué no te fijás? Hay una carrera de computación, en la facultad de Exactas, creo que se llama Computador científico..." supongo que un arquitecto no podía permitir que su hijo fuera ingeniero. Para reforzar, en el CBC lo conocí a Damián, que estaba anotado para esa carrera de exactas en la que "cuando entrás, lo primero que te dicen es que lo del CBC no sirve para nada, y estudiás todo de nuevo, pero bien" ¡A mi juego me llamaron! Así llegué un día de Marzo de 2003, por primera vez, a Ciudad Universitaria.

Todavía con la simultaneidad a cuestas. Y no me fui más.

Muchas cosas y mucha gente. Imposible hacerle honor a todas y todos. Aparece Damián nuevamente, consiguiéndome mi primer trabajo en el área. Él y Gutes ("el de ORT, que viene en auto"). Mi primer grupo de TP. El BebeJugando. Qué de chorizos a la pomarola, ñoquis caseros y asaditos que nos mandamos esos fines de semana de estudio entre Flores y Floresta (sin chiste por favor). Y cuando no nos juntábamos, mate en la mesa con mis hermanas, Tute, Chancho y Chuletas. Después aparecen Mati, Román, Piter, Tommy, Guido. Estábamos cursando (si se puede decir) AlgoII y empezamos a juntarnos a discutir "cosas" de la facultad junto con Charly y Nico Maur con quienes después fundamos la AEI. La agrupación con la que conquistamos el CoDep de computación por vaaaaarios años consecutivos. Mati y Piter de nuevo, AlgoIII. Gran cursada. Hace su aparición el piquetero programador, el remise con el TP, el té con tortas, Marmol y las pastas ¿Se acuerdan cuando nos ofendimos porque nos aprobaron un TP que claramente NO estaba bien?

Después la carrera se me va a un segundo plano. Sobresale la militancia. Itai y Fede. Charly y Nico de nuevo. Leo. Con quienes construimos una locura. A contramano y sin permiso. Con la bandera del pelado, un "¿Por qué nacimos?¿Por qué luchamos?" diseñado entre gallos y medianoches, a conquistar los sueños de una juventud inconforme, crítica, rebelde. Qué distante y a la vez, qué fundamental.

Y en el medio de la vorágine militante, se me aparece Ali "esa, la que habló en la asamblea, la troska". Resultó que troska no era. Pero sí fue (y es) mi compañera. Cuántas tardes estudiando en lo de Graciela. Bah, ella estudiaba y mientras me decía a mí "dale, estudiá". Cuántas noches en vela por el centro de estudiantes. Cuántos cronogramas, ajustes de cronogramas y ajustes de los ajustes de cronogramas que me armaste. Cuánta compañía, cuánta banca, cuánta paciencia, cuánto empuje. Y cuántas otras cosas para las que sobran las palabras.

Me estoy olvidando de muchos. Mis compañeros docentes. Estoy convencido de que aprendí más siendo docente que siendo estudiante. Se me vienen los que me ayudaron en mis primeros pasos: Schapa, Charly (ya dejá de aparecer che!), DFS. Aquellos con los que nos cargamos materias al hombro, Mati, Piter, Nico Rosner. La banda que hizo patria en Sistemas, Pablete, Mariano.

Los pibes del FEM!, en especial Marce con quien siempre compartimos interminables discusiones sobre la ciencia, la política científica y la historia universitaria. Manu y Nico, Lau, Lipa y Nano, Mauro, la primera camada. La CoNEAU ¿se acuerdan de la CoNEAU?

Los fieles de la bondiola. Dani V., Fran, Facu, Mica, Fede, Marianito y tantos otros de los que me estoy olvidando. Interminables comidas bajo la incumplible consigna de "hoy no hablamos del DC".

Toda la gente que ocupa, física o espiritualmente, la oficina 8. Mariano,

Marcelo, Manu, Cani, Pau, Nico, Charly ¡Limpien el mate cuando se van!

Ya vengo llegando al final. Mis últimos recuerdos son para mis directores, Nico y Charly. Qué veranito ¡eh!

Al final, no sé en cuál de todos estos recuerdos está el punto de inflexión que me trajo hasta acá. Creo que fue un poco de todo. Lo que es seguro es que si uno de ustedes hubiera faltado, yo no habría sido el mismo, mi trabajo no habría sido el mismo. Por eso, a todos ustedes, muchas gracias. Más que muchas, infinitas.

Índice general

1.. Introducción 1
1.1. SAT Solving 4
1.2. SAT Solving paralelo y distribuido 5

2.. Preliminares 9
2.1. Lógica proposicional 9
2.2. El problema de determinar la satisfactiblidad de una fórmula proposicional 11
2.2.1. *SAT Solving* 12
2.2.2. Aprendizaje 14

3.. Un *sat-solver* paralelo y distribuido 19
3.1. Objetivos de diseño y desafíos asociados 20
3.1.1. Aspectos básicos de escalabilidad 20
3.1.2. Aspectos de automatización e interactividad 21
3.1.3. Otros aspectos de calidad 22
3.2. Arquitectura 23
3.2.1. *backend* 24
3.2.2. *frontend* 27
3.3. Sobre la automatización 29
3.3.1. Consideraciones sobre la estrategia 29
3.3.2. La estrategia implementada 33
3.4. Resultados experimentales 36
3.4.1. Casos de estudio 36
3.4.2. Resultados obtenidos 38
3.4.3. Discusión y conclusiones 40

4.. Aprendizaje en un *sat-solver* paralelo y distribuido 41
4.1. Justificación del enfoque 41
4.2. Prueba de concepto 44
4.3. Resultados experimentales 46
4.3.1. Criterios de selección de cláusulas 46
4.3.2. Resultados obtenidos 47

5.. Conclusiones y trabajo futuro 59
5.1. Conclusiones 59
5.2. Trabajo futuro 60

1. INTRODUCCIÓN

En la actualidad el *software* forma parte integral de nuestra vida. Este interviene en todo tipo de tareas. Desde el envío de un mensaje de texto o la realización de una llamada telefónica hasta el sistema de control de una central nuclear, pasando por el control de un ascensor o el sistema de velocidad crucero de un automóvil, utilizan sistemas basados en *hardware* y *software* informático. La variedad, alcance y **criticidad** de las responsabilidades asignadas a las piezas de *software* hacen de las mismas un componente de alto impacto en nuestra realidad cotidiana.

Por otra parte, la alta complejidad del *software* hace que la construcción del mismo sea una tarea propensa a errores. El impacto de una falla en un componente de *software* puede variar desde la imposibilidad de utilizar un artefacto doméstico hasta una catástrofe de magnitudes como la fundición del núcleo de un reactor nuclear.

Son estos aspectos los que motivan la necesidad creciente de construir métodos (formalismos, metodologías, herramientas, etc.) que permitan garantizar la calidad del *software* en un sentido general. La posibilidad de establecer la ausencia de errores en un programa radica en la capacidad de establecer inequívocamente que ese programa cumple determinadas propiedades que expresan el adecuamiento del mismo al comportamiento esperado. Por ejemplo si quisiéramos establecer el correcto funcionamiento del mecanismo de seguridad de un horno de microondas podríamos enunciar una propiedad como la siguiente: "No es posible encender el microondas si la puerta está abierta".

La **verificación formal** de *software* consiste en demostrar que un determinado sistema o algoritmo es correcto respecto a una **especificación formal** del mismo. Los lenguajes naturales son intrínsecamente ambiguos y por lo tanto una descripción de un sistema realizada en lenguaje natural resulta muy difícil de verificar. En lo subsiguiente asumiremos que una especificación será formal en tanto está descripta en un lenguaje que posee: a) reglas para determinar cuándo una sentencia pertenece al lenguaje (sintaxis), b) reglas que permitan interpretar las sentencias (bien formadas) de manera precisa y significativa (semántica), y c) reglas para inferir información útil a partir de la especificación (*proof theory*).

El enfoque de la verificación formal de software, tal vez uno de los más ambiciosos en el campo de la ingeniería de software, enfrenta dos problemas fundamentales. En primer lugar está la dificultad de construir una especificación formal que exprese de manera correcta el comportamiento esperado del sistema (a lo que llamaremos validación) ya que el mismo normalmente proviene de requerimientos expresados en lenguaje natural. El segundo

problema fundamental es que las lógicas subyacentes a la mayoría de los lenguajes formales capaces de expresar un conjunto lo suficientemente amplio e interesante de propiedades son normalmente indecidibles, es decir que no existe un algoritmo capaz de, para toda fórmula del lenguaje, establecer si la misma es un teorema o no.

El problema de la indecidibilidad se transforma en un escollo fundamental en tanto impide la construcción de métodos de verificación formal completamente automáticos. Diversos enfoques han sido desarrollados para atacar este problema [21] [26]. La mayoría de estos métodos consisten en restringir de alguna manera el lenguaje formal utilizado de modo de generar un sublenguaje del mismo que sea decidible. Uno de los enfoques desarrollados en este sentido es el conocido como *Bounded Model Checking* (verficación acotada de modelos).

Model Checking [17] es una técnica desarrollada para la verificación de sistemas reactivos. La técnica consiste en la exploración exhaustiva y automática del espacio de estados de un sistema modelado como una máquina de estados finita. Cuando la cantidad de estados del sistema es relativamente grande la exploración exhaustiva del espacio de estados se vuelve inpracticable. Esto motivó el desarrollo de la técnica conocida como *Symbolic Model Checking* [10, 24] que ataca el problema antedicho a partir de evitar construir el grafo de estados del sistema utilizando fórmulas lógicas para caracterizar conjuntos y relaciones. Cuando se trata de fórmulas proposicionales, en *Symbolic Model Checking* el chequeo de las fórmulas es realizada típicamente mediante el uso de BDDs (*Binary Decision Diagrams*) [9], una forma canónica de representar dichas fórmulas. A pesar de que las técnicas de *Symbolic Model Checking* basadas en BDDs son capaces de manejar sistemas con cientos de estados, el tamaño de los BDDs generados para sistemas grandes se vuelve prohibitivo.

El problema del espacio requerido por los BDDs motivó la creación de la técnica conocida como *Bounded Model Checking* [7]. La técnica presentada en [7] consiste en buscar contraejemplos de una longitud acotada k a partir de generar una fórmula proposicional que sea verdadera si y sólo si un contraejemplo de dicha longitud existe. Para verificar estas fórmulas los autores reemplazaron la utilización de BDDs por la utilización de procedimientos SAT ya que para esa época los mismos eran capaces de manejar fórmulas proposicionales con varios miles de variables y no sufrían del problema de la explosión en memoria. Si bien la técnica mencionada es correcta y completa para sistemas modelados como máquinas de estados finitos ya que el espacio de búsqueda, aunque usualmente sea exponencialmente grande, es finito, no es cierto que esta metodología aplicada a un lenguaje formal cualquiera sea siquiera correcta: tómese, por ejemplo, cualquier lenguaje lógico que no posea la propiedad de modelo finito. En tales casos existen conjuntos de fórmulas cuyos modelos son necesariamente infinitos; así, la exploración de un subespacio finito en busca de un modelo nunca podría garantizar una

respuesta negativa. Esta debilidad, en el caso general, generada por la introducción de la cota en el tamaño del contraejemplo es lo que, aplicado a un sistema lógico indecidible, permite obtener un subconjunto del lenguaje original que resulte decidible y, por lo tanto, automáticamente verificable, aun cuando las respuestas del procedimiento utilizado resulten parciales respecto de la validez de una propiedad determinada en el lenguaje original.

Un ejemplo típico de esto se da en lenguajes de primer orden donde la indecidibilidad se da a partir de la existencia de cuantificadores existenciales sobre dominios no acotados. En estos casos la cota en el tamaño del contraejemplo puede ser la resultante de establecer una cota a cada uno de los dominios que aparecen en la signatura. De este modo las cuantificaciones existenciales se convierten en disyunciones y por lo tanto el lenguaje se vuelve decidible (y en particular codificable como una fórmula proposicional susceptible de ser verificada automáticamente). Por ejemplo la fórmula:

$$(\exists x)\ (\alpha(x)) \tag{1.1}$$

a partir de la introducción de la cota sobre el dominio del cual toma valores x se transforma en la fórmula finita:

$$(\exists t \in \{t_1, \ldots, t_n\})\ (\alpha(t)) \tag{1.2}$$

o lo que es lo mismo:

$$\alpha(t_1) \vee \ldots \vee \alpha(t_n) \tag{1.3}$$

Cabe mencionar que esta restricción al tamaño de los contraejemplos introducida por forzar que la interpretación de los dominios sea sobre conjuntos finitos hace que el método de análisis no sea correcto ya que la ausencia de contraejemplos a una propiedad determinada dentro de ciertas cotas no garantiza la ausencia de contraejemplos para cotas de mayor tamaño. Por esta razón, *Bounded Model Checking* no puede ser considerada una técnica de verificación en el sentido estricto. Sin embargo la utilización de *Bounded Model Checking* permite ganar confianza en la propiedad que se quiere verificar antes de embarcarse en la costosa tarea de demostrarla formalmente. Algunas aplicaciones exitosas de *Bounded Model Checking* son la generación automática de casos de test [5], validación de código y especificaciones [18], etc.

Existen una variedad de herramientas formales que utilizan el enfoque de *Bounded Model Checking* basado en sat-solving que mencionamos en los párrafos anteriores. Una metodología ampliamente difundida y adoptada es Alloy [21]. Alloy es un lenguaje formal de primer orden diseñado para expresar las propiedades estructurales de un sistema; posee una sintaxis declarativa cercana a los lenguajes orientados a objetos y lo suficientemente poderosa como para expresar propiedades complejas y a la vez plausible de ser analizada de forma completamente automática (utilizando la herramienta homónima) a partir de una traducción como la que mencionamos en los párrafos precedentes.

1.1. SAT Solving

El interés en construir procedimientos sistemáticos y automáticos de demostración se remonta al interés que tuvieron diversos matemáticos en construir un método de esas características para la lógica de primer orden. Uno de los máximos exponentes fue David Hilbert, quien enunció que la búsqueda de un algoritmo de decisión para la lógica de primer orden era el problema central de la lógica matemática. La demostración de que un procedimiento de estas características no era posible elaborada por Church y Turing provocó una pérdida de interés en esta clase de problemas. Sin embargo la construcción de procedimientos, no ya de decisión, sino de demostración para la lógica de primer orden (entre los cuales se destaca el desarrollado por Martin Davis y Hilary Putnam [15]) sumada a los crecientes campos de aplicación, dentro de las ciencias de la computación, de lógicas decidibles o de fragmentos decidibles de lógicas no decidibles, dieron un nuevo impulso al desarrollo de estos métodos.

En particular la variedad de aplicaciones que fueron surgiendo para la lógica proposicional (*Planning* en inteligencia artificial, demostración automática de teoremas, *Model Checking*, verificación de *software*, etc.) impulsaron fuertemente el desarrollo de técnicas y procedimientos para determinar si una fórmula de la lógica proposicional podía ser verdadera. El establecimiento de este problema como el primer problema **NP-Complete** [12] no impidió el desarrollo de heurísticas que hoy permiten manejar fórmulas de cientos de miles de variables eficientemente.

Si bien existe una variedad de métodos para determinar si una fórmula proposicional puede ser verdadera, los más utilizados son aquellos derivados del algoritmo desarrollado por Davis, Longemann y Loveland (`DPLL`) en [14] que es un refinamiento del algoritmo presentado en [15]. Este procedimiento es, en esencia, un algoritmo de *backtracking*.

Tuvieron que pasar varios años para que Marques-Silva introdujera en [23] la primera optimización radical al algoritmo `DPLL` que dio origen a lo que hoy se conoce como procedimiento *Conflict Driven Clause Learning* (`CDCL`) cuya base es el aprendizaje de nuevas cláusulas ante cada conflicto. Ésto permite recortar el espacio de búsqueda de manera dramática. A partir de ese momento una serie de mejoras han sido desarrolladas para cada una de las partes críticas del algoritmo desarrollado por Marques-Silva[1].

Una de las partes fundamentales de los algoritmos de *sat-solving* es la conocida como *Boolean Constraint Propagation* (o simplemente propagación). Cuando una fórmula φ de la lógica proposicional es una conjunción, para que φ sea verdadera, cada una de sus partes deben serlo. Es por esto que si una de sus partes es una fórmula que contiene únicamente un literal, ese

[1] Para un relevamiento completo de las optimizaciones aplicadas al algoritmo `CDCL` ver [22]

literal debe necesariamente ser verdadero para que φ pueda serlo. Bajo esta observación se desarrolló el procedimiento de `BCP` que consiste precisamente en *propagar* una decisión (asignación de valor de verdad a un literal) observando qué partes de la fórmula quedan con un único literal libre a partir de esta decisión. Luego esos literales deben ser necesariamente asignados al valor de verdad `True` convirtiéndose en *implicaciones* de la decisión tomada.

A pesar de la enorme mejora que representó la incorporación de aprendizaje propuesta por Marques-Silva, con el correr del tiempo se estableció que los *sat-solvers* invierten la mayor parte de su tiempo (cerca del 90 %) en propagar las implicaciones que se siguen de cada una de las decisiones tomadas durante la búsqueda. Esta propagación incluye el recorrido del conjunto de cláusulas que conforman el problema cuya complejidad es, en mejor caso, lineal. De esta forma, la incorporación de nuevas cláusulas (el *leitmotiv* del procedimiento `CDCL`) a la vez que proporciona una ventaja al recortar el espacio de búsqueda representa una desventaja en tanto que incrementa la cantidad de cláusulas sobre las cuales es necesario realizar las propagaciones.

Diversas optimizaciones se han desarrollado para reducir el tiempo invertido en la propagación. Una de las más importantes es la presentada en [25] que permite reducir drásticamente el tiempo de propagación a partir de mantener una referencia a dos literales no asignados de cada cláusula del problema. De esta forma, una cláusula se vuelve (potencialmente) unitaria únicamente cuando uno de los dos literales *observados* es asignado. De otra manera la cláusula posee al menos dos literales libres y por lo tanto no se produce una implicación a partir de dicha cláusula.

Más allá de las optimizaciones presentadas, el tiempo de propagación sigue dominando el tiempo de procesamiento utilizado por los *sat-solvers* secuenciales. Esta situación, sumado al hecho de que la incorporación de nuevas cláusulas también representa un problema desde el punto de vista de la utilización de la memoria de la computadora, han llevado a la comunidad a establecer que, por lo general, es mejor tener un política agresiva de purga de la base de datos de cláusulas aprendidas [6]. Si bien se han desarrollado distintas heurísticas con muy buenos resultados, el problema de determinar cuántas y cuáles cláusulas descartar (o su equivalente, preservar) sigue siendo un problema ampliamente estudiado en la comunidad *sat*.

1.2. SAT Solving paralelo y distribuido

El establecimiento del problema *sat* como un problema **NP-Complete** hace pensar que no es posible obtener un algoritmo polinomial que resuelva dicho problema. Al mismo tiempo, en los últimos años, las mejoras a los algoritmos secuenciales de *sat-solving* han sido de carácter marginal y no parece haber avances que permitan esperar un salto en el orden de magnitud de los problemas resueltos por los *sat-solvers* secuenciales. Por otro lado, la rapidez en el incremento de la velocidad de las computadoras ha ido dis-

minuyendo (esencialmente debido al problema de la disipación del calor y el consumo de energía) y en su lugar se ha producido un incremento en la cantidad de unidades de procesamiento disponibles así como su abaratamiento. Estos hechos han impulsado a la comunidad *sat* a explorar la posibilidad de construir algortimos paralelos y/o distribuidos de *sat-solving* de modo de: a) sacar el máximo provecho de la nueva tendencia en construcción de computadoras y b) proveer un algoritmo **escalable** que permita aprovechar el abaratamiento del *hardware* .

La aparición de los primeros *sat-solvers* paralelos y distribuidos [11,32] es contemporánea a la aparición del algoritmo `CDCL`. Por lo tanto los primeros *sat-solvers* paralelos fueron desarrollados sobre la base del algoritmo `DPLL` y es por esta razón que estos *sat-solvers* no incorporaban aprendizaje de nuevas cláusulas.

Una de las razones que dieron origen a implementaciones tempranas de algoritmos paralelos para el problema *sat* es que este problema presenta una forma intuitiva y fácil de ser dividido. La idea consiste en seleccionar un subconjunto pequeño de variables del problema, digamos n, y generar las 2^n asignaciones posibles de valores de verdad a este subconjunto. Cada una de estas asignaciones es una valuación parcial del problema original y por lo tanto lo que obtenemos son 2^n subproblemas **disjuntos** que pueden ser resueltos por distintas unidades de procesamiento. Cada unidad de procesamiento puede utilizar entonces alguna implementación de un algoritmo de *sat-solving* secuencial para resolver el subproblema que le haya tocado. Esta técnica, llamada *guiding path* [32], introduce una pregunta fundamental: ¿De qué modo determinar un *guiding path* efectivo? Esto es, cómo construir un *guiding path* que distribuya equitativamente la carga de trabajo entre las distintas unidades de procesamiento.

El enfoque anterior de división de un problema *sat* es adecuado para un escenario con unidades de procesamiento que se comunican a través de una red, como pude ser un *cluster* o en computación *grid*. La proliferación de computadoras con múltiples procesadores y múltiples núcleos impulsó la investigación sobre formas de paralelización de *sat-solvers* en un escenario donde las múltiples unidades de procesamiento tienen acceso uniforme a una memoria común. Este nuevo escenario permitió el desarrollo de una forma radicalmente distinta de paralelización de un problema *sat*. La misma consiste escencialmente en lo que se conoce como *portfolio*. La idea se fundamenta en la observación de que distintos *sat-solvers* secuenciales, o un mismo *sat-solver* con distintos parámetros, pueden resolver un mismo problema en tiempos sumamente diferentes. Sobre esta base se desarrollaron *sat-solvers* paralelos que en lugar de dividir el problema a resolver, ejecutan distintos *sat-solvers* sobre el mismo problema (o un mismo *sat-solver* con distintos parámetros [20]) y comparten las cláusulas aprendidas entre las distintas instancias. Es importante notar que así como en los *sat-solvers* secuenciales, la incorporación de aprendizaje es un factor fundamental en la

mejora del rendimiento, en el contexto de *sat-solving* paralelo con el enfoque *portfolio* también lo es. Esto se debe a que sin aprendizaje compartido entre las distintas instancias el resultado del enfoque *portfolio* sería, en el mejor caso, tan bueno como el mejor de los *sat-solvers* utilizados. Por el contrario, la incorporación de aprendizaje compartido permite potencialmente obtener mejores resultados que el del mejor *sat-solver* utilizado. La información experimental ha confirmado esta hipótesis.

Por supuesto, distintas implementaciones concretas han sido desarrolladas sobre esta idea, e incluso implementaciones híbridas que combinan las dos formas de paralelización antedichas [20] [28]. Sin embargo, a lo largo de los años, ha tomado un impulso mucho mayor el enfoque basado en memoria compartida. Esto puede verse por un lado en la disponibilidad de herramientas. Hoy en día existen por lo menos una decena de *sat-solvers* paralelos para procesadores multi*core* que son activamente desarrollados y que compiten año a año en las competencias internacionales de *sat-solving*. Por otro lado el sólo hecho de que las competencias internacionales de *sat-solving* hayan incorporado categorías específicas para *solvers* paralelos (por lo menos desde el año 2008) es también un síntoma de la alta actividad que se viene desarrollando en este área. Por el contrario, el interés en los *sat-solvers* distribuidos no ha obtenido la misma atención por parte de la comunidad. De hecho, la mayoría de los *sat-solvers* distribuidos han sido abandonados. Es sumamente difícil encontrar herramientes de este campo disponibles para ser utilizadas, y las que existen han quedado obsoletas.

Es importante destacar que el enfoque *portfolio* ha mostrado resultados prometedores en la resolución más eficiente de problemas basados en *sat* y los datos de estos avances no sólo están documentados en numerosas publicaciones que lo avalan, sino también en el surgimiento y crecimiento de la comunidad que compromete sus esfuerzos en esta nueva área de investigación. [1] Es nuestra opinión que este enfoque no da respuesta al hecho de que hoy la tendencia indica que la escalabilidad proviene del trabajo cooperativo y distribuido de unidades independientes de cómputo.

El objetivo de esta tesis es la de dar respuesta a la construcción de un *sat-solver* paralelo y distribuido basado en la técnica *guiding path* que: '

- realice una adecuación y balanceo de la carga de procesamiento en *run-time* sobre una cantidad y configuración de equipamiento desconocida,

- realice una adecuación y balanceo del uso de memoria primaria y secundaria en *run-time*, también sobre una cantidad y configuración de equipamiento desconocida, y

- sea extensible y fácil de parametrizar en todos aquellos aspectos que determinen el comportamiento del procedimiento de resolución del problema.

Al mismo tiempo, someteremos a prueba la implementación de un método de aprendizaje distribuido utilizando distintos criterios de aprendizaje con el objetivo de identificar si esta técnica que ha demostrado ser revolucionaria en la optimización de los *sat-solvers* secuenciales aporta alguna mejora en este nuevo escenario. Este segundo objetivo encuentra su motivación en la hipótesis de que al distribuir el trabajo entre distintas unidades de cómputo se abre la posibilidad de que se produzca una enorme cantidad de retrabajo que podría ser sustancialmente minimizado mediante la incorporación de cláusulas aprendidas por otras instancias. Al mismo tiempo, la comunicación de cláusulas aprendidas entre distintas unidades de procesamiento conlleva un *overhead* intrínseco que podría anular los efectos beneficiosos producidos por la disminución del retrabajo. Esto último, sumado al hecho de que el incremento en los tamaños de las bases de datos de cláusulas aprendidas de cada instancia ralentiza la propagación de implicaciones nos pone en una situación en la que es importante encontrar un equilibrio entre la ganancia obtenida por una parte y la pérdida producida por el *overhead*. A su vez, la incorporación de aprendizaje en un ambiente distribuido presenta desafíos propios del entorno como ser: lidiar con la velocidad de acceso a recursos que se encuentran en equipos remotos (órdenes de magnitud más lento que el acceso a memoria principal), la necesidad de tener en cuenta la contención para evitar la saturación de un equipo, etc. Por lo tanto esta incorporación se diseñará e implementará sin perjuicio de las características mencionadas anteriormente que creemos centrales en el desarrollo de una herramienta que pretende ejecutar en ambientes paralelos y distribuidos que *a priori* no son conocidos.

La presente tesis se divide de la siguiente forma. El Cap. 2 presenta las definiciones y resultados básicos del área de *sat*, en conjunto con algunas valoraciones del autor, con el objeto de fijar notación y establecer un piso de contenidos que permitan el desarrollo de los aspectos novedosos contenidos en esta tesis. El Cap. 3 presentaremos la arquitectura y los aspectos relevantes de la implementación de un *sat-solver* paralelo y distribuido basado en la técnica *guiding path*. El Cap. 4 exhibe el diseño e implementación de un mecanismo de aprendizaje para el *sat-solver* presentado en el Cap. 3 en conjunto con una prueba de concepto de la técnica, el desarrollo de los experimentos y la sistematización de resultados. Para finalizar, en el Cap. 5, presentaremos las conclusiones derivadas del desarrollo de la herramienta y los experimentos sobre ella realizados, y las líneas de investigación que se abren a partir de esto.

2. PRELIMINARES

En el presente capítulo presentaremos las definiciones y resultados preliminares comprendiendo desde las definiciones lógicas a fin de fijar notación y terminología, hasta el estado del arte en lo que refiere a la resolución del problema *sat*.

2.1. Lógica proposicional

La lógica proposicional es una formalización matemática que se ocupa del estudio exhaustivo de los valores de verdad y su combinación a partir de los operadores conocidos como booleanos. Estos operadores usualmente son la negación ($\neg$), la conjunción ($\wedge$), la disyunción ($\vee$), la implicación ($\Rightarrow$), etc. Usualmente se suelen tomar como conjunto minimal capaz de representar a todos los operadores binarios la negación ($\neg$) y la disyunción ($\vee$) puesto que estos son un conjunto apropiado que permite expresar a los restantes, pero por comodidad a la hora de presentar el problema a resolver utilizaremos la negación ($\neg$), la conjunción ($\wedge$) y la disyunción ($\vee$).

La formalización más utilizada, desde un punto de vista matemático, de la lógica proposicional es la desarrollada por el matemático inglés George Boole [8] y conocida como álgebra de Boole, pero a efectos de esta presentación y por comodidad nos concentraremos en su presentación lógica.

Definición 2.1.1. ***[Fórmulas bien formadas]*** Sea $\mathcal{V}$ un conjunto infinito numerable de variables, entonces $Form(\mathcal{V})$, el conjunto de fórmulas bien formadas sobre $\mathcal{V}$, es el menor conjunto F tal que:

- $\mathcal{V} \cup \{\bot, \top\} \subseteq F$, y
- para todo $f, f' \in F$, $\{\neg f, f \wedge f', f \vee f'\} \subseteq F$.

Definición 2.1.2. ***[Soporte]*** Sea $\mathcal{V}$ un conjunto infinito numerable de variables, se define $sop : Form(\mathcal{V}) \to 2^{\mathcal{V}}$, el soporte de una fórmula proposicional de la siguiente forma:

- $sop(\bot) = \emptyset$,
- $sop(\top) = \emptyset$,
- $sop(p) = \{p\}$,
- $sop(\neg\varphi) = sop(\varphi)$,
- $sop(\varphi \wedge \psi) = sop(\varphi) \cup sop(\psi)$, y

- $sop(\varphi \vee \psi) = sop(\varphi) \cup sop(\psi)$.

A continuación, en las siguientes dos definiciones, se presenta la forma en la que la sintaxis presentada anteriormente se interpreta en el conjunto $\{0, 1\}$.

Definición 2.1.3. ***[Valuación]*** Sea $\mathcal{V}$ un conjunto infinito numerable de variables, una valuación de las variables proposicionales es una función total $v : \mathcal{V} \to \{0, 1\}$.

Definición 2.1.4. ***[Valuación de términos]*** Sea $\mathcal{V}$ un conjunto infinito numerable de variables y una valuación de las variables v, $p \in \mathcal{V}$ y $f, f' \in Form(\mathcal{V})$, se define $v^* : Form(\mathcal{V}) \to \{0, 1\}$

- $v^*(\bot) = 0$,
- $v^*(\top) = 1$,
- $v^*(p) = v(p)$,
- $v^*(\neg f) = \begin{cases} 0 & \text{, si } v^*(f)) = 1 \\ 1 & \text{, si } v^*(f)) = 0 \end{cases}$,
- $v^*(f \wedge f') = \begin{cases} 0 & \text{, si } v^*(f) = 0 \text{ ó } v^*(f') = 0 \\ 1 & \text{, si } v^*(f) = 1 \text{ y } v^*(f') = 1. \end{cases}$, y
- $v^*(f \vee f') = \begin{cases} 0 & \text{, si } v^*(f) = 0 \text{ y } v^*(f') = 0 \\ 1 & \text{, si } v^*(f) = 1 \text{ ó } v^*(f') = 1. \end{cases}$.

Definición 2.1.5. *[$\models$]* Sea $\mathcal{V}$ un conjunto infinito numerable de variables, v una valuación de las variables proposicionales y $f \in Form(\mathcal{V})$, entonces $v \models_{\mathcal{V}} f$ si y sólo si $v^*(f) = 1$.

Definición 2.1.6. Sea $\mathcal{V}$ un conjunto infinito numerable de variables y $\varphi \in Form(\mathcal{V})$, φ se dice:

- satisfacible si y sólo si existe una valuación v tal que $v \models_{\mathcal{V}} \varphi$,
- tautología si y sólo para toda valuación v $v \models_{\mathcal{V}} \varphi$, y
- contradicción (o insatisfacible) si y sólo para toda valuación v $v \not\models_{\mathcal{V}} \varphi$.

Definición 2.1.7. ***[Restricción]*** Sea $\mathcal{V}$ un conjunto infinito numerable de variables, $S \subseteq \mathcal{V}$ y v una valuación, se define la restricción de v a S, $v|_S : S \to \{0, 1\}$ de la siguiente forma, para toda $p \in S$, $v|_S(p) = v(p)$.

Observación 2.1.1. *Sea $\mathcal{V}$ un conjunto infinito numerable de variables, $S \subseteq \mathcal{V}$, $\varphi \in Form(\mathcal{V})$ y, v y v' valuaciones, si $sop(\varphi) \subseteq S$ y $v|_S = v'|_S$, entonces $v \models_{\mathcal{V}} \varphi$ si y sólo si $v' \models_{\mathcal{V}} \varphi$.*

A partir de la observación anterior podemos definir la relación de satisfacción $\models_{\mathcal{V}}^{fin}$ en función de la ya presentada relación de satisfacción $\models_{\mathcal{V}}$ de la siguiente forma:

Definición 2.1.8. Sea $\mathcal{V}$ un conjunto infinito numerable de variables, $S \subseteq \mathcal{V}$ tal que $sop(\varphi) \subseteq S$, $\varphi \in Form(\mathcal{V})$ y v una valuación, $v|_S \models_{\mathcal{V}}^{fin} \varphi$ si y solo si $v \models_{\mathcal{V}} \varphi$.

La definición anterior expone un hecho de importancia mayor ya que en la práctica, para determinar la satisfactibilidad de una fórmula proposicional sólo es necesario generar la porción finita de la valuación cuyo dominio está restringido al soporte de dicha fórmula. En adelante, usaremos la palabra valuación refiriéndonos a esta porción finita de una valuación.

Definición 2.1.9. ***[Literal]*** Se denomina literal a una fórmula proposicional que es o bien de la forma p o bien de la forma $\neg p$ con p una variable proposicional.

Definición 2.1.10. ***[Forma Normal Conjuntiva]*** Una fórmula proposicional φ se encuentra en Forma Normal Conjuntiva (`CNF` por sus siglas en inglés) si la misma es una conjunción de cláusulas, donde cada cláusula es una disyunción de literales.

Por ejemplo, la fórmula $\phi = (p \vee q) \wedge (\neg p \vee q)$ se encuentra en `CNF` y posee dos cláusulas $C_1 = p \vee q$ y $C_2 = \neg p \vee q$. Dado que los conectores lógicos están implícitos cuando una fórmula se encuentra en `CNF` normalmente escribimos $\phi = \{[p, q], [\neg p, q]\}$ de modo equivalente.

Definición 2.1.11. ***[Fórmulas equivalentes]*** Dado $\mathcal{V}$ un conjunto de variables proposicionales y $\varphi_1, \varphi_2 \in Form(\mathcal{V})$, φ_1 y φ_2 son equivalentes (denotado como $\varphi_1 \equiv \varphi_2$) si y solo si para toda valuación v, $v \models \varphi_1$ si y solo si $v \models \varphi_2$.

Observación 2.1.2. *Dado $\mathcal{V}$ un conjunto de variables proposicionales y $\varphi \in Form(\mathcal{V})$, existe $\psi \in Form(\mathcal{V})$ tal que $\varphi \equiv \psi$ y ψ se encuentra en `CNF`.*

Más aun, existe $F^{cnf} : Form(\mathcal{V}) \to Form(\mathcal{V})$ tal que para toda $\varphi \in Form(\mathcal{V})$, $F^{cnf}(\varphi)$ está en `CNF`. [29]

2.2. El problema de determinar la satisfactiblidad de una fórmula proposicional

Como se desprende de lo dicho anteriormente, el problema de determinar la satisfactibilidad de una fórmula proposicional φ (también denominado *SAT problem*) consiste en determinar si existe alguna valuación v tal que $v^*_{sop(\varphi)}(\varphi) = 1$. En caso de que dicha asignación exista decimos que ϕ es satisfacible (o simplemente *sat*) y la asignación o valuación (usualmente

expresada como $\{p_1 \mapsto v_1, \ldots, p_n \mapsto v_n\}$ donde correspondiente determina un **modelo** de ϕ. Es importante notar que en caso de que una fórmula sea *sat* podría existir más de una valuación que haga que la fórmula sea verdadera. Si no existe ninguna valuación que haga verdadera a ϕ decimos que ϕ es insatisfacible (o *unsat*).

El problema de determinar si una fórmula proposicional dada es *sat* o *unsat* pertenece a la clase de problemas **NP-Complete** [12]. Es decir que no se conoce un algoritmo cuya complejidad temporal sea polinomial que pueda determinar si una fórmula es *sat* o *unsat*. La importancia que reviste el problema SAT para diversas áreas de las ciencias de la computación, como ser el diseño de circuitos o la verificación automática, ha impulsado el desarrollo de algoritmos relativamente eficientes conocidos comúnmente como *SAT solvers*.

En vista de la Obs. 2.1.2 el problema *sat* se define equivalentemente como el problema de determinar si una fórmula en `CNF` es satisfacible o no.

2.2.1. *SAT Solving*

Se conoce con el nombre de *SAT solvers* a los algoritmos que permiten determinar si una fórmula ϕ de la lógica proposicional tiene alguna valuación que la haga verdadera. Esto se lleva a cabo a partir de construir dicha valuación. El algoritmo general de *SAT solving* consiste en un algoritmo de *backtracking* que podríamos esquematizar como sigue:

Listing 2.1: Algoritmo *naïf*

```
NAIF–SAT (F ∈ Form(V) in CNF)
    if F is a consistent set of literals
       then return true
    if F contains an empty clause
       then return false
    v ← unassigned–var–from(F)
    G ← assign(v, 1, F)
    if NAIF–SAT(G)
       then return true
    else
       G ← assign(v, 0, F)
       return NAIF–SAT(G)
```

La mayoría de los *SAT solvers* son variaciones del algoritmo general DPLL [14] (que a su vez se basa en el presentado por Davis y Putnam en [15]) que presentaremos a continuación, y actúan sobre fórmulas proposicionales expresadas en Forma Normal Conjuntiva (`CNF`).

Definición 2.2.1. ***[Cláusula unitaria]*** Una cláusula se llama **unitaria** si posee un único literal.

Por ejemplo en la fórmula $\gamma = \{[-p], [q, r]\}$ la clásula $[-p]$ es unitaria. Cuando una fórmula CNF posee una cláusula unitaria, el único modo de lograr que la fórmula completa sea verdadera es asignando el valor de verdad 1 al único literal dentro de la cláusula unitaria.

Listing 2.2: Algoritmo DPLL

```
1 DPLL (F ∈ Form(V) in CNF)
2     if F is a consistent set of literals
3         then return true
4     if F contains an empty clause
5         then return false
6     for every unit clause l in F
7         F ← unit-propagate(l, F)
8     l ← choose-literal(F);
9     return DPLL(F ∧ l) or DPLL(F ∧ ¬ l)
```

El algoritmo presentado en List. 2.2, comúnmente conocido como *DPLL*, es un procedimiento que permite implementar un *sat-solver* cuyo comportamiento resulta radicalmente más eficiente que el presentado en List. 2.1. Esta modificación en la eficiencia del procedimiento proviene del tratamiento que se hace de las cláusulas unitarias y cómo la asignación de valores a estas repercute sobre la fórmula que se está analizando.

Los puntos claves del algoritmo son los que aparecen en las filas identificadas como 2, 4, 7, 8 y 9. En el caso de 2 y 4 porque son los puntos en los que se determina si la fórmula es satisfacible o no. En el caso del condicional que aparece en la fila 2 la fórmula se determina como satisfacible por determinar que ya no restan decisiones para ser tomadas; si F es un conjunto consistente de literales, entonces éste conforma una valuación de las variables proposicionales que satisface la fórmula puesto que si $p \in F$ entonces la valuación construida asigna 1 a p, y si $\neg p \in F$ entonces le asigna 0. En el caso de 4, si una cláusula resulta vacía, esto implica que no habrá forma de asignar ningún valor a ninguna variable proposicional de forma de hacerla verdadera y por lo tanto la fórmula no será satisfacible. Las filas 6 y 7 realizan lo que se denomina *unit propagation* o BCP (por el inglés *Boolean Constraint Propagation*): propagar en todas las cláusulas las decisiones ya tomadas, de forma que si un literal aparece en F, se eliminen todas las cláusulas que lo contengan, y se elimine toda aparición de dicho literal negado de cualquier cláusula en que aparezca mencionado. En la fila 8 se lleva a cabo una de las tareas más importantes del procedimiento: determinar cuál será la próxima decisión a tomar a partir de escoger un literal. El orden en que se eligen las variables modifica drásticamente el rendimiento de la herramienta, puesto que así se decide qué fragmento del espacio de búsqueda será explorado primero. Una vez elegido el literal que se utilizará para partir el espacio de búsqueda se aplica el procedimiento recursivamente sobre los dos espacios

disjuntos de soluciones determinados por l y por $\neg l$.

La mayor diferencia entre el algoritmo del List. 2.2 y el de List. 2.1 es precisamente la introducción de la técnica de BCP. Esta técnica permite aplicar un conjunto de decisiones ya tomadas simplificando la fórmula y reduciendo así el espacio de búsqueda a recorrer. Es sabido [25] que los *sat-solver* modernos que implementan esta técnica de propagación invierten aproximadamente el 90 % del tiempo en la realización de esta tarea, y es por esta razón que las partes más críticas de la implementación de un *sat-solver* son aquellas que están involucradas en la resolución de BCP. Ejemplos de esto son las técnicas, algoritmos y estructuras de datos para representación de cláusulas especialmente concebidas para optimizar BCP, como ser el caso de *Head/Tail* [31] y, más recientemente, los esquemas basados en *2-watched literals* [25].

2.2.2. Aprendizaje

Uno de los factores más determinantes en la mejora del rendimiento de los *sat-solver* modernos ha sido la incorporación de técnicas de aprendizaje de cláusulas derivadas de los conflictos que hacen que bajo una cierta cadena de decisiones la fórmula no sea satisfacible. La idea general es identificar contradicciones inherentes a un cierto conjunto de decisiones y agregar a la fórmula cláusulas que obliguen al *sat-solver* a descartar todo fragmento del espacio de búsqueda que de una u otra forma las implique.

Definición 2.2.2. ***[Cláusulas del problema]*** Por *cláusula del problema* nos referimos a una de las cláusulas que forman parte del problema original, es decir alguna de las contenidas en la fórmula CNF provista por el usuario.

Definición 2.2.3. ***[Cláusulas aprendidas]*** Llamaremos *cláusula aprendida* a una cláusula que pueda obtenerse por medio de la aplicación de una cantidad finita de pasos de resolución [27] sobre las cláusulas del problema, y/o sobre cláusulas aprendidas anteriormente.

Proposición 2.2.1. *Sea F un problema en* `CNF` *y c una cláusula aprendida a partir de F. Para toda valuación v, $v \models F$ implica $v \models c$.*

Demostración. Se sigue trivialmente de la definición de resolución para la lógica proposicional. □

Proposición 2.2.2. *Sea F un problema en* `CNF`*, C un conjunto finito de cláusulas aprendidas a partir de F. Para toda valuación v, $v \models F$ implica $v \models \bigwedge C$.*

Demostración. Por inducción en el cardinal de C y usando Prop. 2.2.1. □

Proposición 2.2.3. *Sea F un problema en* `CNF`*, C un conjunto finito de cláusulas aprendidas a partir de F. Para toda valuación v, $v \models F$ si y sólo si $v \models F \wedge \bigwedge C$.*

Demostración. La demostración se sigue separando en casos dependiendo de si una valuación dada v satisface o no F. Supongamos que $v \models F$, por Prop. 2.2.2, $v \models \bigwedge C$, y consecuentemente, $v \models F \wedge \bigwedge C$. Si $v \not\models F$, luego $v \not\models F \wedge \bigwedge C$, por monotonía de la lógica proposicional. □

Observación 2.2.1. *Al adosar una o más cláusulas aprendidas a las del problema no se pierde correctitud ni completitud: el nuevo problema y el anterior son equisatisfacibles. Sin embargo, la presencia de estas cláusulas adicionales sí agrega información en el sentido de orientación de la búsqueda (posiblemente permitiendo detectar futuros conflictos más rapidamente).*

Definición 2.2.4. ***[Nivel de decisión]*** Por *nivel de decisión* debe entenderse la cantidad de variables respecto de las cuales se han tomado decisiones arbitrarias (que posiblemente requieran *backtracking*) como parte del proceso de búsqueda. Cada vez que se toma una tal decisión (y se propagan, mediante BCP, todas las implicaciones de la misma y de las anteriores) se está incrementando en 1 el nivel de decisión actual del *sat-solver*.

Análisis de conflictos, UIPs y backjumping

Definición 2.2.5. ***[Conflicto]*** Llamaremos *conflicto* a toda situación en la que el *sat-solver*, como consecuencia de las decisiones tomadas, derive tanto un literal como su negación.

Cada vez que la cadena de decisiones tomadas –más las implicaciones obtenidas por medio de BCP tras cada una de ellas– conducen al *solver* a un conflicto, se lleva a cabo un *análisis de conflicto*, cuyos objetivos incluyen:

- Determinar las causas del conflicto hallado, y caracterizarlas mediante una o más *cláusulas de conflicto* que pueda(n) almacenarse para procurar evitar otros conflictos similares en lo sucesivo.
- Minimizar esas cláusulas hasta lograr que la caracterización antedicha sea lo más ajustada posible, para así maximizar el potencial de poda del espacio de búsqueda que en lo sucesivo pudiere permitir cada cláusula aprendida.
- Calcular el punto más alto al que se pueda regresar de modo correcto y completo en la cadena de decisiones (es decir, cuánto restar al nivel de decisión actual), evitando por completo tener que inspeccionar ciertas partes del espacio de búsqueda que, a la luz del conflicto hallado, sólo podrían ser infructuosas. Esto se conoce como *backjumping* o *backtracking no cronológico* [23, 25].

En la actualidad, para aprovechar la estructura del grafo de implicaciones (la clausura recursiva de las razones del conflicto, expresadas en términos

de cláusulas, decisiones tomadas y consecuencias implicadas), el análisis de conflictos de la mayor parte de los *sat-solvers* CDCL suele basarse en la noción de *Unit Implication Point* [23] y en la más reciente de *First UIP* [33].

Recorte de la base de datos

La acumulación de cláusulas aprendidas en conflictos pasados resulta muy ventajosa, pero no puede más que convertirse en una desventaja si se la implementa de manera irrestricta. En primer lugar, la cantidad de cláusulas aprendidas es proporcional a la de conflictos hallados, que a su vez es –en peor caso– exponencial en la cantidad de variables. Así, un solver que aprenda pero nunca descarte cláusulas correría el riesgo de agotar la memoria disponible.

En segundo lugar, la proliferación de cláusulas aprendidas conlleva cierto *overhead* en otros aspectos del proceso de búsqueda (y en particular en el corazón del mismo, ya que realizar BCP sobre un mayor número de cláusulas resulta más costoso). En consecuencia, al acumular más y más cláusulas también se corre el riesgo de alcanzar el punto en el que los beneficios de recortar el espacio de búsqueda ya no compensen la penalidad que supone un costo de propagación mucho mayor.

Estos riesgos motivan la necesidad de contar con políticas de purga: todo *sat-solver* que mantenga una base de datos de cláusulas aprendidas debe implementar algún criterio de borrado parcial periódico.

Surge así la necesidad de decidir cuáles y cuántas de las cláusulas aprendidas descartar, y cuándo o cada cuánto hacerlo. En este sentido se han estudiado, en las últimas dos décadas, diversos enfoques y criterios.

Longitud, actividad y LBD

Un primer factor que naturalmente se tuvo en cuenta es la longitud de las cláusulas aprendidas, en tanto suele ser el caso [23] que las más breves tienen mayor poder de recorte del espacio de búsqueda.

Otra propuesta, que por sus buenos resultados fue adoptada por muchas implementaciones desde entonces [19], consiste en medir el grado o nivel de *actividad* de las cláusulas aprendidas, observando y contabilizando la cantidad de conflictos futuros en los que cada tal cláusula se ve involucrada.

La mayoría de los *sat-solvers* CDCL modernos implementan esta métrica mediante alguna variante sencilla del enfoque AIMD (*additive increase, multiplicative decrease*): por lo general se incrementa en alguna constante fija el puntaje de actividad de cada cláusula responsable de un nuevo conflicto, y periódicamente se dividen los puntajes de actividad de todas las cláusulas por una constante fija [16].

Más recientemente se ha destacado otra métrica denominada LBD, por *Literals Blocks Distance* [6].Dados una cláusula c y una partición de sus lite-

rales en n subconjuntos acorde con la valuación parcial actual, de modo tal que los literales son particionados según sus respectivos niveles de decisión, decimos que el LBD de c es exactamente n.

3. UN *sat-solver* PARALELO Y DISTRIBUIDO

La implementación de nuestra herramienta como un sistema distribuido persigue el objetivo fundamental de mejorar el tiempo de ejecución ya sea decrementando el tiempo necesario para resolver un problema determinado, o transformando un problema previamente irresoluble (en tiempo razonable) en resoluble. Para ello el sistema debe ser **escalable** en tanto que debe tener la capacidad de sacar provecho de la incorporación una mayor cantidad de *hardware* a la infraestructura utilizada en la resolución del problema. Por otro lado, el uso de este *hardware* debe ser **eficiente** en el sentido de que la utilización de mayor cantidad de *hardware* debe resultar en un mejora de la performance del sistema (en nuestro caso entendida como se describió previamente).

Con estos dos objetivos en mente desarrollamos una *sat-solver* distribuido basado en la idea de *guiding path* y orientado a que su utilización sea en *clusters* de computadoras. La idea de *guiding path* se basa en la observación de que, en el peor caso, todas las posibles valuaciones de una fórmula proposicional φ deben ser evaluadas. Una vez que una valuación que satisface a una fórmula es encontrada, el procedimiento de *sat-solving* puede darse por finalizado[1]. Por el contrario, si la fórmula es *unsat*, la única alternativa para poder asegurarlo es agotar todas las posibles valuaciones y determinar que ninguna de ellas satisface a la fórmula en cuestión. Por lo tanto, una forma de dividir el problema es tomar una variable v y construir los dos subproblemas $\varphi_{v\leftarrow 0}$ y $\varphi_{v\leftarrow 1}$ que se derivan de las dos posibles asignaciones de valores de verdad a dicha variable. Una vez hecho esto, determinar que el problema original φ es *unsat* se reduce a determinar que tanto $\varphi_{v\leftarrow 0}$ como $\varphi_{v\leftarrow 1}$ son *unsat*. Por el contrario, determinar que el problema es *sat* reduciría a establecer que $\varphi_{v\leftarrow 0}$ es *sat* o bien

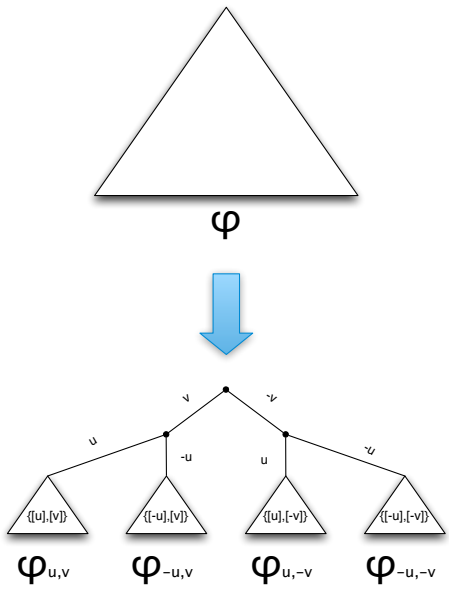


Fig. 3.1: Esquema de partición por *guiding path*

[1] Existe otra variante en la que, además de determinar si una fórmula es *sat* o *unsat*, interesa enumerar todos los posibles modelos de dicha fórmula. En este escenario el procedimiento de *sat-solving* no puede ser detenido una vez que se encuentra una valuación sino que el mismo debe continuar hasta agotar todas las posibles valuaciones que satisfacen a la fórmula en cuestión.

que $\varphi_{v\leftarrow 1}$ lo es. Ahora, $\varphi_{v\leftarrow 0}$ y $\varphi_{v\leftarrow 1}$ pueden ser considerados como dos problemas independientes y volver a aplicar la misma idea recursivamente. Naturalmente este proceso puede realizarse seleccionando más de una variable en cada paso. Al seleccionar n variables obtendremos entonces 2^n subproblemas. Llamaremos *levantar variables* al proceso de dividir un problema φ en los 2^n subproblemas que resultan de aplicar todas las posibles valuaciones sobre el conjunto de n variables levantadas. Se desprende de esto que una de las tareas que debe poder realizar la herramienta es la de dividir un problema en los subproblemas que derivan de adoptar todos los posible *guiding path* para un conjunto de variables.

3.1. Objetivos de diseño y desafíos asociados

3.1.1. Aspectos básicos de escalabilidad

En primer término consideraremos los desafíos asociados con ciertos aspectos básicos que, de no ser adecuadamente manejados, podrían atentar de modo flagrante contra la escalabilidad de la herramienta.

Almacenamiento distribuido

La necesidad de que el almacenamiento de problemas pendientes de resolución se encuentre distribuido se debe a un doble aspecto. Por un lado, la cantidad de subproblemas producidos puede ser muy grande. Por lo tanto no es razonable requerir que la cantidad de espacio necesario para almacenar todas las tareas pendientes de resolución se encuentre disponible en una misma ubicación.

Aún si fuera posible contar con la cantidad de espacio necesaria para almacenar todos los subproblemas en una ubicación centralizada, surge un segundo problema que presentaría este enfoque: el de la contención de acceso a la información asociada a un problema. Si pretendemos que el almacenamiento no se vuelva un cuello de botella, es vital distribuir las tareas pendientes de modo que cuando los *workers* requieran nuevas tareas para resolver, los múltiples pedidos no recaigan siempre sobre un mismo equipo.

Simetría en las capacidades de los *workers*

El requisito de que los nodos de trabajo (*workers*) sean simétricos también se desprende de la necesidad de eliminar los potenciales cuellos de botella. La simetría, entendida en su máxima expresión como la posibilidad de que todas las unidades de procesamiento puedan realizar todas las funciones necesarias, provee la capacidad de distribuir la carga de trabajo de la manera más conveniente en cada momento.

Esto no podría lograrse si los nodos de trabajo tuvieran funciones específicas, ya que podría darse el caso de tener nodos ociosos por falta de

trabajo pendiente de la clase de trabajo que esos nodos realizan, a la vez que otros nodos se encuentran sobrecargados. Esta situación podría empeorar seriamente a medida que la cantidad de nodos aumenta.

En nuestro caso particular, el requisito de simetría se traduce en que todos los *workers* deben poseer las capacidades de: a) analizar un subproblema hasta lograr obtener un veredicto (consumir), b) dividir un subproblema demasiado difícil en nuevos subproblemas (producir) y c) almacenar, solicitar y transferir subproblemas (distribuir).

Movilidad de tareas pendientes

El triple rol de productor, consumidor y distribuidor asignado a los *workers* transforma la migración de tareas en un desafío en tanto que un *worker* puede estar –y generalmente lo estará– ocupado analizando un subproblema cuando otro *worker* le solicite algún subproblema pendiente que se encuentre en su poder.

Es claro que, ante esta situación, no sería aceptable que el *worker* que está esperando una tarea se vea obligado a esperar hasta que aquel que la generó deje de *solvear* para poder acceder a la misma. Esto da origen a un requerimiento de asincronicidad entre *solving* y *serving*.

Sin embargo, tampoco sería aceptable que el impacto de servir tareas bajo demanda vaya en detrimento del análisis, que es, al fin y al cabo, el verdadero objetivo del *worker*. De aquí surgen un requerimiento de sincronicidad (por ejemplo: encolar los pedidos de tareas pendientes y atender sólo una cantidad limitada a la vez) y otro de balanceo de carga (por ejemplo: evitar que la capacidad de almacenamiento de un *worker* llegue a saturarse transfiriendo preventivamente tareas a otros *workers*).

En resumen, lograr ortogonalidad entre análisis y migración de tareas es importante para lograr escalabilidad, y sienta las bases para el desarrollo de mecanismos avanzados de balanceo automático de carga.

3.1.2. Aspectos de automatización e interactividad

La automatización de la operatoria es un requerimiento crucial. Desde el punto de vista del usuario, es fundamental que la herramienta sea capaz de llevar a cabo su objetivo –demostrar la propiedad o exhibir un contraejemplo– sin requerir supervisión alguna ni depender de complejas parametrizaciones que requieran numerosos experimentos para ser halladas.

Sin embargo, generar una estrategia automática que proporcione buenos resultados en la ejecución de un amplio espectro de problemas es sumamente difícil. Desde el punto de vista del experto que necesita diseñar, implementar, evaluar, depurar y mejorar una estrategia no supervisada, es importante contar con alguna interfaz interactiva que posibilite la inyección

de comandos arbitrarios que permitan la modificación del comportamiento de la herramienta en tiempo de ejecución.

La necesidad de atender estos requerimientos encontrados motivó la adopción de un enfoque en el que la maquinaria de cómputo distribuido (i.e. la comunidad de *workers*) no posee inteligencia propia, sino que se limita a proveer una serie de funcionalidades básicas que pueden ser invocadas remotamente, ofreciendo un alto grado de generalidad. La operación del sistema se lleva a cabo desde un tablero de control que permite explorar manualmente trazas de ejecución arbitrarias. Sobre esto se monta luego la automatización; en primer término la de aspectos reiterativos de rutina (i.e. carga de tareas al momento de la liberación de la unidad de cómputo, etc.), y finalmente la que implementa la toma inteligente de decisiones estratégicas (i.e. cuándo es conveniente partir un problema en subproblemas, etc.).

De lo antedicho surgen tres desafíos concretos. En primer lugar, el de lograr una arquitectura adecuada para ser totalmente controlable por un operador remoto humano. En segundo lugar, el de diseñar las interfaces de modo tal que provean una base adecuada para automatizar, tanto parcial como totalmente, dicho control. Y finalmente el de diseñar e implementar la estrategia automática propiamente dicha.

3.1.3. Otros aspectos de calidad

Modularidad del componente *sat-solver* de los *workers*

Otro de los objetivos que perseguimos a la hora de diseñar nuestra herramienta fue que la misma pudiera utilizar un *sat-solver off-the-shelf* para la resolución local de un subproblema concentrando las tareas vinculadas al desarrollo del *sat-solver* en los aspectos de orquestación del cómputo distribuido. Esto nos permite aprovechar los numerosos avances logrados por la comunidad en el área de investigación en *sat-solving* secuencial en los últimos años, y a la vez nos permite, a futuro, evolucionar a la par de los *sat-solvers* secuenciales, capitalizando sus nuevos logros a bajo costo y mitigando el riesgo de que la herramienta devenga obsoleta ante un nuevo avance en dicho campo de estudio.

Mantenibilidad y modificabilidad

El último objetivo que perseguimos durante el desarrollo de nuestra herramienta fue que la misma presentara facilidad para ser modificada sin que esto impactara negativamente en la *performance*. Este objetivo, entre otras cosas, determinó la elección de las tecnologías a utilizar para su desarrollo. Por lo tanto, se utilizó el lenguaje de programación `Python` para el desarrollo de toda la herramienta con excepción de las secciones destinadas al cómputo intensivo.

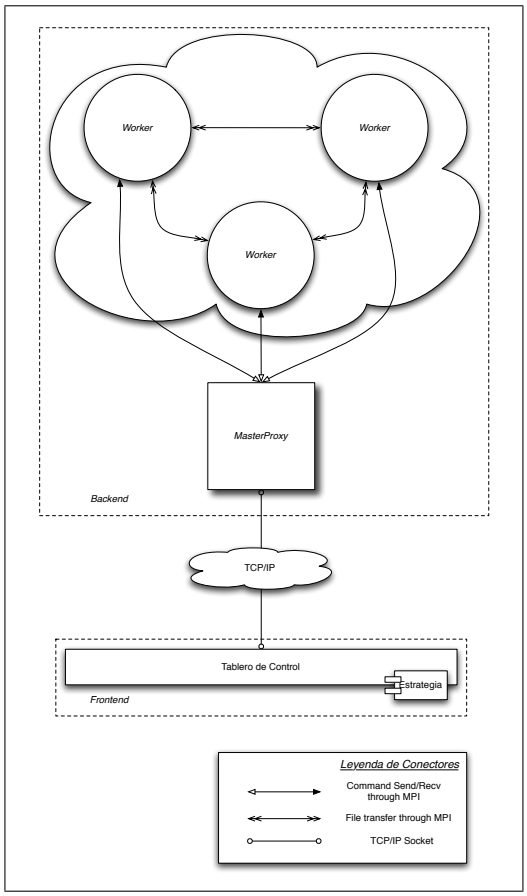


Fig. 3.2: Diagrama de componentes y conectores del *sat-solver* distribuido

Al utilizar un *sat-solver off-the-shelf* no fue necesario elegir un lenguaje para su desarrollo. En la implementación actual de la herramienta utilizamos el *sat-solver* `Minisat 2.2.0` que está desarrollado en el lenguaje `C++`. Para ello se desarrolló un *wrapper* que permite utilizar `Minisat` desde un entorno `Python`, obteniendo todos los beneficios de un lenguaje de programación dinámico sin pérdida de eficiencia en los algoritmos vinculados al cómputo intensivo.

Para el intercambio de mensajes entre los *workers* se utilizó el estándar `MPI` a través de la biblioteca `mpi4py` [13].

3.2. Arquitectura

La Fig. 3.2 muestra la arquitectura de la herramienta. En la misma se distinguen claramente dos componentes: el *backend* que ejecuta sobre un *cluster* de computadoras y el *frontend* que ejecuta en un equipo que se encuentra posiblemente fuera del centro de cómputos.

3.2.1. *backend*

En el *backend* se puede observar que existen dos clases distintas de elementos. Por un lado tenemos un proceso llamado *MasterProxy* que es el encargado de manejar la comunicación de órdenes desde el *frontend* hacia los *workers* y de reenviar las respuestas correspondientes desde los *workers* hacia el *frontend*.

En segunda instancia encontramos otro de tipo de procesos, los *workers*. Los *workers* son los encargados de relizar el cómputo (*sat-solving*) y la división de un problema en nuevos subproblemas. Asimismo son quienes almacenan las tareas pendientes de ejecución y por lo tanto deben proveer acceso a dicho almacenamiento a los demás *workers*.

Es importante destacar que si bien el *backend* presenta una arquitectura `Master-Worker`, el *MasterProxy* en este caso no toma ninguna decisión sino que simplemente actúa como intercambiador de mensajes entre el ambiente externo (el *frontend*) y el ambiente interno.

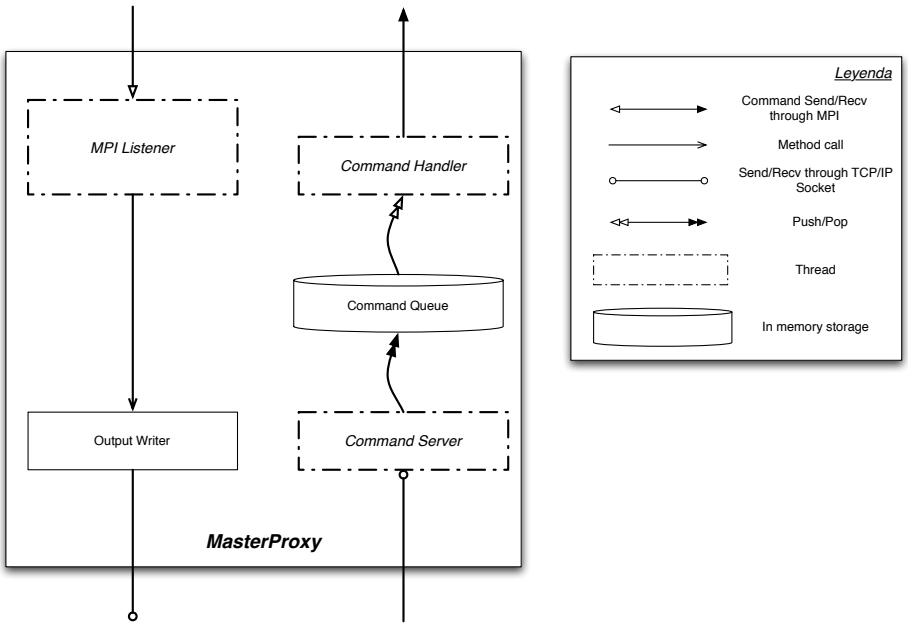


Fig. 3.3: Detalle de arquitectura del componente *MasterProxy*

La Fig. 3.3 detalla la arquitectura del *MasterProxy*. En el diagrama se puede observar que el *datapath* de entrada (*frontend* hacia *backend*) y el de salida (*backend* hacia *frontend*) son independientes. Esto es una consecuencia directa que se desprende de la ausencia de inteligencia en el *MasterProxy*.

Esto implica que no es necesario mantener el estado del sistema ya que en este componente no se toman decisiones. La ausencia de estado hace que el proceso *MasterProxy* consuma un mínimo de recursos, tanto de cómputo como de almacemiento.

La intervención de dos *threads* distintos en el *datapath* de entrada proporciona un alto nivel de respuesta ya que permite que algunos comandos sean implementados internamente mediante comunicación sincrónica a través de `MPI` sin que esto implique una pausa en el procesamiento del flujo de datos de entrada desde el *frontend* hacia el *backend*. La comunicación sincrónica permite simplificar algunos comandos como el envío del problema original hacia todos los *workers*. Al mismo tiempo se mantiene el orden entre los comandos enviados desde el *frontend* lo cual también contribuye a la simplificación del modelo de cómputo a pesar de tratarse de un sistema distribuido.

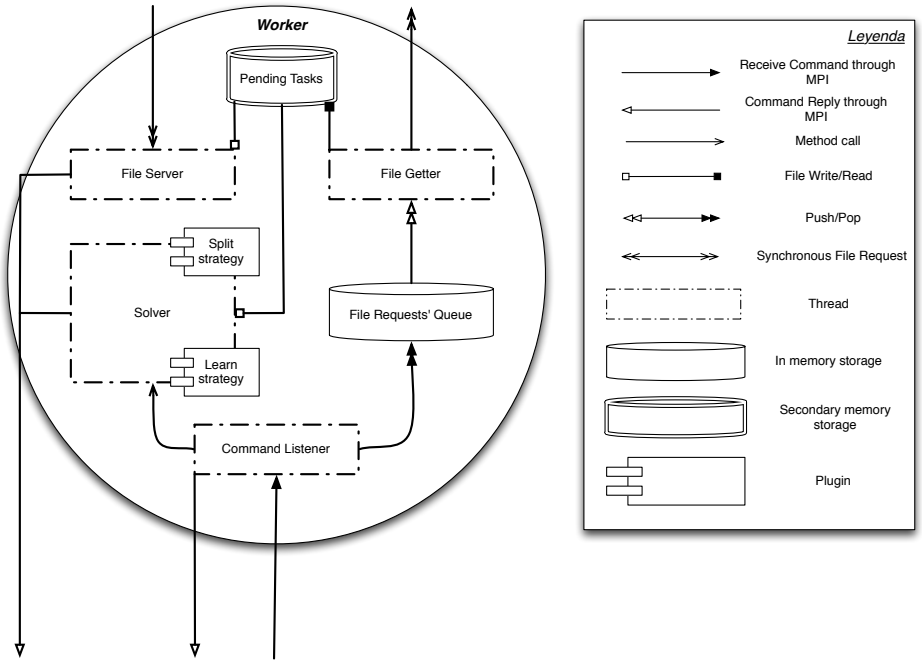


Fig. 3.4: Detalle de arquitectura del componente *worker*

La Fig. 3.4 muestra el detalle de diseño del componente *worker*. Como se puede observar, las funciones de *solving*, comunicación y transferencia de archivos se implementaron en *threads* separados. Dado que la función de *solving* utiliza principalmente tiempo de `CPU`, mientras que el resto de las funciones consumen principalmente tiempo de `I/O`, la separación de éstas

en distintos *threads* permite la ejecución concurrente de estas tareas sin que esta característica impacte negativamente en el desempeño del *sat-solver*. A la vez, esta característica permite satisfacer el objetivo de movilidad de las tareas, aún cuando el *worker* se encuentre ocupado *solveando*.

Interesa destacar también que la estrategia de partición de un problema se encuentran implementadas en los *workers*. Si bien este diseño es contrario a la idea general basado en la existencia de un *tablero de control* consideramos que el nivel de granularidad necesario para poder controlar remotamente la realización de estas dos tareas era demasiado alto y atentaba contra la usabilidad de la herramienta. Asimismo, la herramienta es capaz de poseer diversas estrategias implementadas de modo que el usuario puede seleccionar aquella que le resulte más apropiada (y sus parámetros en caso que corresponda) desde el *frontend*.

Por último, corresponde mencionar a qué nos referimos en concreto cuando hablamos de *tarea*. El término *tarea* fue mencionado en varias ocasiones a lo largo de la explicación. Si bien conceptualmente una tarea podría ser simplemente una fórmula φ en `CNF`, en la práctica esto podría producir un *overhead* en la transferencia demasiado grande ya que, por ejemplo al partir un problema levantando n variables podríamos generar 2^n subproblemas, cada uno de los cuales estaría representado esencialmente por la misma fórmula φ más las cláusulas unitarias que determinan el *guiding path* al que cada subproblema corresponde. Esta situación se repite innumerables veces a lo largo de la ejecución de la herramienta para resolver un problema. Es por esto que optamos, como modo de funcionamiento, por enviar la fórmula `CNF` del problema original a todos los *workers* al comienzo de una ejecución de la herramienta de modo que luego un subproblema esté representado sólo por el *guiding path* identificado por las cláusulas unitarias necesarias para forzar las decisiones correspondientes a ese subproblema. Esto hace que las tareas pendientes de resolución sean mucho más económicas de manipular.

Incluso pensando a futuro y considerando la posibilidad de implementar estrategias de aprendizaje es necesario que, como parte de una tarea, se puedan transferir también las cláusulas aprendidas que deban ser consideradas a los efectos de resolver esa tarea. Inclusive es posible que en un futuro una tarea no represente únicamente un subproblema que aún se debe intentar *solvear*, sino que podría también incluir otro tipo de actividades, como la partición de un problema. Esto último nos motivó a optar por un concepto de tarea genérico donde una tarea no es más que un paquete de archivos que el *worker* sabrá interpretar apropiadamente. En la implementación actual las tareas son exclusivamente un subproblema pendiente de ser atacado y se representan como un paquete con un archivo que incluye las cláusulas unitarias que deben adicionarse al `CNF` original para construir el subproblema correspondiente más dos archivos que pueden o no contener información. Estos últimos transportan la información de cláusulas aprendidas que deben ser incorporados al intento de *solving* del subproblema en cuestión.

3.2.2. *frontend*

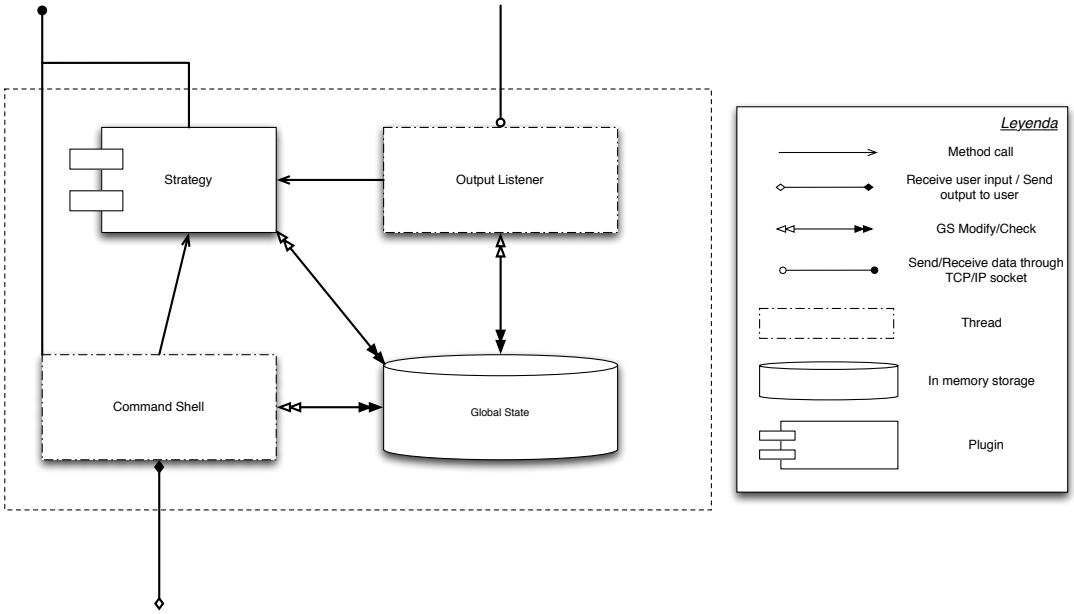
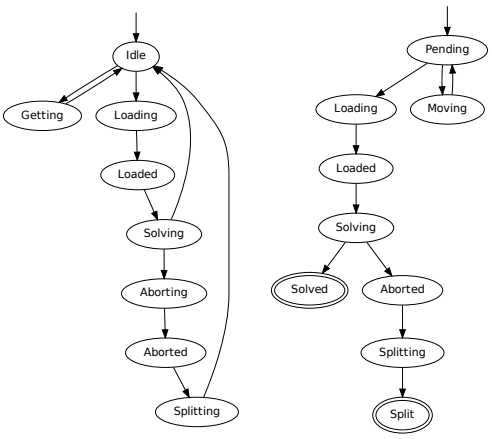


Fig. 3.5: Detalle de arquitectura del *frontend*

En la Fig. 3.5 se esquematiza el diseño de componentes del *frontend*. En esta figura se puede observar que el tablero de control proporciona un modo doble de interacción con el *backend*. Por un lado, el tablero de control es capaz de recibir comandos del usuario en *run-time* a través de una interfaz de tipo consola. Estos comandos representan acciones a ser llevadas a cabo en el *backend* como ser indicar a un *worker* ocioso que cargue una nueva tarea, indicar a un *worker* que se



(a) *Workers* (b) Tareas

Fig. 3.6: Diagramas de estado en el tablero de control

encuentra trabajando en un problema que aborte dicha ejecución, mover una tarea desde un *worker* a otro, etc. Por otro lado, el tablero de control posee también una *estrategia* (intercambiable) que permite llevar a cabo acciones automáticamente frente a determinados eventos. El *frontend* posee también un componente que procesa permanentemente la información que llega desde el *backend*.

Es importante observar que todos los componentes mencionados comparten un estado global. Esto se debe a que, tal como se explicó en la Sec. 3.2.1, el *backend* no tiene conocimiento del estado global del sistema. Por lo tanto, esta responsabilidad recae en el *frontend* en la medida que este estado es necesario para poder tomar decisiones ante los distintos eventos. El acceso concurrente a este estado global nos obliga a introducir un mecanismo de sincronización entre los distintos *threads* de ejecución (*locking*) que garantice la consistencia del estado en todo momento. Hemos tenido el cuidado necesario para evitar que el *locking* atente contra la performance del componente en cuestión.

En la Fig. 3.6 presentamos las máquinas de estado correspondientes a los *workers* y a las tareas. Estos diagramas esquematizan la información básica que es mantenida automáticamente en el modelo del estado global del *frontend*. Una estrategia particular podría requerir un estado más rico, que incluya mayores detalles, para tomar sus decisiones. Por ejemplo, se podría requerir mantener cierta información histórica sobre la ejecución en curso. En tal caso es responsabilidad de dicha estrategia la implementación de las estructuras de datos y algoritmos adicionales necesarios.

Listing 3.1: Esquema de comando sincrónico

```
DECIR_A W que obtenga la tarea T
ESPERAR que tarea T este en W
DECIR_A W que cargue tarea T
ESPERAR que tarea T este cargada
DECIR_A W que comience a solvear T
```

En cuanto a las estrategias, se siguió un modelo reactivo (*eventos*). Esto se debió en primera instancia a la dificultad de ejecutar secuencias de comandos de forma sincrónica. Por ejemplo, si quisiéramos implementar la obtención automática de una nueva tarea cuando un *worker* se queda sin tareas pendientes y quisiéramos además que el *worker* comenzara a *solvear* dicha tarea en el momento en el que la misma esté dispo-

Listing 3.2: Interfaz Strategy

```
class Strategy(object):
    def register_globalstate(globalstate)
    def register_socket(commandsocket)
    def on_init(worker)
    def on_createroot(worker, task)
    def on_init_finished(nworkers)
    def on_getfile(worker, task)
    def on_file(worker, task)
    def on_load(worker, task)
    def on_unsat(worker, task)
    def on_sat(worker, task, modelstr)
    def on_abort(worker, task)
    def on_split_newtask(worker, newtask)
    def on_split_finished(worker,
                          parenttask,
                          nchildren)
    def on_shutdown()
```

nible, deberíamos implementar un algoritmo similar al expuesto en List. 3.1. El problema en este algoritmo se encuentra en las líneas que comienzan con la palabra clave `ESPERAR` ya que no es aceptable que la consola de comandos o la estrategia se bloqueen a la espera de un evento que puede tardar una cantidad arbitraria de tiempo en ocurrir. Esto se debe principalmente a que en el transcurso de esa cantidad arbitraria de tiempo el *frontend* recibiría una cantidad de información desde el *backend* (por ejemplo, el hecho de que otro *worker* terminó de *solvear* el subproblema en el que estaba trabajando) que podría requerir que se tomen acciones con el fin de mantener el sistema funcionando eficientemente. Sin embargo, estas acciones no podrían ser tomadas porque el *frontend* se encontraría bloqueado esperando a que un evento particular de un *worker* particular ocurra. Ante este escenario optamos por un modelo de eventos que presentamos en el List. 3.2. Esta interfaz modela el conjunto de eventos que pueden ocurrir. Una estrategia particular debe implementar dicha interfaz de modo de poder responder ante estos eventos.

3.3. Sobre la automatización

En la presente sección nos proponemos señalar los aspectos importantes a tener en cuenta relacionados con la automatización del funcionamiento de nuestra herramienta. Además señalaremos las decisiones adoptadas por nosotros en cada uno de los puntos relevantes.

3.3.1. Consideraciones sobre la estrategia

Comenzaremos esta sección desglosando las principales cuestiones a resolver por una estrategia completamente automática para nuestra herramienta. Surge así la primer pregunta relevante que cualquier estrategia debe responder: ¿Cuándo declarar que un (sub)problema es demasiado difícil para atacarlo como tal? Es decir: ¿Cuándo se toma la decisión de partir un problema en nuevos subproblemas?

Dificultad de un problema

En este punto es importante destacar el hecho de que la partición de un problema en nuevos subproblemas se lleva a cabo con el objetivo de distribuir las distintas porciones del espacio de búsqueda entre distintas unidades de cómputo. Así, invertir demasiado tiempo en intentar obtener un resultado secuencialmente para un problema difícil atenta prohibitivamente contra el paralelismo. Por lo tanto la decisión de cuándo abortar un *solving* en curso se vuelve crucial.

Si un problema va a tomar demasiado tiempo no tiene sentido invertir tiempo de *solving* secuencial en intentar resolverlo y es conveniente par-

tirlo *cuanto antes*. Por otra parte, partir un problema en un momento tal que si hubiéramos esperado una fracción pequeña de tiempo más el *worker* habría arribado a un resultado para el problema implica que *todo* el tiempo de cómputo previamente invertido es desperdiciado. Estos dos errores se vuelven catastróficos cuando se repiten en forma sistemática sobre los subproblemas a los que un problema da origen.

Subyace a toda esta cuestión la pregunta fundamental de cómo determinar que un problema dado es fácil o difícil. Si tuviéramos algún criterio que nos permitiese siquiera aproximar la dificultad de un problema dado, o el porcentaje de avance durante una corrida secuencial de un *sat-solver*, podríamos elaborar una estrategia que atienda razonablemente a las inquietudes planteadas arriba. Lamentablemente no se conoce ninguna métrica que permita establecer *a priori* la dificultad de un problema. Si bien la complejidad de los algoritmos de *sat-solving* se expresa en función de la cantidad de variables de un problema (y es exponencial), la experiencia indica que existen problemas con pocas variables sumamente difíciles y problemas con muchas variables razonablemente sencillos.

Cuándo partir: criterios estáticos vs. dinámicos

Resulta claro entonces que una primera aproximación *naive* a este problema, como por ejemplo la elección de un *timeout* fijo a modo de criterio para determinar cuándo un problema debe ser partido, resulta una mala alternativa. La disparidad en la dificultad de los distintos problemas hace imposible dar con un único valor prefijado de *timeout* que resulte adecuado para el caso general. Otra posibilidad sería entonces que el usuario seleccionara el *timeout* deseado para cada problema. Sin embargo, esto atenta sustancialmente contra la usabilidad de la herramienta, ya que requiere asumir un *expertise* por parte del usuario y un conocimiento previo del problema a resolver que no son razonables.

Además, a medida que partimos recursivamente un problema, los subproblemas generados –si la elección de variables a levantar fue apropiada– son *cada vez* más fáciles. Por lo tanto un valor de *timeout* que resulte adecuado en cierto momento ya no lo será más adelante. Esto podría llevarnos a elaborar un criterio un tanto más refinado que lo anterior. Por ejemplo, podríamos optar por tener un valor de *timeout* para cada *nivel* del problema. Sin embargo, surge nuevamente el mismo inconveniente: cada vez que un problema es partido, los subproblemas generados difieren notablemente en su nivel de dificultad. Más aún, las distribuciones de dificultad entre los hijos de dos subproblemas distintos de un mismo problema (a misma cantidad de variables levantadas) pueden ser muy distintas. En consecuencia, este refinamiento tampoco resulta adecuado.

En general, la ausencia de un criterio estático para determinar la dificultad de un problema dado hace imposible la elección de un criterio de corte

que sea también estático. Por lo tanto se vuelve necesario que el criterio para determinar que un problema debe ser partido sea dinámico. Es decir, que el mismo se vaya adaptando durante la ejecución. Un criterio de estas características tiene dos fuentes de información fundamentales. Por un lado el estado general del sistema y su evolución (su historia). Esto puede incluir métricas sobra la carga del sistema, cantidad de problemas pendientes, espacio de almacenamiento disponible, etc. La segunda fuente de información surge de las características del problema que sean observables a lo largo de la ejecución. Ejemplos de esto son la cantidad de subproblemas producidos cada vez que se partió un problema, estadísticas sobre los tiempos que demoraron los subproblemas ya resueltos, etc.

Cualquier estrategia automática razonable deberá entoces dar respuesta a la problemática de decidir cuándo partir un problema teniendo en cuenta las observaciones mencionadas. Una vez resuelto este dilema surge la segunda pregunta fundamental a responder que es: ¿Cómo partir un problema? Que esencialmente se traduce en: ¿Cuántas y qué variables levantar en cada ocasión?

Cómo partir: criterios de selección de variables

La determinación de qué variables levantar en el momento en que se decide partir un problema es uno de los grandes temas en el mundo del *sat-solving*. En un sentido, este problema es equivalente al problema que enfrentan los *sat-solver* secuenciales cuando deben tomar una decisión. Cuando un *sat-solver* secuencial agotó las propagaciones de la última decisión tomada, debe tomar una nueva decisión. Tomar una decisión implica en primera instancia seleccionar una variable y luego seleccionar cuál de los dos posibles valores de verdad asignar a esa variable. En nuestro caso, dado que al levantar una variable los dos valores de verdad son considerados *simultáneamente*, el problema se reduce a elegir cuidadosamente la variable a levantar. Uno de los métodos más difundidos para elegir la próxima variable en los *sat-solvers* secuenciales, se basa en un criterio de actividad conocido como VSIDS. Del mismo modo que para las cláusulas –ver Sec. 2.2.2– cada vez que una variable *participa* de un conflicto su actividad es incrementada. Luego, cuando el *sat-solver* se encuentra en la situación de tener que elegir una variable para continuar la búsqueda, elige aquella con mayor actividad.

La elección de las variables a ser levantadas es crucial. Como se ve en la Tabla 3.1, una buena elección de variables puede generar una descendencia muy fácil o muy difícil. En general, no es posible predecir cuál es la mejor elección de variables a ser levantadas. Sin embargo la heurística VSIDS ha reportado muy buenos resultados en el mundo del *sat-solving* secuencial.

Además de la determinación de qué variables levantar, es importante también tener en cuenta cuántas levantar en cada momento. En primer lugar, es importante tener en cuenta que la cantidad potencial de subproblemas

Subconjunto de variables	Cant. hijos no triviales	Máximo	Media	Suma total	Desv. est.	Mediana
A	136	8.09	1.13	154.17	1.37	0.51
B	136	18.06	1.28	173.48	1.95	0.72
C	192	36.25	8.74	1678.88	7.45	5.78
D	174	63.62	1.25	218.29	6.27	0.05
E	192	89.41	2.18	418.04	7.66	0.39
F	192	288.79	10.18	1954.07	22.36	2.46
G	256	376.70	86.03	22024.53	81.98	56.98

Tab. 3.1: Ejemplo: siete elecciones de variables distintas para partir un mismo problema. Tiempos secuenciales, en segundos, para los subproblemas resultantes de cada partición.

generados es exponencial con respecto a las variables levantadas. Esto afecta en un doble sentido. En primera instancia es necesario observar que, dado que el proceso de particionado se repite recursivamente, la generación de una excesiva cantidad de subproblemas ante cada particionado genera una explosión exponencial de problemas. Si este factor no es tenido en cuenta es altamente probable que el sistema diverja y no sea capaz de arribar a una solución. Sumado a esto, el tiempo insumido en generar los subproblemas producto de un problema puede volverse excesivamente grande atentando una vez más contra el paralelismo y la eficiencia de la herramienta.

Una vez más debemos recordar que la idea de levantar variables se basa en que, ante una elección razonable de cuáles variables levantar, a mayor cantidad de variables levantadas, más fáciles serán los subproblemas generados. Es por esto que si la cantidad de variables a levantar es demasiado pequeña, es probable que los subproblemas que sean generados no sean suficientemente fáciles como para ser resueltos y que, por lo tanto, sea necesario partirlos también en el futuro haciendo que el tiempo invertido en intentar resolverlo se convierta en tiempo desperdiciado. Además, si la cantidad de variables levantadas es muy baja, existe la posibilidad de que la cantidad de subproblemas generados no sea suficiente para aprovechar al máximo el *hardware* disponible.

La última observación referida al criterio de particionado tiene que ver con qué problemas son dignos de ser distribuidos realmente. Hemos mencionado muchas veces que al levantar n variables, la cantidad **potencial** de subproblemas generados es 2^n. El uso de la palabra potencial tiene que ver con el hecho de que varios de los subproblemas generados al levantar n variables podrían ser resueltos trivialmente ¿A qué nos referimos con trivialmente? En primer lugar a aquellos subproblemas en los que alcanza con propagar[2] las implicaciones de las decisiones tomadas para obtener un re-

[2] El uso del término propagación en este contexto es estricto. Es decir que nos referimos exclusivamente a la realización de la clausura de *Boolean Constraint Propagation* sobre el

sultado –*sat* o *unsat*–. Pero más en general, nos referimos a todos aquellos subproblemas en los que el *overhead* de generar la tarea, trasladarla a otro *worker*, cargar la tarea en el nuevo *worker*, y *solvear* dicho problema hasta arribar a un resultado sea suficientemente grande como para que sea conveniente resolverlo directamente sin generar una nueva tarea.

Se introduce entonces una nueva variable en el criterio de particionado que es cuáles de los potenciales 2^n subproblemas generar como tareas a ser resueltas por otros *workers* y cuáles resolver directamente sin generarlos como nuevas tareas. Se debe observar que *no generar* un subproblema requiere invertir un tiempo de cómputo suficiente para determinar el resultado de ese subproblema. Una estrategia automática debe entonces decidir qué hacer con esta variable teniendo en cuenta que invertir poco o nada de tiempo de cómputo en filtrar los subproblemas puede degradar significativamente la eficiencia del sistema debido a que una cantidad de subproblemas demasiado fáciles serán generados como nuevas tareas. Pero también debe tener en cuenta que invertir demasiado tiempo de cómputo en el filtrado de los subproblemas tiende a degradar significativamente el paralelismo.

Otros aspectos de interés

Otro aspecto que influye en la *performance* de la herramienta es la forma en que se recorre el espacio de búsqueda. Cuando un *worker* queda ocioso debe tomar una nueva tarea (si hubiera tareas pendientes). Surge entonces el problema de determinar cuál de todas las tareas pendientes debe ser resuelta. Como en los casos anteriores existen dos factores que deben ser balanceados. Por un lado, si la estrategia (i.e. toma automática de decisiones) es adaptable según ciertas métricas sensadas dinámicamente, el orden en el que se recorren las tareas a resolver alterará el resultado. Luego, nos gustaría poder forzar a la herramienta a seguir el orden *más adecuado* en relación a dichas métricas. En este contexto, la tarea *ideal* puede encontrarse almacenada en otro *worker*. Surge así el segundo factor a tener en cuenta. En aras de la escalabilidad, el criterio para elegir la próxima tarea a ser resuelta no debe generar una sobreutilización de la red de comunicaciones.

Un último factor a tener en cuenta es la necesidad de maximizar la utilización del *hardware* disponible. Es decir que, idealmente, si hay *hardware* ocioso no debería haber tareas pendientes, aún cuando estas se encuentren almacenadas en otros *worker*.

3.3.2. La estrategia implementada

Teniendo en cuenta los factores mencionados anteriormente nos volcamos al desarrollo de una estrategia automática. Nuestro objetivo no estuvo puesto en el desarrollo de una estrategia óptima ya que cada uno de los factores

subproblema. Esta propagación no incluye la toma de nuevas decisiones.

mencionados con anterioridad podría ser objeto de un extenso y minucioso estudio. Más aún, no existen garantías de que tal estrategia óptima exista. Por lo tanto nuestro esfuerzo estuvo dedicado a no incurrir en los errores más flagrantes, de modo que la herramienta funcionase razonablemente bien con una serie de problemas y poder empezar a obtener resultados que nos permitan resaltar los aspectos fuertes y los aspectos que aún necesitan un esfuerzo importante.

En cuanto al criterio de particionado las decisiones tomadas fueron:

- La cantidad de variables a levantar se fijó en 10. Aunque este valor puede ser alterado por el usuario.
- Las variables a levantar se seleccionan utilizando la heurística VSIDS. Es decir que cuando un problema va a ser partido, se consulta la actividad de cada variable y se seleccionan –de acuerdo con el punto anterior– las diez más activas.
- Cada potencial subproblema es *solveado* por 50ms. Si en ese tiempo no se arriba a un resultado, se genera una nueva tarea pendiente.

Respecto al orden en el cual se recorre el espacio de búsqueda:

- Si un *worker* queda ocioso y no tiene tareas pendientes en su espacio de almacenamiento, obtiene la tarea que corresponda de acuerdo al recorrido BFS –*breadth-first search*– del árbol de que representa el espacio de valuaciones aún no explorado.
- Si al quedar ocioso el *worker* cuenta con tareas pendientes en su espacio de almacenamiento, se selecciona la que corresponda por BFS de las tareas locales.

Sobre cuándo declarar que un problema es demasiado difícil, adoptamos una estrategia basada en el estado del sistema. En concreto nos enfocamos en que el sistema mantenga una cierta tasa de progreso. Para ello medimos la cantidad de subproblemas resueltos por unidad de tiempo y luego establecimos un umbral objetivo. De esta forma, si la frecuencia actual se encuentra por debajo del umbral objetivo, entonces debemos escojer uno de los problemas que actualmente están siendo resueltos y partirlo.

Llamaremos **frecuencia objetivo** al umbral, entendido como cantidad de problemas resueltos por *worker*, por segundo por debajo del cual se decide que algún problema debe ser partido. Llamaremos **frecuencia actual** a un valor que estima la cantidad de problemas terminados por *worker*, por segundo a partir de promediar los problemas que han terminado durante los últimos t segundos. Llamaremos **tamaño de la ventana de muestreo** al

mencionado valor t. Llamaremos **frecuencia de observación** a la frecuencia con la que se revisa si la frecuencia actual se encuentra por encima de la frecuencia objetivo.

En concreto los parámetros utilizados fueron:

- Frecuencia objetivo: 0,15 p/*worker* p/seg.
- Tamaño de la ventana de muestreo: $20s$
- Frecuencia de observación: cada $5s$

Que la frecuencia objetivo dependa de la cantidad de *workers* permite ajustar la cantidad de problemas generados de acuerdo a la capacidad de consumirlos que tiene el sistema. Es importante notar que el tamaño de la ventana influye en cuánto peso se le da a la historia. Tomar una ventana de mayor tamaño exhibirá un comportamiento más estable en el tiempo, mientras que una ventana más pequeña se adapta más rápidamente y consecuentemente mostrará mayor variabilidad. La frecuencia de observación administra cuánto tiempo de tolerancia se tiene ante una frecuencia actual menor a una frecuencia objetivo. Si la frecuencia de observación es alta –menor valor– entonces la tolerancia será baja y ante cualquier caída de la frecuencia actual por debajo del objetivo el sistema reaccionará. Por otro lado, una frecuencia baja –mayor valor– representará una tolerancia mayor, dando oportunidad al sistema de tener períodos en los que la frecuencia actual no alcance la frecuencia objetivo. La combinación de estos tres parámetros configura la agresividad del sistema a la hora de determinar que un problema es demasiado difícil.

Además del criterio por frecuencia decidimos que, ante la situación en la que hay *workers* sin tareas para realizar y en el sistema no hay más tareas pendientes, se debe partir un problema en curso. De este modo evitamos que haya recursos ociosos. En particular esta decisión es útil al comienzo de la ejecución de un problema para distribuir rápidamente el trabajo entre las distintas unidades de cómputo.

Falta determinar entonces cuál de los problemas en curso será el seleccionado para ser partido. En este punto optamos por aquel en el que se ha invertido más tiempo de cómputo. De este modo evitamos que un *worker* quede atorado intentando resolver un problema que podría ser extremadamente difícil, a la vez que procuramos evitar que un problema sea partido demasiado pronto.

Con respecto al balanceo del almacenamiento se tomó la decisión de que cada vez que un *worker* queda ocioso y sin tareas locales pendientes, además de obtener la tarea que corresponde por BFS, el *worker* obtendrá una conjunto de tareas pendientes de aquel *worker* que posea mayor cantidad. Llamemos w_{idle} al *worker* que quedó sin tareas y $w_{overwhelmed}$ al que más

tareas pendientes posee, entonces la cantidad de tareas transferidas en esta ocasión será:

$$\frac{\#tareas_pendientes(w_{overwhelmed})}{\#workers} \tag{3.1}$$

La idea detrás de esto es que w_{idle} obtiene *la porción que le corresponde* de las tareas de $w_{overwhelmed}$. Para evitar transferencias extremadamente largas, la cantidad expresada en Eq. 3.1 satura en 10 tareas. Para evitar una mala distribución, las tareas que conforman una porción a ser transferida son seleccionadas al azar del conjunto de tareas pendientes de $w_{overwhelmed}$. Vale la pena notar que si un *worker* solicitara las n tareas que mejor califican según el criterio de elección, éste concentraría un conjunto de tareas de alta prioridad, generando un potencial problema de contención.

3.4. Resultados experimentales

En esta sección presentamos los experimentos realizados con el fin de evaluar el desempeño de nuestra herramienta. Los experimentos se ejecutaron en el *cluster* CeCAR[3]. Se utilizaron 17 equipos, cada uno de los cuales cuenta con dos procesadores `Intel Dual Core Xeon 5030 2.67 Ghz`, `2 GB` de memoria RAM y un disco rígido `Western Digital WD800JD 80Gb SATA 7200RPM`. Uno de los equipos fue utilizado para ejecutar el tablero de control y el *MasterProxy*, mientras que se ejecutaron 4 *workers* en cada uno de los restantes 16 equipos totalizando 64 *workers*.

3.4.1. Casos de estudio

Debido a la orientación del grupo de investigación, nos interesa evaluar la herramienta principalmente en lo que respecta a su uso para la verificación formal de software. En la práctica, nos interesa hacerlo sobre todo para problemas resultantes de la traducción de especificaciones de sistemas y propiedades en el lenguaje Alloy; esto es, de modo de evaluar la viabilidad de usar la herramienta para capitalizar la disponibilidad de *hardware* ocioso y reemplazar el uso de la herramienta secuencial Alloy Analyzer en aquellos casos en que esta última demora demasiado en demostrar una propiedad (o en que sus capacidades directamente se ven excedidas, de modo que el problema no resulta tratable en secuencial).

Otro aspecto de los problemas difíciles provenientes de especificaciones Alloy que resulta interesante para conformar un *benchmark* es que pueden ser generados en distintos grados de dificultad a medida que se elevan las cotas para los dominios de datos atómicos sobre los que predican. Como se mencionó en el Cap. 1, para que el análisis resulte decidible se procede a

[3] Centro de Cómputos de Alto Rendimiento de la FCEyN. `http://cecar.fcen.uba.ar/`

acotar dichos dominios (así, las propiedades se demuestran automáticamente pero, por ejemplo, "... para hasta 12 trenes, 8 semáforos y 8 pasos a nivel"). En el caso de Alloy, estas cotas suelen denominarse *scopes*. En otras palabras, el problema de verificar cierta propiedad sobre cierto modelo da origen a una sucesión o familia de problemas (para *scopes* cada vez mayores) que son estructuralmente similares pero ostentan cantidades de variables, cláusulas y tiempos de *solving* secuencial estrictamente crecientes.

Otra potencial fuente de problemas que resulta natural considerar son los *benchmarks* de las competencias anuales de *sat-solving*. Sin embargo, la forma en que tradicionalmente se evalúan las herramientas en dichos eventos consiste en medir, para cada *sat-solver* S y para cada cantidad de tiempo t, la *cantidad* de problemas que S logra resolver en t segundos. (Esto se debe a que algunos problemas fáciles para un *solver* son difíciles para otros y viceversa.) En consecuencia, los *benchmarks* de las competencias suelen incluir gran cantidad de problemas muy pequeños, pequeños y medianos cuyo tratamiento optimizado escapa al alcance de nuestra herramienta.

Garbage collection usando Mark & Sweep

Entre los casos de estudio provistos con el programa Alloy Analyzer se incluye un modelo formal de una implementación de *garbage collection* basada en el algoritmo conocido como "mark and sweep". El modelo incluye una representación del *heap* del sistema, dominios de datos `HeapState` y `Node`, una especificación del algoritmo y propiedades `Soundness` y `Completeness`.

La propiedad de `Soundness` afirma que, tras haber ejecutado el algoritmo de *garbage collection*, no hay intersección entre el conjunto de nodos alcanzables en el *heap* y la lista de nodos libres. La propiedad de `Completeness` afirma que, tras haber ejecutado el algoritmo, todo nodo inalcanzable en el *heap* efectivamente aparece en la lista de nodos libres.

Hemos incluido ambos problemas en el *benchmark* para la evaluación de la herramienta. Ambos son problemas difíciles que requieren exponencialmente más tiempo de análisis conforme crece la cantidad de `Node`s en el *scope*. Sin embargo, el segundo problema (el de demostrar `Completeness`) exhibe una curva de dificultad aún más pronunciada que el primero.

Ruteo en redes heterogéneas

Este caso proviene de un modelo formal de ruteo en redes heterogéneas, desarrollado en AT&T, que involucra agentes móviles, identificadores, rutas y múltiples dominios de red. El modelo en cuestión incluye diversas propiedades, de las cuales la última, `StructureSufficientForPairReturnability`, resulta particularmente costosa de demostrar. Para más detalles referimos al lector al artículo "Compositional Binding in Network Domains" [30].

Clausura relacional reflexo-transitiva

Se trata de un modelo sencillo que involucra un dominio de datos A y una relación $R \subseteq A \times A$, a la que (sólo) se le exige que sea reflexiva y transitiva. La propiedad a verificar afirma que $R^* = R$, esto es, que la clausura reflexo-transitiva de una relación reflexiva y transitiva es exactamente esa relación. Esta es una propiedad válida, cosa que podría probarse matemáticamente, por fuera del modelo. El interés radica en que se trata de un modelo breve y sencillo cuya traducción engendra problemas con cantidades muy pequeñas de variables proposicionales, pero que a pesar de ello resultan de considerable dificultad para los *sat-solvers* y exhiben una curva exponencial (tiempo de *solving* para $|A|$ creciente) relativamente pronunciada.

3.4.2. Resultados obtenidos

La Tab. 3.2 muestra la comparación entre el tiempo de ejecución requerido por el *sat-solver* secuencial (columna "seq. walltime (s)") y el tiempo percibido requerido por nuestra herramienta (columna "avg. par. walltime (s)"). Los tiempos reportados para el caso distribuido se calcularon tomando el promedio de varias corridas. La columna "reps" indica la cantidad de repeticiones realizadas para cada problema. La columna "speedup" es el resultado de la siguiente división:

$$\frac{\text{seq. walltime (s)}}{\text{avg. par. walltime (s)}}$$

Este valor denota la mejora relativa obtenida mediante la utilización de nuestra herramienta. La columna "efficiency" es el resultado de la siguiente división:

$$\frac{\text{speedup}}{\#workers}$$

Esta medida indica la ganancia neta de *speedup* por *worker* utilizado.

En primera instancia corresponde aclarar por qué la cantidad de repeticiones realizadas para cada problema son diferentes. Esto se debe a que procuramos realizar la mayor cantidad de repeticiones posibles para cada problema. Por lo tanto, la cantidad de repeticiones para los problemas más grandes fue menor que la de los problemas más pequeños.

Una de las hipótesis de trabajo es la disponibilidad de *hardware* y, en particular, la de *clusters* de cómputo intensivo que pueden ser usados para validar y/o verificar software.

Respecto a los resultados obtenidos, el *speedup* que se exhibe en la tabla, tal como lo evidencian los datos consignados en la columna *efficiency*, a menudo dista considerablemente del ideal si se considera que es resultante de utilizar 17 unidades de cómputo. Sin embargo debe destacarse que:

problem	scope	seq. walltime (s)	reps.	avg. par. walltime (s)	speedup	efficiency
Routing	8	308.26	7	60.46	5.10x	0.08
Routing	9	76168.16	6	407.34	186.99x	2.92
Routing	*10	>1209600.00	5	5046.84	>239.67x	>3.74
Closure	11	749.65	14	291.28	2.57x	0.04
Closure	12	3983.36	5	1914.45	2.08x	0.03
Closure	13	16261.35	5	4362.61	3.73x	0.06
GC Soundness2	9	217.31	10	200.85	1.08x	0.02
GC Soundness2	10	2855.30	7	1376.89	2.07x	0.03
GC Completeness	8	180.25	7	61.50	2.93x	0.05
GC Completeness	9	18643.06	5	1825.60	10.21x	0.16

*: Se ejecutó por 14 días sin que el *sat-solver* secuencial consiga un resultado

Tab. 3.2: Tiempo de ejecución (en segundos) distribuido vs. secuencial

- en general el *speedup* es creciente respecto del tamaño de los problemas que conforman el *benchmark*, indicando esto que el enfoque resulta apropiado para resolver problemas de gran tamaño que no son tratables en el contexto de *sat-solving* secuencial,

- si exceptuamos los casos extremadamente positivos –Routing–, y el caso GC Soundness2 para scope 9 en el que la mejoría percibida es muy baja, el rendimiento del enfoque reporta una ganancia mínima de 2,07x, promedio de 4,35x y una ganancia máxima de $10,21x$.

Uso de recursos en componentes centralizadas

En la Sec. 3.2 presentamos la arquitectura de la herramienta. En ella puede observarse que existen dos puntos centralizados de carga: el *Master-Proxy* y el Tablero de control. Cabe entonces preguntarse si estos puntos centralizados no representan una amenaza a la capacidad del sistema de incorporar más unidades de cómputo en el futuro.

La Tabla 3.3 muestra el impacto que tiene el agregado de nuevas unidades de cómputo sobre el desempeño del nodo en el que fueron ejecutados tanto el *MasterProxy* como el Tablero de control. En los datos consignados se puede observar que el impacto de estas componentes en la carga y en el uso de memoria del sistema son (y se mantienen) extremadamente bajos. Si bien lamentablemente no pudimos contar con grandes cantidades de nodos adicionales para garantizar que estos resultados se mantengan a gran escala, la tendencia parece confirmar que es posible incrementar considerablemente la cantidad de unidades de cómputo utilizadas sin que las componentes en cuestión se conviertan en un factor limitante.

workers	8	16	32	64
max. memory(1) (KB)	11344	11328	12152	10844
max. memory(2) (KB)	12784	12696	12728	13104
avg. system load	$< 1\,\%$	$< 1\,\%$	$< 1\,\%$	$< 1\,\%$

(1): Tablero de control. (2) *MasterProxy*

Tab. 3.3: Utilización de recursos del Tablero de control y del *MasterProxy* para cantidad de *workers* crecientes

3.4.3. Discusión y conclusiones

En términos generales, la herramienta consiguió disminuir el tiempo necesario para resolver problemas difíciles. Resulta particularmente interesante el haber podido comprobar la existencia de problemas del mundo real, provenientes de aplicaciones industriales, para los cuales el enfoque logra *speedups* fuertemente supralineales.

En el resto de los casos estudiados se obtuvieron *speedups* que, si bien son significativos y sin duda pueden resultar útiles en la práctica, distan mucho de ser lineales.

Nuestra conjetura es que esto está relacionado con el hecho de que, al partir un problema en subproblemas, suele obtenerse una considerable reducción en el máximo (es decir, en el camino crítico a la hora de reducir el tiempo total percibido por el usuario), pero a la vez un incremento potencialmente muy fuerte en la suma total del esfuerzo requerido para resolver el colectivo de subproblemas resultantes.

Asociamos el fenómeno antedicho con un importante factor de *retrabajo* que bien podría deberse a la ausencia de aprendizaje en nuestro enfoque distribuido de *sat-solving*. Es sabido que la incorporación y el correcto manejo de cláusulas aprendidas son un aspecto clave del enfoque CDCL. Sin embargo, en nuestra herramienta distribuida, cuando un análisis en curso es abortado con el objeto de partir el problema (identificado como difícil) en subproblemas, todo el conocimiento adicional hasta entonces aprendido por ese *sat-solver* secuencial es descartado.

Nuestra hipótesis es que es necesario aprovechar las cláusulas aprendidas para reducir el impacto del mencionado factor de retrabajo en el tiempo total de cómputo. En el capítulo siguiente presentaremos y evaluaremos distintas alternativas, basadas en herencia recursiva de padres a hijos, para la reutilización de este conocimiento.

4. APRENDIZAJE EN UN *sat-solver* PARALELO Y DISTRIBUIDO

En el Cap. 3 vimos que el enfoque de distribución aplicado consigue disminuir el tiempo necesario para resolver una serie de problemas de tamaño considerable. Al mismo tiempo observamos que existen casos en los que la eficiencia –entendida como cuánto ganamos por cada unidad de *hardware* agregada al cómputo de un problema– resulta pobre. Planteamos como conjetura que esta situación se debe, en gran medida, a una importante cantidad de retrabajo que se produce como consecuencia del particionado de un problema en diversos subproblemas. Nuestra hipótesis es que la reutilización de cláusulas aprendidas puede ayudar a disminuir la cantidad de retrabajo que se produce como efecto del particionado.

La existencia de retrabajo se evidencia en que si bien al partir un problema en subproblemas, el subproblema más grande suele ser más chico que el padre, la suma del tiempo requerido para resolver todos los subproblemas hijos es considerablemente mayor que el tiempo requerido para resolver el problema padre. Debido al esquema de particionado recursivo, este incremento en el tiempo total de cómputo requerido para resolver un problema se vuelve sustancial.

En el presente capítulo desarrollaremos un mecanismo de reutilización del conocimiento adquirido durante la ejecución de cada subproblema. Este mecanismo persigue dos objetivos principales: verificar la hipótesis planteada y, de ser posible, disminuir los tiempos requeridos para resolver un problema aumentando la eficiencia de la herramienta.

4.1. Justificación del enfoque

El enfoque de particionamiento recursivo, basado en *guiding paths*, que hemos adoptado para la construcción de la herramienta también influye en la elección de un mecanismo adecuado para compartir información de cláusulas aprendidas.

Consideremos ahora que un problema no es ya únicamente una fórmula φ sino una tupla $< \varphi, C >$ donde C es un conjunto de cláusulas tales que $(\forall v)(v \models \varphi \iff v \models \varphi \bigwedge C)$ llamado conjunto de cláusulas aprendidas. En este contexto el problema original es la tupla $< \varphi_R, \emptyset >$ donde φ_R denota la fórmula original que se desea verificar. En este modelo, al partir un problema $< \varphi, \emptyset >$ levantando (por ejemplo) la variable u obtenemos dos nuevos subproblemas $< \varphi_u, \emptyset >$ y $< \varphi_{-u}, \emptyset >$.

Notemos entonces que dado un problema $< \varphi, C >$ podemos generar los subproblemas que resultan de levantar la variable u como sigue: $t_u =<$

$\varphi_u, C_u >$ y $t_{-u} =< \varphi_{-u}, C_{-u} >$ (donde C_u es el conjunto de cláusulas que resulta de reemplazar las apariciones de la variable u por el valor 1 en el conjunto C y C_{-u} es el resultado de reemplazar las apariciones de la variable u por el valor 0) sin alterar la corrección de la herramienta. Esto se debe a que si $v \models \varphi_u$ entonces $v \cup \{u \leftarrow 1\} \models \varphi$ y por lo tanto $v \cup \{u \leftarrow 1\} \models \varphi \bigwedge C$ de lo que se deduce que $v \models \varphi_u \bigwedge C_u$. Lo mismo vale para el caso φ_{-u}.

Más aún, lo antedicho vale también para cualquier subconjunto de C. Esto induce una manera *natural* de reutilización de las cláusulas aprendidas que llamaremos *herencia*. Este mecanismo consiste en que, al generar nuevos subproblemas a partir de un problema padre $< \varphi, C >$, los subproblemas hijos son generados con un subconjunto de C como conjunto de cláusulas aprendidas. Llamaremos *criterio* a cada forma distinta de obtener un subconjunto de un conjunto de cláusulas C.

Consideraciones de escalabilidad

Más allá de la naturalidad del enfoque inducido por el esquema de particionado, nos interesa también evitar cualquier método que introduzca nuevos cuellos de botella en la arquitectura. Por ejemplo, no sería aceptable la incorporación de una gran base de datos centralizada en la que todos los *workers* almacenen las cláusulas que van aprendiendo y que todos los *workers* deban consultar.

Tampoco sería aceptable un grafo completo de comunicación permanente para lograr ese mismo fin, es decir, hacer que cada *worker* le comunique (o consulte) a todos los demás las cláusulas que va(n) aprendiendo.

Independencia del *sat-solver* particular utilizado

Por otro lado, como ya hemos señalado en el Cap. 3, nos interesa que el *sat-solver* secuencial que utiliza cada uno de los *workers* sea fácil de actualizar y/o de reemplazar por otro componente *off-the-shelf* similar. Si bien suelen ser necesarios algunos cambios para lograr que un *sat-solver* permita la inyección *inicial* de cláusulas aprendidas (en lugar de comenzar sin ninguna), por lo general tales cambios se reducen a cuestiones de interfaz y no revisten mayor dificultad.

Es más ambicioso, en cambio, pretender poder inyectar nuevas cláusulas aprendidas en cualquier momento *durante* el proceso de búsqueda. Para ello habría que introducir modificaciones mucho más fuertemente acopladas a la versión particular de *sat-solver* en uso, sus algoritmos, estructuras de datos e invariantes particulares.

Es por estos factores que optamos por implementar un esquema de reutilización de cláusulas aprendidas basado en herencia. En esencia el esquema funciona de acuerdo a lo detallado al comienzo de esta sección. En particu-

lar, a la hora de partir un problema en nuevos subproblemas un *criterio* es aplicado al conjunto de cláusulas aprendidas que el problema padre poseía al momento de ser abortado. De esta manera se generan los subconjuntos de cláusulas aprendidas que los subproblemas hijos deberán incorporar antes de comenzar el análisis. Cabe destacar que si bien la herramienta permite seleccionar un nuevo criterio de aprendizaje cada vez que un problema es partido, en la evaluación realizada no se utilizó esta característica sino que se mantuvo un mismo criterio para toda la corrida.

Sobre los criterios

En principio podríamos preguntarnos de dónde surge la necesidad de tener criterios para la herencia de cláusulas aprendidas. O lo que es lo mismo, por qué no heredar todas las cláusulas aprendidas del padre. Existen varias razones que motivan la necesidad de introducir criterios de selección de cláusulas.

En primer lugar la incorporación de cláusulas aprendidas a un problema incrementa la cantidad de cláusulas sobre las que es necesario propagar las decisiones. Esto provoca una ralentización del proceso de propagación que puede resultar en una disminución del rendimiento del *sat-solver* secuencial. Como ya se mencionó en 2.2.2, es sabido que incrementar en demasía la base de datos de cláusulas aprendidas en un *sat-solver* secuencial se torna contraproducente. Es por esto que todos los *sat-solvers* `CDCL` secuenciales cuentan con políticas de recorte de dicha base de datos y que, por lo general, estas políticas son bastante agresivas.

Otro factor que aboga en contra de mantener la totalidad de las cláusulas aprendidas por el padre es el hecho de que los subproblemas generados suelen ser más chicos que el padre. Incluso es bastante común que una buena proporción de los subproblemas generados al partir un problema padre sean considerablemente más chicos. En todos estos casos la incorporación de un conjunto de cláusulas aprendidas extremadamente grande, resultará inmediatamente contraproducente debido a lo expuesto en el párrafo anterior.

El último motivo para la introducción de criterios de selección es que los subproblemas generados no son resueltos inmediatamente sino que pasan a formar parte de la cola de tareas pendientes. Dado que el conjunto de cláusulas aprendidas de un problema es potencialmente muy grande, heredar la totalidad de las mismas en cada uno de los subproblemas hijos nos obligaría a pagar un alto costo debido a la escritura y posterior lectura –hacia y desde memoria secundaria– de cada uno de estos conjuntos (potencialmente muy grandes) más el costo de la posible transmisión por medio de la red. Esto atentaría directamente contra el desempeño de nuestra herramienta, incrementando en demasía los costos *fijos* asociados al enfoque.

4.2. Prueba de concepto

Con el objetivo de evaluar la pertinencia y viabilidad del enfoque propuesto para el aprendizaje, llevamos a cabo una prueba de concepto. La misma consistió en la ejecución de distintos problemas realizando una única partición y aplicando distintos criterios de selección de cláusulas aprendidas y comparando los tiempos obtenidos. Para ello cada problema fue ejecutado durante 60 segundos y luego partido levantando 5 variables para obtener 32 subproblemas. Cada uno de los subproblemas fue ejecutado hasta su finalización. Dado que, como ya mencionamos, la elección de variables a levantar influye fuertemente en la distribución de dificultades de los subproblemas resultantes, cada criterio fue testeado utilizando 10 secuencias de variables distintas generadas pseudoaleatoriamente. Luego se compararon tanto los tiempos totales de cómputo invertidos en resolver el problema como el tiempo del camino crítico (determinado por el subproblema que más tiempo tomó).

Resultados obtenidos

A continuación presentamos los resultados obtenidos en la realización de la prueba de concepto. Las Fig. 4.1 a 4.5 se organizan de la siguiente manera. Cada figura corresponde a la ejecución de distintos criterios de selección de cláusulas para un problema particular. Cada par de columnas denotadas con un número decimal corresponde a los resultados obtenidos con uno de los órdenes pseudoaleatorios. La columna de la izquierda de cada par de columnas indica el tiempo en segundos requerido para resolver el camino crítico. Mientras tanto la columna de la derecha indica la sumatoria del total de tiempos requerido para resolver todos los subproblemas. A su vez, el último par de columnas de cada figura, denotado con la leyenda "AVERAGE", contiene los promedios de los tiempos obtenidos.

No nos interesaba en esta etapa evaluar cuál o cuáles de los criterios eran mejores. Por lo tanto obviamos la denominación de los mismos en esta presentación. Sin embargo es importante aclarar que la última fila contiene los tiempos obtenidos para el criterio nulo, es decir aquel criterio que devuelve siempre el conjunto vacío.

El coloreo de las tablas se realizó utilizando una escala desde el color amarillo hacia el rojo para aquellos criterios que requirieron más tiempo que el criterio nulo, mientras que se usó una escala del amarillo hacia el verde para los que requirieron menos tiempo. La escala hacia el color rojo se saturó en dos veces el tiempo requerido por el criterio nulo mientras que la escala hacia el verde se saturó en 0,5 veces el tiempo requerido por el criterio nulo.

Como puede observarse en las figuras, los resultados obtenidos varían fuertemente de uno a otro problema y en función de cada elección particular

1		2		3		4		5		6		7		8		9		10		AVERAGE	
152.12	161.65	135.09	192.42	164.68	222.9	181.23	223.15	148.73	342.59	132.03	238.48	147.73	194.78	160.08	232.89	154.27	458.25	166.41	192.33	156.39	245.66
146.02	154.80	151.31	205.33	154.07	212.96	185.14	224.88	147.70	354.90	195.18	317.35	151.74	200.12	140.34	221.04	144.02	505.32	146.75	173.83	156.23	257.05
120.84	138.68	157.12	212.22	149.28	207.9	158.46	203.27	153.74	345.77	173.98	283.96	171.44	221.11	139.66	212.56	150.23	498.55	174.42	206.18	154.82	253.03
183.89	198.21	132.45	199.26	166.45	238.05	184.52	228.22	168.18	387.43	217.79	370.43	174.09	217.03	182.85	252.05	181.14	568.71	155.16	196.03	174.65	284.41
132.73	140.14	154.19	209.80	179.12	220.8	168.41	213.25	189.33	383.20	172.89	259.30	157.94	206.13	121.20	214.69	170.84	535.17	129.56	156.14	154.92	254.78
133.14	149.39	202.35	255.85	178.78	240.92	196.24	237.28	167.61	366.82	154.37	244.07	138.23	182.38	128.67	215.98	150.85	450.13	113.39	160.97	154.33	249.29
168.98	180.55	241.21	307.78	233.48	293.96	271.43	314.86	233.27	467.52	232.88	346.41	205.80	258.34	175.22	258.97	166.81	622.57	262.33	301.96	219.01	335.29
176.48	198.52	213.91	285.20	224.82	358.31	234.07	277.10	195.75	451.37	216.28	378.38	220.91	271.83	174.53	246.69	159.22	522.36	221.88	256.22	203.88	324.60
214.36	239.29	189.19	253.77	202.52	341.59	212.38	256.07	213.03	480.84	230.08	421.71	235.50	297.65	174.85	261.51	213.70	671.30	241.02	282.30	212.66	349.60
182.28	198.22	269.62	341.25	247.01	328.52	249.31	282.12	222.48	501.19	255.32	457.89	244.63	294.16	189.59	252.49	178.04	675.50	200.29	229.62	221.86	353.07
268.40	297.33	195.83	255.28	306.44	409.70	309.19	385.03	297.42	480.04	311.34	511.60	294.97	354.77	223.08	352.97	230.78	858.76	299.16	307.10	273.85	441.35
199.77	216.82	246.19	330.10	228.20	451.74	223.93	277.29	198.06	453.96	238.93	382.85	222.54	275.54	190.41	272.96	187.41	671.90	213.87	247.81	213.83	358.10
207.90	235.95	149.82	217.86	248.93	363.28	189.99	244.23	242.03	520.23	196.60	358.53	233.55	279.25	232.85	303.85	206.78	717.31	213.70	249.84	212.20	348.04
200.31	219.93	182.04	250.64	233.87	312.48	225.94	277.95	274.04	543.92	257.02	464.48	282.44	331.97	223.38	301.40	215.96	661.56	212.82	255.67	230.87	361.98
201.69	215.89	223.51	296.56	212.99	367.80	197.38	241.06	186.20	420.59	261.43	410.50	200.04	250.45	174.90	242.39	183.54	598.43	195.44	230.58	203.71	327.43
192.91	210.62	187.32	247.78	254.22	448.06	234.02	279.05	164.01	387.55	247.63	409.94	208.77	263.62	128.24	200.20	187.22	634.65	218.82	253.85	202.12	333.53

Fig. 4.1: Routing *scope* 8

1		2		3		4		5		6		7		8		9		10		AVERAGE	
6507.51	6576.71	4897.54	9916.88	10243.32	29151.82	5444.41	10978.45	3733.39	4405.46	4964.59	9206.28	56093.81	69346.04	4802.65	5970.60	4973.84	5933.68	6017.30	6448.86	10837.78	15693.49
5887.69	5951.87	4689.38	8734.54	10682.55	35470.39	6862.81	14348.13	5472.52	6068.96	5940.54	10179.44	9301.18	18779.22	5972.66	4773.33	5873.06	6877.58	8973.89	9379.45	6786.01	12056.32
5957.48	6019.92	5922.43	11964.79	9773.39	39390.88	7100.77	14252.52	7085.47	8248.18	9013.86	15890.08	8475.50	14927.89	7077.41	7825.15	7573.18	8432.88	10064.40	11590.50	8264.31	14047.18
7824.37	7893.30	7494.45	12694.15	9268.32	35910.11	14861.25	23937.66	8017.65	8710.99	6106.03	9134.24	4248.38	10458.22	6142.83	6869.73	4562.45	5775.09	12535.32	13808.18	8178.28	13430.05
5336.42	5389.59	6216.18	12102.47	11944.24	37547.78	7708.89	14487.37	13257.63	13921.18	5361.15	10215.41	6562.25	13470.73	6374.32	7337.54	4824.30	5796.11	5487.15	5918.71	7224.78	12531.68
5623.55	5690.07	7577.27	13034.29	8251.38	32152.67	12217.39	21082.28	5854.35	6422.91	10324.81	15405.48	5938.06	11276.60	6085.44	6870.07	8332.61	9512.78	5484.59	5871.90	7509.94	12767.00
6295.40	6371.10	8195.93	13882.13	14893.24	41367.18	7071.11	14430.16	5385.23	6171.81	5349.50	9670.28	5168.75	12271.65	5032.21	6070.30	4256.19	5361.04	5349.64	5855.17	6690.72	12125.08
5545.83	5637.00	6776.64	10918.75	10241.05	38562.40	12422.51	19984.13	1068.32	1247.02	5328.34	10407.54	6900.13	14728.11	10847.33	11165.18	5338.55	6658.65	5607.09	6015.16	8167.17	14172.43
9675.03	9742.55	7694.74	14831.20	8497.17	35931.32	8982.37	15585.04	6832.44	7456.99	6374.75	11112.35	5138.75	10774.25	7848.07	8684.13	6855.53	8370.21	[illegible]	[illegible]	7132.98	12664.76
9500.69	9564.49	10353.77	22758.68	12597.48	45560.71	4898.36	15240.06	5743.79	6327.01	7639.37	14194.01	13561.82	23459.51	6876.13	7726.26	7031.14	7930.07	7649.41	8068.82	11525.25	18122.92
7031.86	7118.24	7620.15	13699.91	9405.74	47550.94	11275.61	23478.31	9481.39	10179.47	6162.77	15168.26	7963.08	17727.71	9005.83	9997.22	7556.49	8791.54	8879.71	9421.77	9534.15	16336.34
6349.45	6340.39	12453.74	19386.18	7743.23	31810.34	14159.78	22678.64	6637.61	17066.40	6169.39	9965.17	11268.32	18869.43	4111.90	4924.32	8891.77	9924.18	6984.21	7499.76	9408.94	14836.81
11124.44	11192.91	6152.87	11019.37	9404.90	39112.39	13537.81	23730.83	6838.16	7514.32	7042.73	11597.65	6280.09	12120.52	7124.33	8008.42	5996.66	6901.81	6782.35	7285.19	8030.43	13848.32
7274.47	7374.21	5412.87	11127.58	14470.84	47912.48	7023.57	14929.66	10961.79	11231.32	5903.67	10297.71	19186.49	24116.76	7882.83	8326.70	3889.12	4867.42	5747.54	6397.01	9673.57	17027.08
8214.63	8273.32	10481.41	19417.28	10688.08	39875.83	10324.25	17571.45	5957.08	6577.90	6735.65	12424.78	11989.56	18894.56	4767.22	5529.82	4666.81	5661.72	7158.77	7644.02	11199.82	23872.68
6703.11	6782.78	5445.89	10509.32	6101.35	29357.94	7671.09	13623.71	4302.57	4945.44	4422.76	7524.83	6750.15	14924.27	5344.44	6257.49	3637.32	4810.21	6556.81	7029.82	5693.55	10576.58

Fig. 4.2: Routing *scope* 9

1		2		3		4		5		6		7		8		9		10		AVERAGE	
45.02	50.07	43.77	73.22	39.01	105.13	41.31	46.04	38.08	80.79	49.55	57.24	43.62	65.37	36.34	51.23	47.19	54.70	55.07	63.11	43.91	64.67
41.64	45.48	39.40	59.08	36.39	76.88	40.09	43.30	37.59	54.81	41.54	47.77	42.21	57.18	33.17	43.39	43.25	60.81	39.13	48.08	39.43	53.68
38.48	41.88	38.37	56.38	37.28	82.54	41.94	45.55	38.23	54.98	42.86	50.01	47.33	62.24	33.24	42.99	39.57	55.68	39.08	46.66	39.83	53.68
41.18	46.22	51.28	93.72	45.42	111.73	50.13	55.06	42.42	68.21	45.73	54.87	47.30	71.01	35.74	54.08	42.53	68.35	49.53	78.71	45.12	68.20
40.28	43.86	36.18	53.18	37.52	78.74	41.43	44.39	36.88	52.17	38.15	43.50	52.81	68.29	35.60	47.92	44.16	64.78	39.70	47.40	40.27	54.42
39.92	42.26	36.34	52.77	35.37	76.44	38.39	42.08	37.46	54.18	42.92	48.23	39.13	56.88	33.81	45.99	38.19	54.26	42.79	48.23	38.87	55.62
56.43	69.36	72.87	156.99	54.65	198.62	50.48	60.77	54.62	116.58	55.75	75.08	59.19	111.93	48.51	97.54	56.62	170.17	50.87	79.65	56.00	113.67
48.75	58.09	53.07	118.93	49.92	167.08	50.77	58.51	47.71	97.57	66.83	82.22	71.64	117.24	43.91	82.91	50.13	146.24	68.72	93.64	55.14	102.24
49.34	58.91	62.93	133.65	52.33	142.18	63.18	71.93	46.61	98.79	58.99	79.24	65.82	104.49	45.71	86.12	53.36	155.28	53.77	79.76	55.40	101.03
38.34	43.89	56.94	137.61	38.21	78.79	39.24	42.20	37.33	53.78	47.00	56.52	48.03	72.88	32.79	43.28	51.54	150.08	42.15	52.06	43.46	84.27
39.79	43.87	43.78	70.26	36.71	68.77	37.73	40.68	35.47	52.38	39.69	47.03	38.47	53.14	32.75	45.80	34.25	51.19	37.42	45.72	37.82	54.86
60.68	70.79	39.59	56.28	51.94	164.77	65.34	74.65	45.88	104.60	71.65	89.63	68.09	114.26	50.20	85.84	67.10	165.12	63.25	92.14	58.37	101.80
42.34	46.38	53.18	107.81	38.27	84.37	43.34	46.42	38.51	60.21	45.86	51.88	45.99	66.08	35.29	48.50	62.02	110.11	39.13	48.54	44.33	63.28
63.46	75.84	53.67	94.66	57.59	167.09	55.07	63.87	54.25	115.63	64.19	81.91	58.95	99.05	43.70	81.02	58.41	164.22	68.37	98.71	57.57	104.20
47.67	55.14	65.27	135.34	42.94	112.48	45.18	50.38	39.93	68.17	67.38	55.80	53.31	77.05	37.20	55.80	44.34	68.48	51.18	60.83	46.87	76.52
55.90	68.92	48.58	79.84	49.38	182.95	60.72	72.08	54.90	127.95	53.45	74.60	61.81	116.52	47.20	91.29	55.84	165.91	71.28	100.83	55.91	108.10

Fig. 4.3: Closure *scope* 9

1		2		3		4		5		6		7		8		9		10		AVERAGE	
101.59	345.60	102.12	271.52	93.19	132.7	87.29	300.32	78.82	242.18	90.60	124.86	83.40	181.28	105.41	109.28	112.32	212.79	120.11	379.79	97.70	239.0
88.35	239.71	85.22	225.39	89.89	106.43	61.48	178.98	83.71	277.65	71.48	95.41	65.08	112.86	67.89	227.42	78.18	193.08	78.99	266.68	74.60	196.34
93.13	239.88	92.53	249.85	63.28	94.94	66.77	181.79	73.51	260.90	71.40	98.85	68.37	120.97	66.99	209.63	61.87	180.45	71.98	231.07	73.91	190.19
99.29	314.57	86.47	267.99	94.91	142.16	88.08	280.61	80.05	347.19	81.56	99.79	78.18	165.71	66.08	277.96	72.79	221.58	105.29	350.03	83.29	246.75
70.03	230.86	84.26	241.88	79.20	90.79	66.88	185.30	69.27	262.10	69.05	102.34	65.28	119.46	77.14	238.28	76.58	174.91	78.13	257.77	70.98	188.79
73.81	237.50	72.41	197.75	77.37	89.47	61.47	191.09	63.08	216.77	74.18	99.21	73.24	107.88	60.71	191.09	71.42	179.09	69.17	231.90	69.95	156.0
144.21	601.10	169.75	601.76	189.13	287.31	245.06	824.22	262.98	870.33	108.30	195.16	205.38	361.88	130.73	580.86	225.63	670.13	161.15	587.33	184.25	558.01
180.41	555.33	280.62	597.84	128.31	234.82	152.07	345.14	166.50	639.38	126.38	190.88	165.12	284.64	104.68	400.02	118.06	380.35	130.28	460.87	147.85	452.94
178.07	666.97	199.03	608.22	213.90	329.80	155.06	555.95	205.61	667.90	109.24	174.46	193.23	344.78	196.45	646.20	178.55	562.05	213.80	657.88	184.30	521.42
127.13	451.48	210.09	666.74	94.30	167.56	94.93	325.45	87.44	400.82	88.49	134.04	114.14	200.83	83.43	246.13	111.49	344.81	116.99	365.06	114.73	325.38
126.18	378.33	136.00	391.75	96.01	156.88	100.62	327.68	159.92	504.94	88.83	126.78	123.97	213.47	101.47	326.49	98.77	317.91	117.91	339.45	115.01	308.84
222.90	701.12	114.22	316.88	142.96	261.65	215.39	707.87	162.87	589.11	96.32	175.12	109.83	246.75	120.71	544.30	209.66	674.57	139.42	510.65	153.43	472.80
97.[illegible]	268.40	133.03	455.82	119.89	195.56	92.94	298.8	137.34	456.28	98.10	157.01	103.74	186.65	93.65	189.99	89.[illegible]	298.[illegible]	111.09	356.76	105.14	299.28
197.19	706.40	95.01	316.95	115.13	207.24	159.24	521.41	257.26	881.14	191.14	273.75	158.66	288.48	212.32	650.98	152.68	541.00	223.58	679.16	176.23	506.65
166.81	484.81	127.94	506.69	117.64	204.37	98.41	316.16	118.73	473.30	95.33	141.89	167.24	363.98	96.13	210.08	126.28	391.24	195.33	614.52	130.91	387.61
200.20	614.28	121.44	393.22	141.22	274.80	149.22	555.55	193.42	777.29	134.66	250.37	132.22	308.90	148.89	449.44	199.60	575.35	133.78	507.73	155.67	494.69

Fig. 4.4: Closure *scope* 10

1		2		3		4		5		6		7		8		9		10		AVERAGE	
2735.83	7263.39	1543.76	5363.44	3504.94	17535.62	3399.82	14185.23	2561.35	5254.13	2130.99	14345.31	1512.48	5430.41	2565.74	15384.33	2291.95	16076.33	2185.05	8207.63	2423.17	10904.58
2206.35	6788.55	2951.31	6822.56	2896.80	16696.64	2288.05	12747.07	2188.39	6860.82	2647.88	16494.06	1605.42	5481.70	2624.14	15523.30	2490.87	16556.41	3556.42	10014.27	2545.40	11378.61
2340.42	6608.62	1956.30	5997.42	2221.80	15015.77	1779.88	13387.16	1772.98	5859.56	2139.74	14465.44	2270.06	6577.36	1950.78	14014.08	2535.24	16542.18	3601.81	10715.73	2378.98	10422.84
2499.77	7738.69	1749.44	5557.19	2350.38	15259.82	2448.64	13812.55	1765.52	4632.26	2602.48	16509.30	2697.22	6656.66	2312.64	15147.31	2068.74	14984.42	1931.01	7381.86	2342.59	10768.01
2206.45	6508.45	1451.63	4744.97	2320.64	15123.05	2198.81	11412.44	1879.85	4392.52	1953.19	13387.38	1792.18	5783.85	2220.93	11461.81	2658.79	15385.28	2632.19	9003.41	2134.47	9720.92
1875.08	5885.46	1909.08	6751.71	2857.87	16857.24	2598.83	15270.70	1902.74	5876.05	2104.17	15771.33	2143.04	6485.45	1872.84	14031.97	2459.45	16512.56	2369.06	9027.22	2367.01	11344.99
3004.21	7321.93	1814.83	6396.72	1896.72	15604.67	2226.53	12397.07	2756.47	6904.88	2060.23	17165.54	2319.27	6357.11	3703.60	17129.22	2417.09	16245.25	2259.73	8894.85	2445.87	11441.72
2271.83	7066.70	2542.44	6845.21	3041.31	16171.77	2326.81	14872.90	2172.73	5802.77	2246.75	15346.65	1903.74	6065.45	2835.84	13491.77	2715.38	18109.75	2323.92	8978.71	2438.05	11404.97
2268.83	9904.05	2793.84	7608.16	2323.46	13899.37	2758.94	14794.18	2438.55	7710.03	2839.80	16439.55	2513.76	6955.83	3987.14	17481.97	3089.25	19471.03	3357.23	11390.15	3138.66	12505.07
3568.19	11524.34	5413.07	9990.78	2187.74	14990.37	4556.68	16440.28	3540.12	7161.54	1910.85	14108.55	1838.64	6170.69	3633.67	16898.17	2348.56	16713.59	2667.12	10039.00	3166.36	12387.63
1931.84	7074.82	1562.00	6037.33	3326.30	17969.01	1895.81	9667.37	1944.79	5438.24	1955.10	13837.07	1600.21	6080.82	3100.84	15387.74	3414.11	18611.[illegible]	1866.38	8106.60	2261.74	9295.23
1931.83	5649.44	2200.12	6027.02	2221.80	15015.77	1779.88	13387.16	1772.98	5859.56	2139.74	14465.44	2270.06	6577.36	2942.65	14865.79	1893.63	14981.14	2930.44	9626.43	2222.51	10693.33
1623.03	5598.80	1423.23	4435.77	1454.78	11631.66	1450.47	7742.16	1237.58	3932.52	1842.14	12942.98	1548.32	5126.35	1512.06	8180.25	1575.07	12008.81	1688.72	6474.49	1514.73	7817.38
1547.58	5457.46	1563.35	5945.80	1691.68	12866.75	1511.73	10270.19	1449.30	4614.47	1726.14	11761.12	1914.06	5486.88	2081.78	10538.18	1596.27	11805.88	1593.15	6880.86	1627.62	8363.16
1876.07	5960.71	1873.12	6030.50	1735.05	14324.74	1810.64	13281.29	1902.74	5572.90	1856.08	13001.24	2879.32	7423.46	1985.44	10311.65	1972.81	14535.38	1873.30	7173.70	1940.35	9763.29
2011.07	5624.51	1752.23	5339.02	2019.23	13906.11	2231.02	12089.52	2882.70	6428.97	2103.70	16028.02	2146.74	7223.00	2868.29	16645.33	2691.84	14774.04	2494.77	10054.57	2320.16	10801.31
2020.21	6347.35	1784.78	5558.30	1759.52	14351.64	1862.24	11822.26	1980.18	6165.65	2037.49	14912.13	1864.85	7102.70	2128.19	10973.63	1878.79	14804.38	1817.78	7515.54	2032.20	9925.16
1880.12	4348.41	1355.20	3898.44	1658.07	11185.38	2509.20	7211.14	1174.67	3952.65	1553.86	10779.96	1208.28	4786.44	1549.39	9156.41	1555.25	14160.60	1756.32	7801.86	1579.39	7124.44
1520.07	4708.45	1337.06	4281.13	1553.18	10966.30	1279.14	9073.13	1281.72	3797.96	1526.02	10497.10	1732.90	5477.36	1424.19	7466.41	1531.60	11410.23	1624.82	5200.60	1553.80	7234.97
2212.80	7803.03	2000.45	6927.47	1993.42	13865.39	2179.75	10569.32	1936.22	5827.25	2269.78	16238.30	2237.36	8182.67	2185.25	14699.13	2552.84	16400.90	2024.21	8489.14	2252.72	11204.25
1892.42	5086.54	2248.14	6947.72	1649.52	14122.74	2593.77	13772.81	2152.84	6331.73	2049.91	16080.54	1715.88	6369.32	2293.93	14175.28	4157.91	18900.13	1906.17	7776.77	2262.53	10787.36
1373.38	4842.69	1405.84	4257.51	1561.32	10892.5	1588.13	8131.60	1385.59	4260.12	1281.68	10712.19	1607.19	5672.38	1354.20	7544.09	1323.12	9851.80	1393.39	5312.74	1388.59	7147.86
1641.28	5181.55	1444.08	4316.29	1506.68	10873.34	1526.73	10993.59	1138.03	3983.93	1380.28	9466.96	1787.88	6698.57	1107.83	7450.38	1396.71	11417.11	1557.37	5787.13	1440.01	7409.06
2792.42	8921.48	1863.12	5548.61	2365.20	16845.65	1941.47	12333.42	1920.06	4585.25	2209.31	16282.20	2605.97	7636.82	2917.25	16918.11	2661.19	16399.16	2597.52	10123.63	2387.35	11549.43
2022.27	5558.50	2000.65	5516.67	2052.73	15971.78	2456.72	14684.77	3033.84	7527.74	2711.98	16793.82	1584.05	5932.17	3645.40	15834.99	1710.09	13926.79	2118.42	8284.35	2335.63	11003.54
1440.07	4930.51	1331.04	3998.74	1859.23	13461.89	1453.91	7270.13	1200.20	3692.37	1798.42	13852.14	1603.58	5267.35	1784.26	9187.27	1728.85	12739.84	2417.32	8125.02	1662.80	8327.97
1324.19	4692.20	2201.11	5643.33	2138.34	13246.11	1753.21	10901.20	1319.09	4052.45	1727.04	12740.49	1683.37	5799.53	1781.43	9321.57	1805.78	12831.55	1528.18	6788.33	1746.59	8590.68
2300.44	6647.78	1714.83	5304.86	2491.64	17659.03	1520.08	8679.13	1542.37	4619.88	2568.20	16569.06	2361.84	6825.13	2235.96	12521.81	2427.22	17778.18	2289.93	8575.93	2143.25	10498.08

Fig. 4.5: Soundness2 *scope* 8

de variables. Además, los resultados no podrían ser del todo concluyentes en tanto constituyen una aproximación de orden 1 al enfoque *divide-and-conquer*: en esta prueba de concepto, el problema no se parte recursivamente sino una única vez en subproblemas.

Más allá de estas limitaciones, cabe observar que en algunos casos varios de los criterios reportan ganancias considerables. Aún más importante es el hecho de que para la mayoría de los problemas hay al menos un criterio que reporta resultados positivos. Interpretamos esto como evidencia suficientemente alentadora para embarcarnos en una evaluación más realista de la técnica en el marco de nuestra herramienta distribuida.

4.3. Resultados experimentales

En esta sección mostraremos los resultados experimentales obtenidos a partir de la implementación de la herencia de cláusulas aprendidas.

Comenzaremos por definir los criterios de selección de cláusulas utilizados. Cabe aclarar que de aquí en más se considerará que una cláusula aprendida contiene al menos 2 literales. Llamaremos *hechos* a las cláusulas de tamaño 1.

4.3.1. Criterios de selección de cláusulas

Se probaron tres tipos de criterios:

- Por tamaño: Este criterio refiere a la cantidad de literales intervinientes en una cláusula. La motivación de este criterio surge de la observación de que, cuanto más pequeña sea una cláusula, mayor será la porción del árbol de búsqueda que recorte. Recordemos que, en el contexto de nuestro problema, las cláusulas son disyunciones. Por lo tanto una cláusula que contiene n literales nos evita recorrer una $\frac{1}{2^n}$ parte del espacio de búsqueda. Esto se debe a que, de las 2^n valuaciones posibles restringidas a los n literales intervinientes, hay una de ellas –en particular aquella que hace que dicha cláusula sea falsa– que es descartada gracias a la existencia de esta cláusula aprendida. Por lo tanto ninguna valuación que contenga dicha valuación restringida será viable. Por ende, a menor tamaño de cláusula, mayor será el espacio de búsqueda recortado.

- Por actividad: En 2.2.2 se explicó a qué nos referimos por actividad de una cláusula. De ello se desprende que la actividad de una cláusula es un indicador de cuántas veces esa cláusula fue (parcialmente) responsable de que ocurra un conflicto. Es de esperar entonces que las cláusulas con mayor actividad sean más relevantes para evitar recorrer ciertas porciones del espacio de de búsqueda.

- Por *LBD*: También en 2.2.2 introdujimos la noción de *Literals Block Distance*. Elegimos utilizar esta medida porque se han reportado muy buenos resultados en *sat-solvers* secuenciales [2–4] utilizando criterios basados en ella.

Parámetros

En el caso de los criterios por tamaño, los mismos se generaron a partir de un parámetro que indica el máximo tamaño que puede terner una cláusula para ser admitida en la herencia. En este caso, todas las cláusulas cuyo tamaño sea menor o igual que el indicado por el parámetro pasarán a formar parte de la herencia.

Los criterios por *LBD* funcionan de la misma manera que los basados en tamaño. En este caso las cláusulas admitidas en la herencia serán aquellas cuyo *LBD* sea menor o igual al correspondiente parámetro.

Los criterios por actividad se generan a partir de tres parámetros. En primer lugar un parámetro binario que indica si se desean conservar las cláusulas más activas o las menos activas. En segundo lugar un parámetro que indica qué porcentaje de las cláusulas aprendidas del padre se desea conservar y por último un parámetro binario que indica si, además, se deben conservar todas aquellas cláusulas aprendidas cuyo tamaño sea 2 (cláusulas binarias).

La incorporación de las cláusulas binarias en los criterios por actividad surge del hecho de que el *sat-solver* secuencial utilizado las maneja de manera particular. Si bien recorta su base de datos de claúsulas aprendidas de acuerdo a la actividad, nunca descarta las cláusulas binarias.

Además de los tipos de criterios mencionados también se implementó la posibilidad de heredar los *hechos* aprendidos (o *facts*). La herencia de los *facts* se implementó de manera tal que pudiera tanto ser combinada con los distintos tipos de criterios como utilizada de forma exclusiva como criterio de herencia.

4.3.2. Resultados obtenidos

A continuación presentamos los resultados obtenidos. Los mismos se dividen en tres aspectos diferentes.

Mejora de la eficiencia por problema

En la Tabla 4.1 se presenta la comparación entre los resultados obtenidos con el *sat-solver* secuencial, los resultados obtenidos por nuestra herramienta sin herencia de cláusulas aprendidas y los resultados obtenidos por nuestra herramienta con herencia de cláusulas aprendidas.

Las primeras siete columnas poseen el exacto mismo significado que en la Tabla 3.2. En la columna “par. walltime w. inher. best crit.” consignamos

el mejor tiempo obtenido por nuestra herramienta con algún criterio de selección de cláusulas aprendidas –incluyendo el criterio nulo–.

La columna "speedup w. inher. best crit." representa la mejoría en el tiempo percibido obtenida por el mejor criterio. La misma es el resultado de la divisón

$$\frac{\text{seq. walltime}}{\text{par. walltime w. inher. best crit.}}$$

La columna "efficiency w. inher. best crit." es el resultado de la división

$$\frac{\text{speedup w. inher. best crit.}}{\#workers}$$

La columna "effic. mult." es el resultado de la división

$$\frac{\text{efficiency w. inher. best crit.}}{\text{efficiency without inheritance}}$$

En casi todos los casos se obtuvo una disminución en los tiempos de cómputo con respecto a la versión sin herencia de cláusulas aprendidas. Se observa una única excepción, el problema Routing con *scope* 8. En este caso el mejor tiempo fue el obtenido por el criterio nulo. Es decir por la herramienta sin herencia de cláusulas.

Se observa también una tendencia a que la mejora obtenida por la incorporación de herencia de cláusulas crezca considerablemente a medida que crece el tamaño del problema.

Los resultados aquí expuestos permiten verificar la hipotésis de la cual partimos: efectivamente la incorporación de herencia de cláusulas aprendidas permitió mitigar los efectos negativos del retrabajo incrementando la eficiencia en el uso del *hardware* disponible.

Sin embargo, en ausencia de un oráculo que permita determinar *a priori* cuál de los criterios es más conveniente en cada ocasión, la implementación de herencia de cláusulas aprendidas en una herramienta usable en la práctica depende de nuestra capacidad para determinar cuál de los criterios es el que mejor se comporta en el caso promedio. Este problema se ataca en las siguientes secciones.

Multiplicadores de *speedup* por cada criterio

En la Tabla 4.2 mostramos los resultados obtenidos para cada criterio. Llamaremos *speedup multiplier* a la razón que existe entre el *speedup* obtenido aplicando herencia de cláusulas aprendidas y el *speedup* obtenido sin herencia.

La columna "worst speedup multiplier" consigna el multiplicador obtenido por el criterio correspondiente para aquel problema en el que obtuvo el peor multiplicador. La columna "best speedup multiplier" consigna el multiplicador obtenido por el criterio para aquel problema en el que obtuvo el mejor multiplicador.

problem		reps.	seq. walltime	par. walltime without inheritance	speedup without inheritance	efficiency without inheritance	par. walltime w. inher. best crit.	speedup w. inher. best crit.	efficiency w. inher. best crit.	eff. mult.
Routing	8	7	308.26	60.46	5.10x	0.08	60.46	5.10x	0.08	1.00
Routing	9	6	76168.16	407.34	186.99x	2.92	387.44	196.60x	3.07	1.05
Routing	*10	5	>1209600.00	5046.84	>239.67x	>3.74	3213.59	>376.40x	>5.88	1.57
Closure	11	14	749.65	291.28	2.57x	0.04	150.30	4.99x	0.08	1.94
Closure	12	5	3983.36	1914.45	2.08x	0.03	266.68	14.94x	0.23	7.18
Closure	13	5	16261.35	4362.61	3.73x	0.06	645.44	25.19x	0.39	6.76
GC Sound.	9	10	217.31	200.85	1.08x	0.02	168.04	1.29x	0.02	1.20
GC Sound.	10	7	2855.30	1376.89	2.07x	0.03	592.54	4.82x	0.08	2.32
GC Compl.	8	7	180.25	61.50	2.93x	0.05	36.43	4.95x	0.08	1.69
GC Compl.	9	5	18643.06	1825.60	10.21x	0.16	390.48	47.74x	0.75	4.68

*: Se ejecutó por 14 días sin que el *sat-solver* secuencial consiga un resultado

Tab. 4.1: Tiempo de ejecución (en segundos) distribuido sin herencia vs. distribuido con herencia de cláusulas aprendidas

Por último, la columna "avg speedup multiplier" consigna el multiplicador promedio obtenido por el criterio sobre la ejecución de la totalidad de los problemas. Este valor se calculó comparando la suma de los tiempos promedio de todos los problemas –sin herencia de cláusulas aprendidas– contra la suma de los tiempos promedio de todos los problemas utilizando el criterio correspondiente. Hay que tener en cuenta entonces que las mejoras obtenidas en problemas más grandes pesarán más que las mejoras obtenidas en problemas más chicos.

En cada columna se señalaron con color verde los mejores resultados obtenidos en dicha columna y con color rojo los peores resultados obtenidos.

En estos resultados se observa que no hubo ningún criterio que sea siempre mejor o igual que el criterio nulo. A su vez, el criterio que consiste en heredar únicamente los hechos aprendidos –cláusulas de tamaño 1– resultó ser el que mejor "worst speedup multiplier" obtuvo. Es decir que este criterio resultó ser el más seguro en tanto que en el peor caso producirá un resultado un 19 % peor que si no hubiéramos utilizado ningún tipo de herencia.

También se puede ver que hay más de un criterio que presenta una mejora promedio (es decir sobre todos los problemas) superior a $2x$, siendo el máximo obtenido $2{,}61x$. Es decir que de haber utilizado dicho criterio habríamos tenido que invertir el 38 % del tiempo que tomó resolver los mismos problemas sin herencia de cláusulas.

En los criterios por tamaño se observa que el "avg. speedup multiplier" crece estrictamente a medida que crece la cantidad de literales permitida. Esto ocurre tanto en el caso en el que se incluyen los *facts* como en el que no. En casi todos estos casos, los criterios que incluyen los *facts* reportan mejores resultados para esta columna que los que no, aunque esta tendencia se revierte para el caso de tamaño 6.

En los criterios por actividad se observa que aquellos que conservan las cláusulas binarias presentan un mejor "avg. speedup multiplier" que sus contrapartes que no lo hacen. Curiosamente no hay evidencia que permita concluir que es mejor conservar las cláusulas más activas en lugar de las menos activas.

Por último observamos que existen criterios que presentan mucha varianza. Por ejemplo, el criterio que obtuvo el mejor *speedup* ($7{,}18x$) para algún problema obtuvo como peor *speedup* un valor de $0{,}58x$, muy cercano al peor de todos.

Sobre la validez de los datos

Es importante señalar que en la Tabla 4.2 no aparecen absolutamente todos los criterios. El motivo de esto es que no todos los criterios pudieron ser probados con todos los problemas. Como se verá más adelante, algunos criterios resultaron claramente perjudiciales. Para el momento en que lanzamos las ejecuciones de algunos problemas –Completeness *scope* 8 y

criterion	worst speedup multiplier	best speedup multiplier	avg speedup multiplier
lbdle2.+facts	0.68	4.91	1.97
lbdle2.-facts	0.71	3.84	1.24
lbdle3.+facts	0.78	5.16	1.41
lbdle3.-facts	0.75	5.71	1.69
lbdle4.+facts	0.64	6.80	1.68
lbdle4.-facts	0.79	5.83	1.83
lbdle5.+facts	0.69	6.74	1.64
lbdle5.-facts	0.77	5.25	2.14
lbdle6.+facts	0.76	5.79	1.95
lbdle6.-facts	0.70	6.08	1.94
onlyfacts	0.84	3.23	1.45
perc=0.01.+facts.+binary.+active	0.55	3.83	2.12
perc=0.01.+facts.+binary.-active	0.77	3.78	1.92
perc=0.01.+facts.-binary.+active	0.64	3.64	1.19
perc=0.01.+facts.-binary.-active	0.65	3.50	1.39
perc=0.01.-facts.+binary.+active	0.74	2.73	1.91
perc=0.01.-facts.+binary.-active	0.69	2.38	1.30
perc=0.01.-facts.-binary.+active	0.74	1.68	1.22
perc=0.01.-facts.-binary.-active	0.71	2.22	1.22
sizele2.+facts	0.75	3.50	1.57
sizele2.-facts	0.81	3.32	1.19
sizele3.+facts	0.55	5.40	1.63
sizele3.-facts	0.75	3.87	1.39
sizele4.+facts	0.74	6.24	1.91
sizele4.-facts	0.72	4.21	1.81
sizele5.+facts	0.64	6.14	1.92
sizele5.-facts	0.58	7.18	1.90
sizele6.+facts	0.72	5.08	2.17
sizele6.-facts	0.79	5.69	2.61

Tab. 4.2: Mejoras obtenidas mediante la implementación de herencia de cláusulas aprendidas. Resultados por criterio.

Completeness *scope* 9– ya estaba claro que determinados criterios no iban a reportar mejoras y que su ejecución insumiría una importante cantidad de tiempo. Por estos motivos optamos por obviar estas combinaciones de problema/criterio.

Además, los datos presentados aquí no incluyen los resultados obtenidos para los *scopes* más grandes de los problemas Routing y Closure –10 y 13 respectivamente–. En estos *scopes* dichos problemas insumen tanto tiempo que sólo nos fue posible probar un subconjunto mínimo de criterios formado por aquellos que reportaron mejores resultados en los *scopes* anteriores.

La inclusión en la Tabla 4.2 de casos que sólo hayan sido probados con algunos criterios podría resultar injusta para con los demás criterios. Más aún si se trata de *scopes* que insumen mayor cantidad de tiempo.

Comportamiento de la herencia para *scopes* crecientes

A continuación presentamos la comparación de los resultados obtenidos por los distintos criterios para diferentes *scopes* de un mismo problema. Las Figuras 4.6 a 4.9 presentan los resultados de la siguiente manera.

La primera fila (coloreada siempre de amarillo) representa el resultado obtenido por el criterio nulo. Las filas subsiguientes representan los resultados obtenidos por el criterio indicado a la izquierda. El color de cada celda representa la comparación entre el resultado obtenido por ese criterio y el resultado obtenido por el criterio nulo para el correspondiente *scope*, de acuerdo a la escala indicada a la derecha de la figura. Así, el coloreo hacia el verde indica una mejora respecto del criterio nulo mientras que el coloreo hacia el rojo representa un peor desempeño.

En el eje horizontal se ubican de manera creciente los distintos *scopes* probados. El color gris indica la ausencia de resultados para dicha combinación de *scope* y criterio. Como ya se aclaró, en el mayor *scope* de los problemas Routing y Closure nos limitamos a probar los criterios que habían reportado mejores resultados en los *scopes* anteriores, y en el caso del problema "GC Completeness" no se ejecutaron las peores alternativas.

En primer lugar observamos que existe una tendencia a que, a medida que aumenta el tamaño de un problema, la herencia de cláusulas obtiene mejores resultados. Por un lado se ve que, criterios que en un *scope* presentaron resultados positivos, en el *scope* siguiente presentan resultados aún mejores. Del mismo modo los resultados negativos presentados por algunos criterios se suavizan a medida que crece el *scope* del problema.

Consideramos que esto se debe a dos motivos. Por un lado existe un incremento en los costos asociados a la generación, transmisión y carga de las tareas cuando las mismas incluyen cláusulas aprendidas. Estos costos no crecen tan rápidamente como el costo de análisis a medida que crece el *scope*. Por lo tanto el *overhead* mencionado se va diluyendo a medida que el problema requiere mayor tiempo de cómputo. Además, como ya hemos

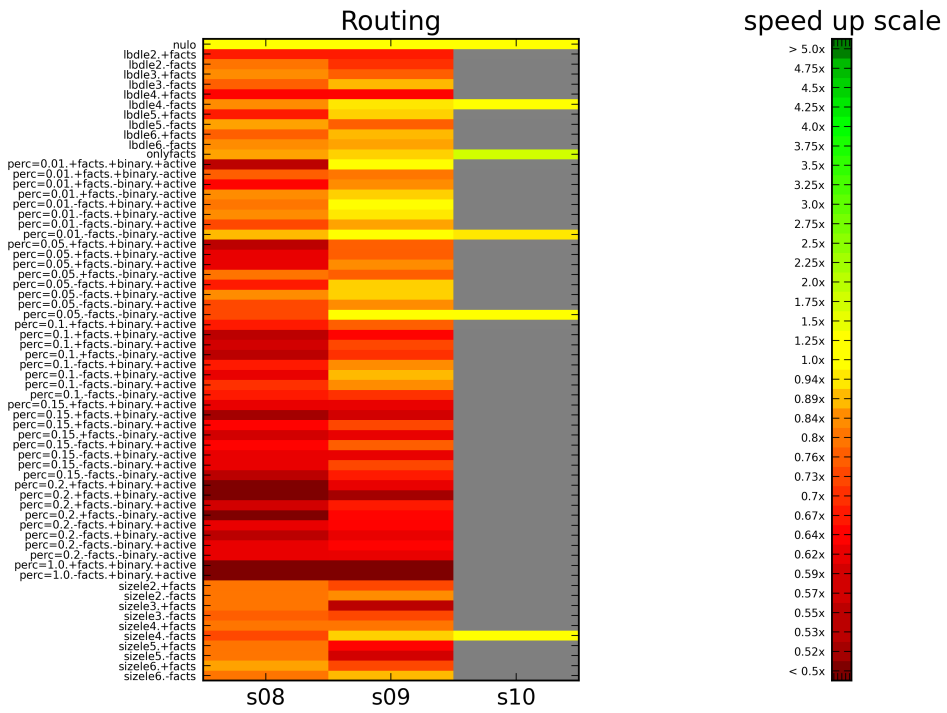


Fig. 4.6: Comparación de criterios de herencia para distintos *scopes* para el problema Routing

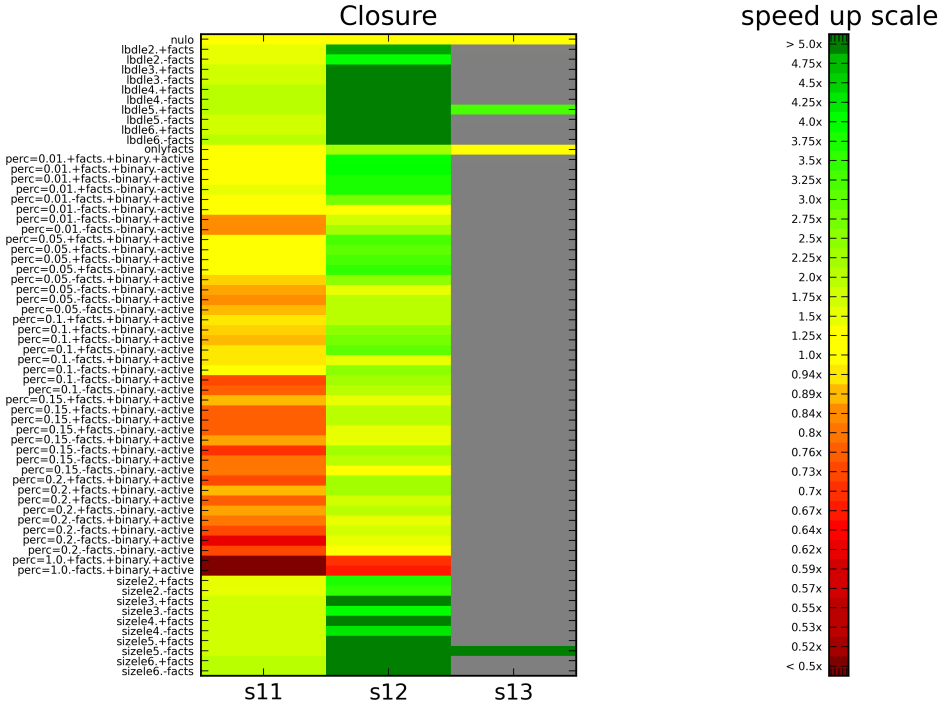


Fig. 4.7: Comparación de criterios de herencia para distintos *scopes* para el problema Closure

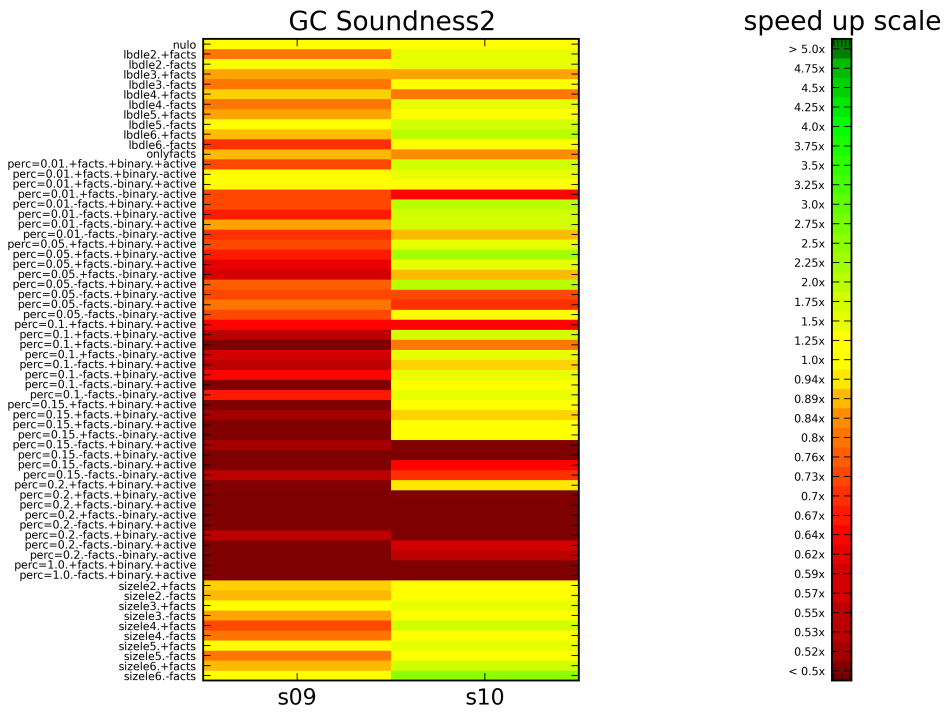


Fig. 4.8: Comparación de criterios de herencia para distintos *scopes* para el problema GC Soundness2

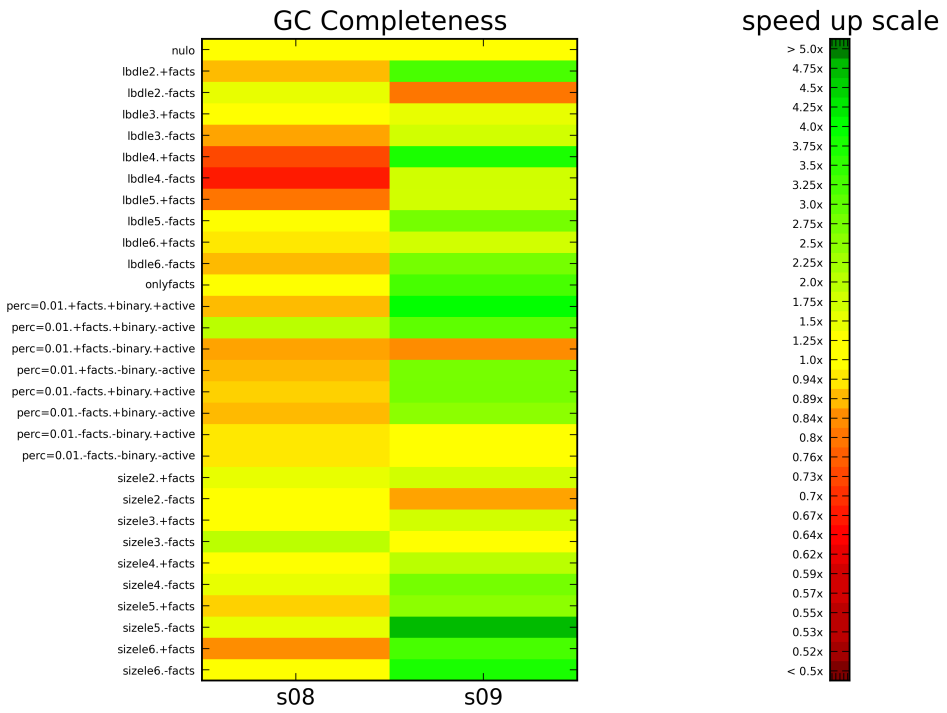


Fig. 4.9: Comparación de criterios de herencia para distintos *scopes* para el problema GC Completeness

comentado en la Sección 2.2.2, el uso de cláusulas aprendidas poda el espacio de búsqueda al costo de ir en detrimento de la velocidad de propagación (BCP). Así, su uso resulta particularmente necesario para resolver eficientemente ciertos problemas grandes (donde el espacio de búsqueda se vuelve inmenso, y cada poda es crucial para lograr agotarlo), pero a menudo es pernicioso para los más chicos (donde el costo extra en cada propagación se vuelve determinante).

También se observa que los resultados obtenidos por la herencia de cláusulas varían considerablemente de problema en problema, pero a la vez se mantienen coherentes a lo largo de los distintos *scopes* de cada problema particular. Como ejemplos paradigmáticos podemos destacar el problema Routing, en el que la técnica tiende a no conseguir mejoras sustanciales aún en los mejores casos, y el problema Closure, en el cual tiende a conseguir buenos *speedups* con la mayoría de los criterios. Esto sugiere la existencia de cualidades intrínsecas a cada uno de los problemas que los hacen inherentemente más (o menos) propensos a beneficiarse del enfoque propuesto. Los experimentos realizados no son suficientes para aspirar a una extracción de patrones comunes que permita aislar dichas cualidades.

Por último cabe destacar que, tal como intuíamos, el criterio que conserva la totalidad de las cláusulas aprendidas por el padre resulta consistentemente malo, tanto en términos absolutos (es siempre peor que no aprender nada) como relativos (es siempre la peor alternativa). Además, observamos que existe un subconjunto de los criterios probados que resulta consistentemente negativo. Nos referimos a aquellos en los que se conserva el 10 % (o más) de las cláusulas aprendidas por el padre. Es altamente probable que el mal desempeño de estas alternativas se deba a las razones expuestas en la Sección 4.1.

5. CONCLUSIONES Y TRABAJO FUTURO

5.1. Conclusiones

En el presente trabajo desarrollamos una herramienta de *sat-solving* paralelo y distribuido, orientada en particular a la verificación automática de propiedades sobre modelos formales de sistemas. Adoptamos un enfoque *divide-and-conquer*, basado en una técnica conocida, para el particionamiento recursivo de subproblemas difíciles. Lo implementamos sobre una arquitectura original, dando respuesta a los aspectos cruciales de escalabilidad, usabilidad y automatización. Identificamos un factor limitante para la eficiencia del enfoque de particionamiento. Propusimos una solución basada en herencia de cláusulas aprendidas. Diseñamos e implementamos diversos criterios y evaluamos su desempeño.

Las principales conclusiones respecto de la herramienta paralela y distribuida propiamente dicha son:

- Se lograron *speedups* considerables en la mayoría de los casos. Incluso en uno de los problemas se consiguieron *speedups* supralineales.
- La arquitectura resultó sólida y robusta, escalando sin inconvenientes al orden del centenar de *workers* sobre decenas de equipos.
- Para esa cantidad de *hardware*, los (escasos pero cruciales) componentes centralizados no presentaron evidencia alguna de saturación. Incluso para corridas de problemas grandes se observó en esos puntos amplia capacidad ociosa.
- Se identificaron correctamente los principales desafíos a resolver para lograr una operación 100 % automatizada. Se diseñaron soluciones pragmáticas, logrando convergencia y resultados satisfactorios para un banco de problemas con características disímiles, en distintos tamaños y grados de dificultad.

Las principales conclusiones respecto de la implementación y evaluación de cláusulas aprendidas en la herramienta distribuida son:

- La herencia de cláusulas aprendidas efectivamente permitió mejorar la eficiencia del análisis distribuido.
- El grado en que eso ocurre parece depender fuertemente del problema.
- Tanto para problemas donde la mejora es significativa como para aquellos donde no lo es, el grado de mejora tiende a aumentar junto con el tamaño de la instancia.

- Es posible implementar un esquema de reuso de cláusulas aprendidas sin atentar contra la escalabilidad del sistema.

- De los criterios de selección de cláusulas que evaluamos, el que logró mejores resultados globales fue aquel que retiene las cláusulas de tamaño menor o igual a 6, sin hechos.

- Las mejoras en la eficiencia logradas gracias a la herencia de cláusulas aprendidas, si bien son significativas, no resuelven por completo el problema del retrabajo. Hay casos en que la eficiencia obtenida, a pesar de mejorar visiblemente respecto de la versión sin learning, sigue distando considerablemente del ideal lineal.

5.2. Trabajo futuro

Mejoras en la estrategia automatizada de toma de decisiones

Uno de los aspectos más importantes a seguir refinando es la estrategia automática de toma de decisiones que hace posible la operación no supervisada del sistema. La versión actual logra evitar la mayor parte de los errores más evidentes, y así resulta capaz de llevar a buen puerto verificaciones breves, medianas y muy largas de propiedades muy diversas. Sin embargo, hay diversas capacidades que podrían ser mejoradas en futuras versiones.

Ante todo, los parámetros numéricos fijos restantes que se describieron en la sección 3.3.2 deberían ser reemplazados por mecanismos autoajustables. También podrían refinarse los algoritmos y las métricas utilizadas para los mecanismos que ya son autoajustables. En particular:

- Además de métricas directas (como la cantidad de problemas producidos y consumidos) y sus derivadas de primer orden (tasas de producción y consumo sobre tiempo), considerar también las de segundo orden (velocidad de estas últimas, i.e. aceleración de las primeras).

- Interpretar dichas métricas para determinar automáticamente la fase o etapa de la corrida en la que el sistema se encuentra (*ramp-up*, meseta, *ramp-down*, etc.), puesto que son varios los comportamientos que podrían beneficiarse variando en función de este conocimiento.

- Mejorar la estimación –y posiblemente la definición– de "progreso", observando con mayor cuidado las métricas extendidas antedichas.

- El nivel de agresividad al partir –cuántas variables levantar, cuántos subproblemas generar como máximo en cada momento dado– debería ser un parámetro dinámico que se autoajuste por retroalimentación en base al estado de las colas, la etapa actual de la corrida, etc.

- Utilizar información histórica para lograr algún grado de predecibilidad respecto de qué subproblemas tienen mayor probabilidad de ser los más difíciles de su camada, para así partirlos más temprano.

Evaluar la herramienta en *clusters* más grandes

Sería importante obtener acceso a (y cantidad suficiente de tiempo de cómputo en) grandes cantidades de *hardware*. En primer lugar, para medir cómo escala el rendimiento conforme se agregan más y más *workers*.

También sería interesante forzar la llegada al punto en que los componentes centralizados sí se conviertan en un factor limitante, para poder determinar si eso ocurre, por ejemplo, para cientos o para miles de *workers*, y para observar si surgen otros factores limitantes que no hayamos imaginado.

Diseñar y evaluar nuevos criterios de aprendizaje

Ante todo sería necesario correr más experimentos, sobre una mayor diversidad de casos de estudio y para *scopes* más grandes; esto último requiere abundante tiempo de cómputo.

Una ventaja en este sentido es el haber confirmado que ciertos criterios de aprendizaje no resultan beneficiosos. Dado que esos criterios solían ser los que más demoraban, su eliminación permitirá ahorrar tiempo de cómputo que podrá invertirse en la evaluación de nuevos criterios, diseñados en base a los que dieron mejores resultados.

En particular, en la Sección 4.3.2 mostramos que en los criterios por tamaño el rendimiento mejoraba a medida que se permitían cláusulas más grandes. Esto es explicable ya que los criterios con mayor tamaño incluyen a las cláusulas de los criterios con menor tamaño. Naturalmente, los criterios con mayor tamaño incluyen más cláusulas. Por lo ya explicado en las Secciones 2.2.2 y 4.1, lo esperable sería que al seguir aumentando el tamaño máximo de cláusula permitido, tarde o temprano la cantidad de cláusulas conservadas alcance un punto tal que los perjuicios por *overhead* superen los beneficios obtenidos por recorte del espacio de búsqueda (de modo que seguir aumentándolo se vuelva contraproducente). Sin embargo, en los experimentos realizados no se llegó a este punto. Consideramos que vale la pena seguir incrementando el tamaño permitido para poder determinar dónde está este punto de inflexión y cómo se comporta con respecto al incremento en *scope*.

Análisis de conflictos, *backjumping* y eliminación de subproblemas

Sería importante pensar cómo llevar a cabo, a nivel distribuido, algo equivalente al análisis de conflictos que realizan las herramientas secuenciales (véase sección 2.2.2) de modo que permita lograr podas análogas a las que en secuencial se logran por medio del *backtracking* no cronológico.

En la implementación distribuida, esto podría complementar el enfoque CDCL al permitirnos demostrar, durante o tras el análisis de un subproblema, que cierta cantidad de *otros* subproblemas (como ser hermanos, primos o incluso ancestros) son necesariamente insatisfacibles, y así poder eliminarlos de las colas de pendientes y/o abortar sus análisis en curso, de haberlos.

Interfaz más práctica para la toma de decisiones manual

El mecanismo actual, basado en una consola interactiva, resulta de suma utilidad para intervenciones esporádicas en momentos clave de una corrida por lo demás automática. También permite controlar una corrida en forma totalmente manual, aunque esto no resulta del todo práctico debido a la gran cantidad de comandos involucrados y su extensión.

Sería mucho más amigable contar con algún tipo de interfaz gráfica en la que las tareas puedan arrastrarse entre *workers*, partirse con un *clic* sin necesidad de tipear largos identificadores de subproblema, etc.

Bibliografía

[1] SAT-Race 2008. `http://baldur.iti.uka.de/sat-race-2008/`, Guangzhou, China, May 12–15 2008.

[2] SAT-Competition 2009. `http://www.satcompetition.org/2009/`, Swansea, Wales, UK, June 30 2009.

[3] SAT-Competition 2011. `http://www.satcompetition.org/2011/`, Ann Arbor, MI, USA, June 19–22 2011.

[4] SAT-Challenge 2012. `http://baldur.iti.kit.edu/SAT-Challenge-2012/`, Trento, Italy, June 17–20 2012.

[5] Pablo Abad, Nazareno Aguirre, Valeria Bengolea, Daniel Ciolek, Marcelo F. Frias, Juan Galeotti, Tom Maibaum, Mariano Moscato, Nicolás Rosner, and Ignacio Vissani. Tight bounds + incremental sat = better test generation under rich contracts. In *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation*, 2013.

[6] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[7] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[8] George Boole. *The mathematical analysis of logic. Being an essay towards a calculus of deductive reasoning.* Barclay & Macmillan and George Bell, Cambridge, UK and London, UK, 1847.

[9] Randall E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 8:677–691, 1986.

[10] J. R. Burch, Edmund M. Clarke, , Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Symposium on Logic in Computer Science '90*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.

[11] Max Böhm and Ewald Speckenmeyer. A fast parallel sat-solver - efficient workload balancing. In *Annals of Mathematics and Artificial Intelligence*, pages 40–0, 1996.

[12] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[13] Lisandro Dalcin. mpi4py. `https://code.google.com/p/mpi4py/`.

[14] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[15] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[16] Niklas Eén and Niklas Sörensson. An extensible sat solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Proceedings of 6th. International Conference on Theory and Applications of Satisfiability Testing, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, Santa Margherita Ligure, Italy, May 2003. Springer-Verlag.

[17] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, December 1982.

[18] Juan P. Galeotti, Nicolás Leandro Rosner, Carlos G. Lopez Pombo, and Marcelo F. Frias. Analysis of invariants for efficient bounded verification. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the 2010 International Symposium on Software Testing and Analysis*, pages 25–35, Trento, Italy, July 2010. ACM Press.

[19] Evgueni Goldberg and Yakov Novikov. BerkMin: A fast and robust sat-solver. In Carlos Delgado Kloos and Jose da Franca, editors, *Proceedings of the conference on Design, automation and test in Europe*, pages 142–149, Paris, France, March 2002. IEEE Computer Society.

[20] Youssef Hamadi and Lakhdar Sais. Manysat: a parallel sat solver. *JOURNAL ON SATISFIABILITY, BOOLEAN MODELING AND COMPUTATION (JSAT)*, 6, 2009.

[21] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[22] Norbert Manthey. Improving sat solvers using state-of-the-art techniques. Diploma thesis, Institut für Künstliche Intelligenz, Fakultät Informatik, Technische Universität Dresden, December 2010.

[23] João P. Marques-Silva and Karem A. Sakallah. GRASP – a new search algorithm for satisfiability. In *Proceedings of the 1996 International Conference in Computer Aided Design (ICCAD)*, pages 220–227, San Jose, CA, USA, November 1996. IEEE Computer Society and Association for the Computer Machinery.

[24] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[25] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In Jan Rabaey, editor, *Proceedings of the 38th. conference on Design automation*, pages 530–535, Las Vegas, Nevada, United States, 2001. ACM Press.

[26] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] M.G.C. Resende. Greedy randomized adaptive search procedures (GRASP). In C. Floudas and P.M. Pardalos, editors, *Encyclopedia of Optimization*, volume 2, pages 373–382. Kluwer Academic Publishers, 2001.

[28] T. Schubert, M. Lewis, and B. Becker. PaMiraXT: Parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:203–222, 2009.

[29] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer, Berlin, Heidelberg, 1983.

[30] Pamela Zave. Compositional binding in network domains. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proceedings of Formal Methods 2006: the 14th. International FME Symposium*, volume 4085 of *Lecture Notes in Computer Science*, pages 332–347, Hamilton, Canada, August 2006. Springer-Verlag.

[31] Hantao Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th. International Conference on Automated Deduction (CADE)*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275, Townsville, North Queensland, Australia, July 1997. Springer-Verlag.

[32] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. *Journal of Symbolic Computation*, 21(4-6):543–560, April 1996.

[33] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In Rolf Ernst, editor, *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 279–285, San Jose, CA, USA, November 2001. Association for the Computer Machinery and IEEE Computer Society, ACM Press.