eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Computación

Implementación de un razonador QBF utilizando una forma normal inductiva

Julen Beristain Oliden

Dirección
Javier Álvez Giménez

13 de septiembre de 2025

Agradecimientos

A Javier Álvez, Montserrat Hermo y Mikel Molina, por acompañarme en esta introducción a temas de computación teórica, tanto en las asignaturas de la carrera como en este proyecto de fin de grado.

A los siguientes profesores de la facultad, por transmitirme tantos conocimientos sobre la informática y por encender mi pasión por este campo durante estos cuatro años. Por orden alfabético:

Itziar Aldabe, Carlos Amuchástegui, Patxi Angulo, Rosa Arruabarrena, Myriam Arrue, Arkaitz Artetxe, Gorka Azkune, Idoia Bergés, Jose Javier Dolado, Elsa Fernández, Nestor Garay, Manuel María Graña, Julián Gutierrez, Mamen Hernández, Iñaki Inza, Ekaitz Jauregi, Edurne Larraza, Mikel Larrea, Oier Lopez de Lacalle, Begoña Losada, Joseba Makazaga, Iñigo Mendialdua, Alexander Mendiburu, Usue Mori, Iñaki Morlan, Maite Oronoz, Juanan Pereira, Jesus Maria Perez, Iñigo Perona, Jose Maria Rivadeneyra, Aitor Soroa, Maite Urretavizcaya, Imanol Usandizaga, Julian Zapiain, Irune Zubiaga.
* A pesar de mirar en mis apuntes y en la página web del profesado del grado, no he podido encontrar los nombres de la primera profesora de Estructura de Computadores en castellano del curso 20-21 (a la que tuvo que sustituir Nestor), del profesor de Estadística en inglés del curso 21-22 y del profesor de Introducción a Redes de Computadores en inglés del curso 21-22. Perdón por mi mala memoria, que no se debe a que me interesasen poco vuestras clases, sino todo lo contrario, guardo gratos recuerdos de ellas. Gracias y mucha suerte a vosotros también.

A mis compañeros de carrera, por todo lo que me habéis ayudado en los trabajos en grupo y por compartir las mismas ganas de aprender sobre informática.

A mi familia, en especial a mi aita, ama, amona eta anai-arreba Jokin eta Garazi, por apoyarme en todos los sentidos para poder estudiar lo que más he querido en cada momento hasta llegar a encontrar mi pasión.

A Dios, por guiarme y cuidarme hasta encontrar mi vocación, a pesar de haberlo ignorado durante casi toda mi vida.

Eskerrik asko bihotz-bihotzez denoi.

Objetivos de desarrollo sostenible

En este apartado analizamos cómo este proyecto contribuye a los Objetivos de Desarrollo Sostenible (ODS) establecidos en la Agenda 2030. En particular, nos centramos en algunos ODS y describimos de qué manera nuestro trabajo impacta en su cumplimiento.

Innovación en la resolución de problemas complejos (ODS 9: Industria, Innovación e Infraestructura). En este proyecto implementamos un razonador automático para fórmulas booleanas cuantificadas (QBF), una herramienta avanzada orientada al razonamiento y la verificación formal de software. Su desarrollo representa un aporte significativo en la creación de soluciones tecnológicas robustas y eficientes. Esto se alinea con el **objetivo 9**, que promueve la innovación y el fortalecimiento de la infraestructura mediante tecnologías de vanguardia que garanticen precisión y corrección, especialmente en el ámbito de los algoritmos y las ciencias de la computación.

Educación y formación en ciencias computacionales (ODS 4: Educación de Calidad). El razonamiento formal es una disciplina de creciente relevancia en el campo de la informática, y su enseñanza cobra cada vez mayor importancia en la formación académica. Herramientas como los demostradores SAT son utilizados en la enseñanza universitaria. Al desarrollar un razonador QBF correcto, este trabajo representa un avance relevante para el **objetivo 4**, que busca garantizar una educación de calidad.

Impacto en la eficiencia de sistemas (ODS 12: Producción y Consumo Responsables). El uso de herramientas de razonamiento automático permite diseñar algoritmos inteligentes que mejoran la eficiencia y robustez del software. Esto favorece un uso más responsable de los recursos tecnológicos y fomenta la adopción de prácticas sostenibles en el desarrollo del software. En este sentido, el proyecto contribuye al **objetivo 12**, orientado a garantizar modos de consumo y producción sostenibles.

Índice de contenidos

Índice de figuras

Índice de tablas

CAPÍTULO 1

Introducción

En el ámbito de la informática teórica y la ingeniería del software, es fundamental comprender la complejidad computacional de los problemas que se desean resolver. La complejidad computacional se refiere a cómo escala el gasto de ciertos recursos en función del tamaño de la entrada al programa que resuelve el problema. Normalmente, los dos recursos más importantes son el tiempo de cómputo y el espacio en memoria utilizados. Intuitivamente, vemos que no tiene el mismo coste ordenar una lista de números que calcular la mejor jugada de ajedrez dado un estado del tablero y el jugador al que le toca mover. Así, podemos clasificar estos problemas en distintas clases, según la dificultad que suponen: es decir, su complejidad computacional.

Esta clasificación permite: identificar qué problemas son abordables con los recursos disponibles; orientar el desarrollo de algoritmos eficientes y herramientas automáticas; etc. Cabe mencionar que saber que un problema es difícil también es útil, no solo para evitar investigaciones innecesarias que no van a dar frutos, sino para usar esa dificultad a nuestro favor. Por ejemplo, la criptografía moderna se basa en la dificultad de factorizar números enteros enormes con dos factores primos grandes. ¡Ojo!: la dificultad existe porque actualmente no contamos con un algoritmo eficiente que resuelva el problema, pero esto no quiere decir que en un futuro siga siendo así.

1.1. P vs NP

Uno de los pilares más estudiados en esta área es la distinción entre los problemas que pueden resolverse en tiempo polinómico (clase P [1], intuitivamente problemas sencillos que se pueden resolver incluso si la entrada es grande) y aquellos cuya solución puede verificarse en tiempo polinómico, pero no necesariamente hallarse de forma eficiente (clase NP [2], verificación sencilla dada una posible solución llamada certificado, pero resolución complicada). Esta diferencia teórica entre P y NP, y la pregunta sobre si en realidad ambas

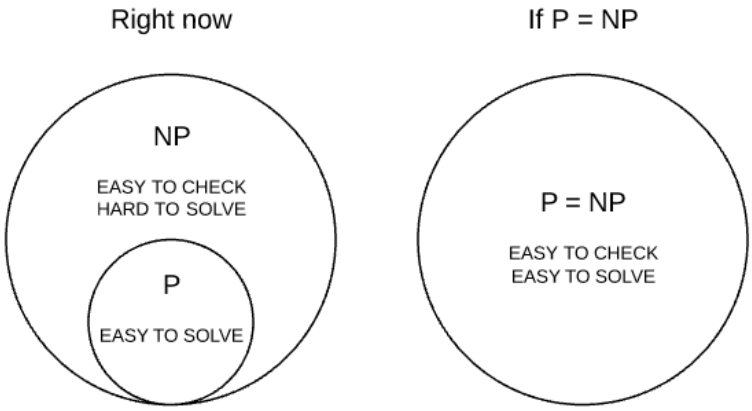

Figura 1.1: Situación actual de P vs NP [5].

clases coinciden, constituye uno de los siete Problemas del Milenio [3]. Es decir, no sabemos si todos los problemas que se pueden verificar con facilidad se pueden resolver también con facilidad o no. Si la respuesta fuera afirmativa, solo necesitaríamos encontrar ese algoritmo eficiente cuya existencia tendríamos asegurada. Si la respuesta fuera negativa, sabríamos que cierta clase de problemas fácilmente verificables no se pueden resolver de manera sencilla. Por ejemplo, la factorización de números primos que hemos comentado antes es fácilmente verificable, tan solo necesitamos multiplicar los factores primos y comprobar que coinciden con el número de la entrada. No obstante, como hemos adelantado antes, no contamos con un algoritmo eficiente que nos dé esos divisores primos de cualquier número de entrada. Por tanto, tenemos la certeza de que la factorización es un problema NP, pero como sigue abierta la incógnita sobre si P y NP son iguales, no sabemos si también es un problema de P. Con todo el esfuerzo invertido en la investigación de este problema en las últimas décadas no se ha podido demostrar que P=NP. Por ello, como nos dice la intuición, se piensa que realmente son dos conjuntos distintos [4].

Hemos dicho que si P es distinto de NP, habría algunos problemas fácilmente verificables pero difíciles de resolver; en términos de clases de complejidad, serían problemas que están en NP y no en P (esto es, en NP-P). Fijémonos que todo problema de P está en NP, ya que si puede resolverse en tiempo polinómico, se verifica igualmente en tiempo polinómico comparando el resultado obtenido con la solución correcta (es decir, la verificación consistiría en la propia ejecución del algoritmo). Entonces, los problemas interesantes de NP son aquellos que no están en P. En la Figura 1.1 podemos ver la situación actual de este problema del milenio.

1.2. Reducciones y completitud

Otros conceptos teóricos importantes son el de la completitud y las reducciones. Podemos usar estos conceptos para tratar de demostrar que P y NP son iguales (como hemos dicho, no ha habido suerte) de una manera sencilla de entender. En cambio, la demostración de

que P y NP son distintos requeriría: a) encontrar una clase de problemas en NP-P que, por su estructura, no pueden estar en P, lo cual es bastante más complicado; o b) alguna clase de demostración que todavía a nadie se le ha ocurrido. Una reducción de un problema A a otro B quiere decir que podemos tomar cualquier entrada w de A, transformarlo (de manera eficiente, en tiempo polinómico) mediante una función f a una entrada válida de B (denotada por $f(w)$), y todo esto de tal manera que la solución que obtenemos para B con $f(w)$ es la misma que la solución para A con w. De esta manera, si contamos con un algoritmo que resuelve B, podemos plantear un algoritmo que resuelve A: transformar la entrada w de A a $f(w)$, y pasarlo como entrada al algoritmo que resuelve B, devolviendo la salida de este último.

Una vez entendidas las reducciones, los problemas NP-completos son aquellos problemas de NP a los que todo problema de NP se pueden reducir. Teniendo un resolvedor de un problema NP-completo, podemos resolver cualquier problema de NP si tenemos la transformación de las entradas f. Así, como f tiene la condición de ser eficiente (es decir, de P, computable en tiempo polinómico), si encontramos un resolvedor eficiente (de P) para un problema NP-completo, tendríamos que todo problema de NP estaría en P; esto es, que P=NP. Las investigaciones para resolver este Problema del Milenio han tenido una vía muy clara siguiendo este razonamiento, pero no se ha encontrado ningún problema NP-completo que se resuelva con facilidad.

El problema canónico de la clase NP, el cual es el primer NP-completo que se demostró [1], es el problema SAT (*Boolean Satisfiability*, [6]). Este problema consiste en determinar si existe alguna asignación de valores booleanos que satisface una fórmula proposicional [7] (más en la Sección 3.1.1). Por ejemplo, dada la fórmula

$$\phi_1 = (x \vee \neg y) \wedge (\neg x \vee z)$$

un razonador SAT puede buscar una asignación para las variables que hace cierta a la fórmula (una posible asignación sería $x = True, z = True, y = True/False$).

Al ser el problema canónico de NP, hay un montón de problemas que se reducen a SAT. Eso ha impulsado el desarrollo de razonadores SAT muy eficientes [8] que sirven como base para resolver problemas de todo tipo: planificación y búsqueda en IA [9], verificación formal de software y hardware [10], teoría de grafos (coloreado, cliques máximos...) [11], puzles lógicos (sudokus, n-reinas...) [12], criptografía [13], optimización (*scheduling*...) [14], etc.

1.3. PSPACE y QBF

La dificultad no solo se puede medir en función del tiempo de cómputo, también en cuanto a otros recursos. El segundo recurso principal a tener en cuenta es el espacio en memoria utilizado al resolver el problema. Al igual que con el tiempo, encontramos diversas clases de

[1] Una vez encontrado un primer problema NP-completo con una demostración *ad-hoc* para dicho problema, una manera sencilla de encontrar otros problemas NP-completos es reduciendo el primero a otros problemas NP, que por tanto, serían a su vez NP-completos, dado que podríamos reducir cualquier problema NP al primero y después a estos nuevos problemas.

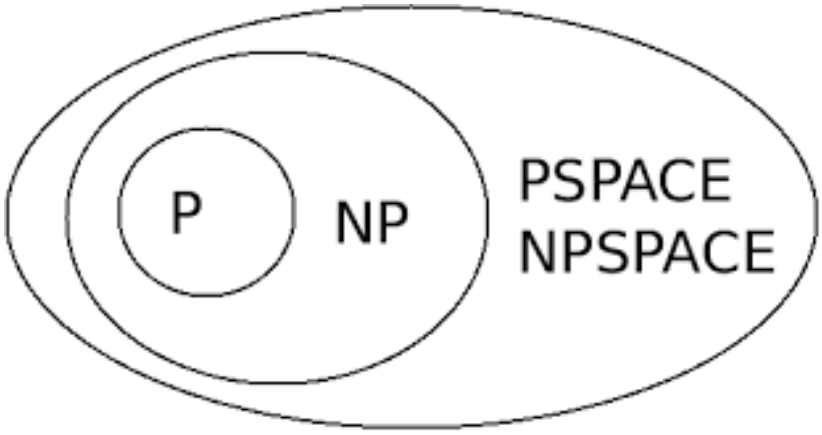

Figura 1.2: Clases de complejidad incorporando a PSPACE y NPSPACE [15].

complejidad, entre las que destacan PSPACE y NPSPACE. Análogamente a los dos anteriores, PSPACE contiene los problemas que se pueden resolver en un espacio de memoria de orden polinómico en relación al tamaño de la entrada; por otro lado, NPSPACE contiene problemas que se verifican en espacio polinómico. En el caso de estas dos clases, está demostrado que incluyen o representan el mismo conjunto de problemas, como apreciamos en la Figura 1.2. Es fácil de ver que si un problema se resuelve o verifica en tiempo polinómico (P o NP), como mucho dicho problema requiere un espacio polinómico, ya que el ordenador puede usar una cantidad de memoria limitada en cada unidad de tiempo.

PSPACE también tiene su problema canónico PSPACE-completo que, curiosamente, es una extensión de SAT: QBF (*Quantified Boolean Formulas*) [16], también conocido como TQBF (*True QBF*), QSAT (*Quantified SAT*) u otros nombres. El problema QBF extiende la lógica proposicional al permitir cuantificadores sobre las variables (existenciales $\exists$ y universales $\forall$). Esto nos permite representar propiedades más complejas, como juegos entre adversarios o razonamientos anidados. Por ejemplo:

$$\phi_2 = \forall x \exists y((x \vee y) \wedge (\neg x \vee \neg y))$$

Aquí se busca una estrategia para y que satisfaga la fórmula para todo valor de x. Como se ve en ϕ_2, se consideran fórmulas donde todas las variables están cuantificadas. Así, estas fórmulas que no tienen ninguna variable libre (no cuantificada) son lógicamente equivalentes a $True$ (satisfactible) o $False$ (insatisfactible). Además, en caso de ser satisfactible, podemos interpretar la fórmula como una función que por cada valor de una variable universalmente cuantificada devuelve un mapeo de las variables existencialmente cuantificadas. Es importante interpretar los cuantificadores en el orden correcto, de izquierda a derecha. Por ejemplo, ϕ_2 es satisfactible, ya que para ambos valores de x obtenemos $\exists y(y)$ o $\exists y(\neg y)$ como fórmula simplificada, los cuales son claramente satisfactibles (con $y = True$ e $y = False$ respectivamente). En cambio, un simple intercambio en el tipo de los cuantificadores

$$\phi_3 = \exists x \forall y((x \vee y) \wedge (\neg x \vee \neg y))$$

basta para que ϕ_3 no sea satisfactible, ya que para ambos valores de x obtenemos fórmulas con la misma matriz o núcleo de antes (y o $\neg y$) pero con el cuantificador universal ($\forall$) para la variable y. Esto es claramente falso dado que las asignaciones $y = False$ e $y = True$ hacen falsa la matriz, respectivamente, incumpliendo la cuantificación universal.

En SAT buscamos una asignación que hace cierta la fórmula. Así, la fórmula en sí es equivalente a la que cuantifica existencialmente toda variable de la fórmula original. Por eso, QBF es una generalización de SAT, donde el cuantificador universal y la intercalación entre los dos tipos de cuantificadores son la novedad. Esta generalización permite la construcción de fórmulas más expresivas, donde la respuesta no es una sencilla asignación sino una estrategia ante un entorno o problema interactivo, jerárquico o adversarial: juegos adversariales [17], *planning* en entornos no completamente conocidos [18], seguridad y análisis de protocolos [19], verificación formal de programas [20], etc.

CAPÍTULO 2

Planificación

En este capítulo definimos la planificación del proyecto, tanto las tareas a realizar como su seguimiento y control. La planificación nos permite acotar qué hitos vamos a realizar y prever las dificultades más probables que vayan a surgir en el desarrollo del trabajo, así como las respuestas que vamos a dar a las mismas.

2.1. Delimitación del proyecto

El proyecto se centrará en realizar una implementación de un razonador QBF usando una forma normal inductiva desde cero, con código propio, sin apenas hacer uso de librerías externas. Una vez realizado el razonador, llevaremos a cabo varias pruebas para valorar la eficacia y eficiencia del programa.

2.1.1. Objetivos

Objetivo general

El objetivo general de este proyecto es implementar un razonador QBF usando una forma normal inductiva que sea más eficiente que un razonador QBF básico.

Objetivos específicos

1. Conocer en más detalle el problema QBF del conjunto PSPACE (representación de los problemas, razonadores *state-of-the-art*, etc.). Asimismo, profundizar nuestro conocimiento sobre fórmulas lógicas de primer orden (definiciones, propiedades, operaciones, etc.), y en específico, sobre la forma normal inductiva.
2. Refinar nuestro conocimiento sobre los lenguajes de programación que vamos a utilizar para implementar el algoritmo de manera eficiente y modular.

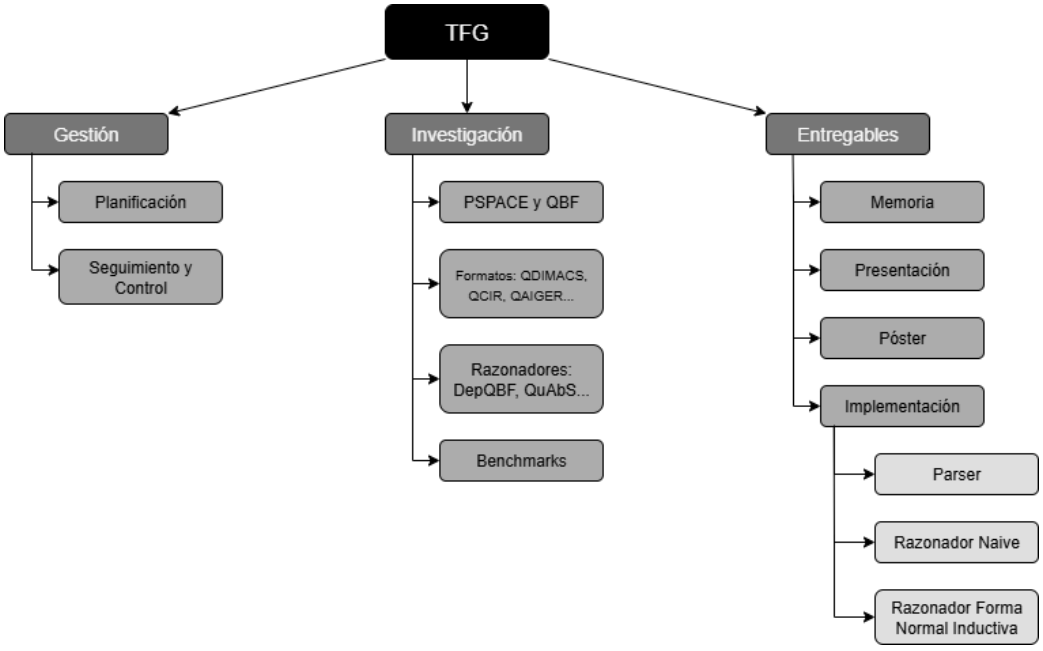

Figura 2.1: Esquema de decomposición de tareas.

3. Realizar la implementación del razonador con el algoritmo *naive*.

4. Realizar la implementación del razonador con el algoritmo no básico propuesto.

5. Evaluar la corrección y eficiencia de la implementación con instancias *benchmark*, realizando mejoras hasta conseguir superar considerablemente al algoritmo *naive*.

6. Posibles mejoras: tratar de refinar la implementación minimizando el uso de memoria y el tiempo de cómputo, probar a implementar con lenguajes de bajo nivel, etc. Volver a realizar pruebas para la versión final.

2.2. Planificación

En los siguientes apartados vamos a definir el plan que vamos a tomar como referencia a lo largo del proyecto.

2.2.1. Paquetes de trabajo

En la Figura 2.1 podemos ver la representación gráfica de las distintas tareas a realizar durante el proyecto.

Definimos así los paquetes de trabajo que aparecen en la Figura 2.1:

- Gestión (G): tareas relacionadas con la planificación y seguimiento del proyecto.
  - Planificación (PL): preparación inicial del proyecto.
  - Seguimiento y Control (SC): control de la concordancia entre el plan y la realización del proyecto. En caso de que hubiera desviaciones, controlar las consecuencias.
- Investigación (I): aprendizaje de herramientas y conceptos teóricos.
  - PSPACE y QBF: lecturas iniciales sobre la clasificación de problemas computacionales, el conjunto concreto PSPACE, y su problema canónico QBF (ver Capítulo 1 y Sección 3.1).
  - Formatos (Fs): aprender sobre los formatos más utilizados para guardar instancias de QBF en ficheros, con qué forma teórica corresponden, etc. (ver Sección 3.2).
  - Razonadores (Rs): buscar información sobre las implementaciones más famosas y eficientes de razonadores QBF (ver Sección 3.4).
  - Benchmarks (Bs): buscar instancias de problemas QBF sencillas y complicadas. Con las sencillas probaremos la corrección de los razonadores, y con las complicadas, la eficiencia espacial y temporal (ver Sección 7.1).
- Entregables (E): diseño, implementación y verificación de los entregables del proyecto.
  - Memoria (M): realización de la memoria del proyecto.
  - Presentación (Pr): realización de las diapositivas y los ensayos para la presentación final del proyecto.
  - Póster (Po): realización del póster que resume el proyecto.
  - Implementación (Imp): escribir el código de los razonadores QBF propuestos y probar su corrección y eficiencia.
    - Parser (Pa): implementación del programa que lee las instancias de los ficheros y devuelve la estructura apropiada para usar como entrada de los razonadores (ver Sección 6.2).
    - Razonador Naive (RN): implementación de un programa que sirve para resolver instancias QBF, pero que sigue una estrategia muy simple y no muy eficiente. Familiarización con el problema y demostración de que hace falta una aproximación más elaborada para obtener resultados eficientes (ver Capítulo 4 y Sección 6.3).
    - Razonador basado en la forma normal inductiva (RFNI): implementación del algoritmo central del proyecto, la cual transforma las instancias a una forma regular sobre la cual podemos definir operaciones eficientes (ver Capítulo 5 y Sección 6.4).

2.2.2. Diagrama de Gantt

Teniendo en cuenta los paquetes de trabajo definidos en la Subsección 2.2.1, en la Figura 2.2 encontramos el diagrama de Gantt con la distribución de las tareas a lo largo de las semanas que se propone inicialmente. Las semanas del diagrama son todas las consecutivas comprendidas entre el 3 de marzo y el 31 de julio.

2.2.3. Estimación de tiempos

Teniendo en cuenta que contamos con unas 300 horas para desarrollar el trabajo, en la Tabla 2.1 se muestra la distribución de horas que planteo inicialmente.

2.2.4. Análisis de riesgos

La Tabla 2.2 describe los riesgos que pueden darse en el proyecto:

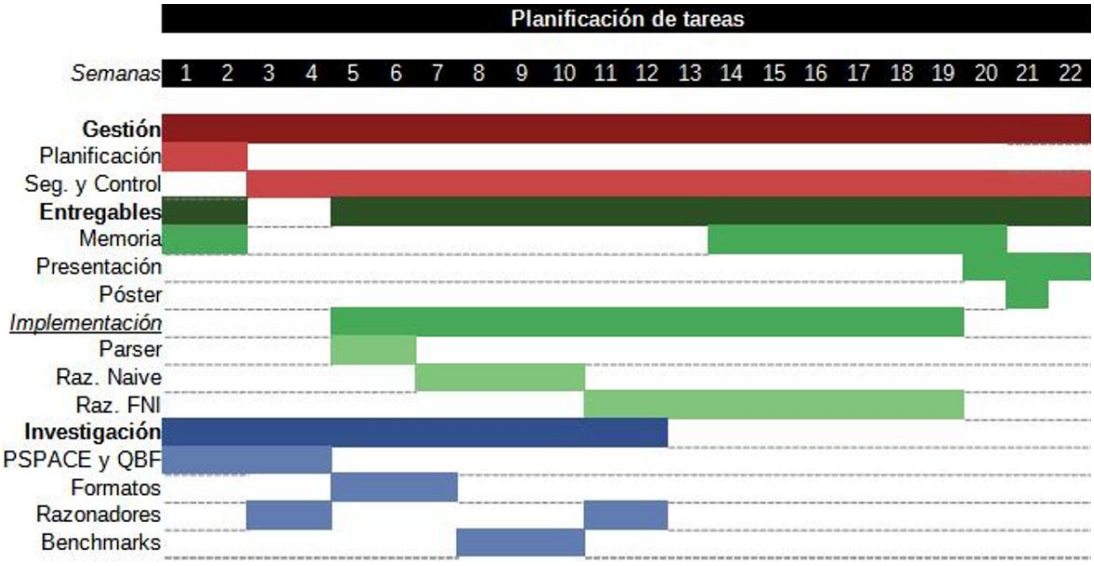

Figura 2.2: Diagrama de Gantt.

Paquetes de trabajo			Estimadas (h)
Gestión	*Planificación*		10
	Seguimiento y Control		20
Investigación	PSPACE y QBF		25
	Formatos: QDIMACS, QCIR...		15
	Razonadores: DepQBF, QuAbS...		20
	Benchmarks		20
Entregables	Memoria		60
	Presentación		20
	Póster		10
	Implementación	*Parser*	20
		Razonador Naive	30
		Razonador FNI	50
Total			**300**

Tabla 2.1: Estimación de tiempos.

Código	Descripción	Probabilidad	Impacto	Respuesta
R1	Planificación inicial optimista	Probable	Alto	Mitigar replanificando
R2	Extensiones no contempladas del alcance	Poco probable	Moderado	Mitigar priorizando objetivos
R3	Tareas no consideradas en la planificación inicial	Poco probable	Moderado	Mitigar recortando tareas superfluas
R4	Los conocimientos adquiridos sobre las nuevas herramientas no son suficientes para poder llevar a cabo la implementación	Poco probable	Muy Alto	Evitar aprendiendo más y pidiendo ayuda
R5	El nivel de productividad puede verse afectado debido a obligaciones externas	Poco probable	Alto	Aceptar y replanificar
R6	El servidor usado como repositorio extra y para realizar las pruebas del código puede dejar de funcionar	Poco probable	Alto	Conseguir otro repositorio extra y hacer las pruebas en local o en otro servidor
R7	Los servidores que contienen los *benchmark* pueden dejar de funcionar	Poco probable	Muy Alto	Usar *benchmarks* ya descargados y conseguir otros

Tabla 2.2: Riesgos del proyecto.

2.2.5. Tecnologías de la información

Para finalizar, vamos a explicar los programas y sistemas que se emplearán para el control de versiones del código, el desarrollo de la documentación y las comunicaciones entre el tutor y el estudiante.

Sistema de información

- Git: vamos a usar esta herramienta para el control de versiones del desarrollo local del código.
- Overleaf: vamos a usar esta herramienta *online* para el desarrollo de la memoria y para compartir los documentos donde se detallan los algoritmos a implementar.
- Google Drive: usaremos este sistema de almacenamiento en la nube para guardar otro tipo de archivos no entregables; por ejemplo, recopilación de enlaces a los archivos benchmarks, resúmenes de información encontrada en la investigación previa a la implementación, documentos de seguimiento y control, etc.

Comunicaciones

- Correo electrónico: vamos a usar el correo electrónico oficial de la Universidad para notificar los cambios en el horario de las reuniones o demás notas que no se hayan podido comentar en las propias reuniones. Para usar el correo de la universidad usaremos la versión *online* de Outlook y Cisco AnyConnect (para la autenticación de dos factores [21], necesaria para el uso del correo oficial como estudiante fuera de las redes de la universidad).
- Reuniones: principalmente se harán de manera presencial en un seminario de la Facultad o en el despacho del tutor, por norma general los viernes a las 15:30. Si alguna semana no fuera posible realizar la reunión, se aplazará o bien se realizará de manera telemática con alguna herramienta como Zoom, Discord, Google Meet...

CAPÍTULO 3

Preliminares

En este capítulo introducimos conceptos fundamentales sobre lógica proposicional que son constantemente referenciadas en los siguientes capítulos. Además, explicamos más aspectos de los problemas SAT y QBF, como formatos de las instancias, competiciones y resolvedores.

3.1. Bases de la lógica proposicional

3.1.1. Formulas proposicionales

Podemos definir una fórmula proposicional de manera inductiva usando los valores literales $True$ y $False$, variables booleanas de un conjunto numerable de variables V y las conectivas lógicas de negación ($\neg$), disyunción ($\vee$) y conjunción ($\wedge$):

- $False$ y $True$ son fórmulas proposicionales.
- Cada variable $v \in V$ es una fórmula (atómica).
- Dada una fórmula ϕ, $\neg\phi$ es una fórmula.
- Dadas dos fórmulas ϕ_1 y ϕ_2, $\phi_1 \vee \phi_2$ y $\phi_1 \wedge \phi_2$ son fórmulas.

Llamamos *literales positivos* a las fórmulas atómicas; respectivamente, llamamos *literales negativos* a las fórmulas atómicas negadas. Llamamos cláusula a una disyunción de literales (pueden ser tanto positivos como negativos), y término a una conjunción de literales. Una fórmula está en DNF (forma normal disyuntiva, [22]) si consiste en una disyunción de términos, mientras que está en CNF (forma normal conjuntiva, [23]) si consiste en una conjunción de cláusulas.

Dadas dos fórmulas ϕ y ψ, ψ es subfórmula de ϕ (denotado como $\phi[\psi]$) si ψ ocurre en ϕ. De esa forma, $\phi[\psi \leftarrow \gamma]$ denota la fórmula donde todas la apariciones de ψ en ϕ son sustituidos por γ.

Anotamos los conjuntos numerados de variables de la siguiente manera: $\{v_1, ..., v_n\} \subset V$, que lo abreviamos con $\overline{v}$. Además, dada la fórmula ϕ, $vars(\phi)$ expresa el conjunto de variables que aparecen en ϕ.

Definición 1. *Una asignación A' es una función que mapea variables lógicas a valores booleanos $True$ o $False$ (abreviando, T o F):*

$$A' : V \times \{F, T\}$$

Definición 2. *La aplicación de una asignación A' a una fórmula proposicional ϕ (denotado por $A(\phi)$) se define inductivamente de esta manera:*

- Si $\phi \in V$, $A(\phi) = \begin{cases} F & \text{si } A'(\phi) = T \\ T & \text{si } A'(\phi) = F \end{cases}$
- Si $\phi = \neg\psi$, $A(\phi) = \begin{cases} F & \text{si } A(\psi) = T \\ T & \text{si } A(\psi) = F \end{cases}$
- Si $\phi = \psi_1 \vee \psi_2,\ A(\phi) = \begin{cases} F & \text{si } A(\psi_1) = F \wedge A(\psi_2) = F \\ T & \text{e.c.c.} \end{cases}$
- Si $\phi = \psi_1 \wedge \psi_2,\ A(\phi) = \begin{cases} F & \text{si } A(\psi_1) = F \vee A(\psi_2) = F \\ T & \text{e.c.c.} \end{cases}$

Llamamos *satisfactible* a una fórmula proposicional ϕ si existe alguna asignación A' tal que $A(\phi) = T$. Además, si $A(\phi) = T$ para cualquier asignación A', entonces llamamos *válida* a ϕ.

El problema SAT que hemos visto en la Sección 1.2 trata precisamente de resolver si una fórmula proposicional (no cuantificada) de entrada es satisfactible o no.

3.1.2. Propiedades de equivalencia

Dos fórmulas proposicionales ϕ y ψ son llamados *equivalentes* (denotado por $\phi \equiv \psi$), si $A(\phi) = A(\psi)$ para toda asignación A'. Éstas son las principales propiedades de equivalencia (para toda fórmula lógica ϕ, ψ y γ):

- Identidad:

$$\phi \vee F \equiv \phi \tag{1}$$
$$\phi \wedge T \equiv \phi \tag{2}$$

- Idempotencia:

$$\phi \vee \phi \equiv \phi \tag{3}$$
$$\phi \wedge \phi \equiv \phi \tag{4}$$

- Dominación:

$$\phi \vee T \equiv T \tag{5}$$
$$\phi \wedge F \equiv F \tag{6}$$

- Conmutatividad:

$$\phi \vee \psi \equiv \psi \vee \phi \tag{7}$$
$$\phi \wedge \psi \equiv \psi \wedge \phi \tag{8}$$

- Asociatividad:

$$(\phi \vee \psi) \vee \gamma \equiv \phi \vee (\psi \vee \gamma) \tag{9}$$
$$(\phi \wedge \psi) \wedge \gamma \equiv \phi \wedge (\psi \wedge \gamma) \tag{10}$$

- Distribución:

$$\phi \vee (\psi \wedge \gamma) \ \equiv \ (\phi \vee \psi) \wedge (\phi \vee \gamma) \tag{11}$$
$$\phi \wedge (\psi \vee \gamma) \ \equiv \ (\phi \wedge \psi) \vee (\phi \wedge \gamma) \tag{12}$$

- Absorción:

$$\phi \vee (\phi \wedge \psi) \ \equiv \ \phi \tag{13}$$
$$\phi \wedge (\phi \vee \psi) \ \equiv \ \phi \tag{14}$$

- Contradicción:

$$\neg T \ \equiv \ F \tag{15}$$
$$\phi \wedge \neg\phi \ \equiv \ F \tag{16}$$

- Tautología:

$$\neg F \ \equiv \ T \tag{17}$$
$$\phi \vee \neg\phi \ \equiv \ T \tag{18}$$

- Doble negación:

$$\neg\neg\phi \ \equiv \ \phi \tag{19}$$

- Leyes de De Morgan:

$$\neg(\phi \vee \psi) \ \equiv \ (\neg\phi) \wedge (\neg\psi) \tag{20}$$
$$\neg(\phi \wedge \psi) \ \equiv \ (\neg\phi) \vee (\neg\psi) \tag{21}$$

3.1.3. Cuantificadores

Una fórmula proposicional *cuantificada* ϕ es una fórmula proposicional donde las variables $v \in vars(\phi)$ pueden estar cuantificadas mediante el cuantificador universal ($\forall$) o el existencial ($\exists$). Cada variable se cuantifica como mucho una vez. Así, construimos inductivamente las fórmulas proposicionales cuantificadas mediante el conjunto de variables numeradas V, conectivas lógicas y cuantificadores de la siguiente manera:

- Toda fórmula proposicional ϕ es una fórmula proposicional cuantificada.
- Dada una variable $v \in V$ y una fórmula ϕ donde toda subfórmula ψ de ϕ ($\phi[\psi]$) cumple que $\psi \neq \exists v\ \gamma$ y $\psi \neq \forall v\ \gamma$ (es decir, de manera intuitiva, v no está cuantificada en ϕ), $\exists v\ \phi$ y $\forall v\ \phi$ son fórmulas proposicionales cuantificadas.

Dada una fórmula proposicional cuantificada ϕ y una variable $v \in vars(\phi)$, decimos que v está cuantificada en ϕ cuando $\phi[\psi]$ para fórmulas ψ y γ tales que $\psi = \exists v\ \gamma$ o $\psi = \forall v\ \gamma$; es decir, si hay alguna subfórmula que cuantifica a v. De la misma manera, una variable $u \in vars(\phi)$ es *libre* si no está cuantificada. Denotamos el subconjunto de variables libres de ϕ como $free(\phi)$.

Aparte de las propiedades de equivalencia de las fórmulas proposicionales (1 - 21), tenemos más propiedades para las fórmulas cuantificadas (para cualquier $\{v, w\} \subset V$ y toda fórmula proposicional cuantificada ϕ, ψ y γ donde $v \notin vars(\gamma)$):

- Conmutatividad:

$$\exists v\, \exists w\ \phi \ \equiv\ \exists w\, \exists v\ \phi \tag{22}$$
$$\forall v\, \forall w\ \phi \ \equiv\ \forall w\, \forall v\ \phi \tag{23}$$

- Distribución:

$$\exists v\ (\ \phi \vee \psi\ ) \ \equiv\ (\exists v\ \phi) \vee (\exists v\ \psi) \tag{24}$$
$$\forall v\ (\ \phi \vee \psi\ ) \ \equiv\ (\forall v\ \phi) \vee (\forall v\ \psi) \tag{25}$$

- Eliminación:

$$\exists v\ \gamma \ \equiv\ \gamma \tag{26}$$
$$\forall v\ \gamma \ \equiv\ \gamma \tag{27}$$

- Negación:

$$\neg\exists v\ \phi \ \equiv\ \forall v\ \neg\phi \tag{28}$$
$$\neg\forall v\ \phi \ \equiv\ \exists v\ \neg\phi \tag{29}$$

Dado un conjunto ordenado de variables $\overline{v} = \{v_1, ..., v_n\} \subset V$, podemos abreviar un bloque de cuantificadores de la forma $Qv_1...Qv_n$, donde $Q \in \{\exists, \forall\}$, como $Q\overline{v}$, o directamente $\exists\overline{v}$ o $\forall\overline{v}$. De la misma forma, podemos abreviar los bloques de cuantificadores $Q_1\overline{v_1} \ldots Q_m\overline{v_m}$ como $\overline{Q}\overline{v}$, donde $\overline{v} = \bigcup_{i=1}^{m} \overline{v_i}$.

Una fórmula proposicional cuantificada ϕ está en *forma normal prenexa* si sigue esta estructura: $\overline{Q}\overline{v}\psi$ para una cierta fórmula proposicional ψ (llamado matriz o núcleo de ϕ) tal que $\overline{v} \subseteq vars(\phi) = vars(\psi)$.

Una fórmula proposicional totalmente cuantificada, abreviado como fórmula QBF (*fully Quantified Boolean Formula*), es una fórmula donde todas las variables están cuantificadas. Esto es, $free(\phi) = \emptyset$ para toda fórmula QBF. Ésta es la semántica de las fórmulas QBF:

Definición 3. *El valor de verdad de una fórmula QBF ϕ se define recursivamente de la siguiente manera:*

- *Si $vars(\phi) = \emptyset$, entonces el valor de verdad de ϕ es T si y solo si $\phi \equiv T$.*
- *Si $\phi = \exists v\ \psi$ para una fórmula proposicional cuantificada ψ, entonces el valor de verdad de ϕ es T si y solo si el valor de verdad de $\phi[v \leftarrow F]$ o $\phi[v \leftarrow T]$ es T.*
- *Si $\phi = \forall v\ \psi$ para una fórmula proposicional cuantificada ψ, entonces el valor de verdad de ϕ es T si y solo si el valor de verdad de $\phi[v \leftarrow F]$ y $\phi[v \leftarrow T]$ es T.*

Las fórmulas $\phi[v \leftarrow F]$ y $\phi[v \leftarrow T]$ de la Definición 3 también son QBF, ya que las fórmulas originales $\phi = \exists v\ \psi$ y $\phi = \forall v\ \psi$ son QBF.

Siguiendo la convención para las fórmulas proposicionales, llamamos *satisfactible* a una fórmula QBF si su valor de verdad es T. Por el contrario, es insatisfactible si su valor de verdad es F.

Finalmente, si la fórmula QBF está también en forma normal prenexa y su matriz está en CNF, decimos que la fórmula está en CNF prenexa, abreviado como PCNF. Es decir, una fórmula en PCNF es una CNF que está totalmente cuantificada, donde todos los cuantificadores están a la izquierda de la fórmula, antes de que aparezca ningún literal; es decir, los bloques de cuantificadores son el prefijo de la matriz o núcleo de la fórmula. Al igual que DNF es la forma dual de CNF, PDNF es la forma dual de PCNF.

3.1.4. Eliminación de cuantificadores

La eliminación de cuantificadores es una técnica fundamental para resolver fórmulas QBF (es decir, para encontrar su valor de verdad). El resolvedor que planteamos en el Capítulo 5 se basa en esta idea. Este procedimiento emplea la Definición 3 del valor de verdad de fórmulas QBF, empezando desde los cuantificadores internos (los más cercanos a la matriz) hasta eliminar el más externo (el primero). También hace uso de las propiedades 26 y 27 de eliminación. La idea se basa en las siguientes equivalencias siguiendo la Definición 3:

$$\overline{Q}\overline{v}\ \exists v\ \phi \equiv \overline{Q}\overline{v}\ (\phi[v \leftarrow T]\ \vee\ \phi[v \leftarrow F])$$
$$\overline{Q}\overline{v}\ \forall v\ \phi \equiv \overline{Q}\overline{v}\ (\phi[v \leftarrow T]\ \wedge\ \phi[v \leftarrow F])$$

Como vemos, la variable v desaparece de las subfórmulas de la matriz ($\phi[v \leftarrow T/F]$) dado que todas sus apariciones son sustituidas por T o F. Así, empleando las propiedades 26 y 27 de eliminación, se elimina también la cuantificación de la variable v.

Ahora bien, antes de seguir aplicando la eliminación con la siguiente variable cuantificada más interna de $\overline{Q}\overline{v}$ (es decir, si $\overline{Q}\overline{v} = Q_1\overline{v_1} \dots Q_n\overline{v}_n$ y $\overline{v}_n = \{v_{n1}, \dots, v_{nm}\}$, sería la variable v_{nm}) tenemos la disyunción o conjunción de las dos subfórmulas mencionadas. Para que dicha operación sea eficiente (la matriz no explote en tamaño) y sencilla de implementar, es conveniente que la matriz original ϕ esté en DNF para el caso del cuantificador existencial, y en CNF para el caso del cuantificador universal. Ya que la asignación a una variable en una fórmula en DNF preserva su forma, y que en el caso del existencial tenemos disyunción entre las dos subfórmulas (que consisten en asignaciones al núcleo original en DNF) la fórmula resultante seguiría estando en DNF. Es totalmente dual el caso del cuantificador universal y las fórmulas en CNF.

Entonces, con ese supuesto, la eliminación de cuantificadores se simplifica mucho. En el caso del cuantificador existencial con matriz en DNF, en la subfórmula donde a v se le asigna T, aquellos términos (conjunciones de literales) donde aparece $\neg v$ desaparecen (porque F es el valor dominante de la conjunción, propiedad 6), mientras que en los términos que contienen a v pasan a tener un literal menos (ya que T es el valor identidad de la conjunción, propiedad 2). Los que no contenían la variable v siguen igual. Por otro lado, en la subfórmula donde a v se le asigna F es totalmente al revés por los mismos argumentos: se eliminan los términos donde aparece v y se eliminan los literales $\neg v$ de todas aquellos términos que lo contienen (y los que no contenían la variable v siguen igual). Así, al calcular la disyunción entre las dos subfórmulas, tenemos los mismos términos de ϕ pero eliminando toda aparición de v, tanto literales positivos como negativos. En el caso de los términos que no contenían a v, no se duplican por la propiedad 3 de idempotencia. El caso del cuantificador universal con matriz en CNF es totalmente dual, la eliminación de variables consiste simplemente en eliminar los literales de la variable cuantificada.

Como nota adicional, en el caso de las fórmulas con matriz en DNF, si al eliminar un literal obtenemos un término vacío, como el elemento neutro de la conjunción es T (propiedad 2), toda la fórmula es equivalente a T, dado que T es el elemento dominante de la disyunción (propiedad 5), y al pasar a tener un núcleo igual a T sin ninguna variable, por eliminación (propiedades 26 y 27) llegaríamos al valor de verdad T. El caso de las fórmulas con matriz en CNF con cláusulas vacías y el valor de verdad F es totalmente dual. De manera similar, si nos quedamos sin términos en una matriz en DNF, la fórmula será equivalente a F (propiedad 1). El caso de las matrices en CNF sin cláusulas es totalmente dual, siendo T el resultado (propiedad 2).

Por último, es importante considerar que la mayoría de las fórmulas QBF intercalan cuantificadores existenciales y universales. Por tanto, la dificultad pasa a estar en las conversiones

de CNF a DNF cuando terminamos de eliminar un bloque de cuantificador universal y la siguiente variable a eliminar está cuantificada existencialmente, y de la misma manera al tener que pasar de DNF a CNF al terminar de eliminar un bloque de cuantificador existencial y la siguiente variable a eliminar sea cuantificada universalmente. La clave para la eficiencia de un algoritmo basado en eliminación de cuantificadores está en manejar estas conversiones de manera eficiente o idear un algoritmo que no requiera conversiones completas y costosas. Hacer conversiones de CNF a DNF y viceversa requeriría aplicar las propiedades distributivas 11 y 12 muchas veces, o pasar por todos los modelos (asignaciones que hacen cierta a la fórmula) de la matriz en CNF (por ejemplo, con un resolvedor de SAT), lo cual puede explotar el tamaño de la fórmula.

3.2. Formas vs Formatos

Distinguimos ahora dos conceptos que a menudo se entremezclan. Es una distinción que es posible que no se encuentre en toda documentación que trate el tema. Los conceptos son la **forma** de una fórmula lógica y el **formato** del archivo que codifica esa fórmula lógica.

En primer lugar, con forma nos referimos a la estructura de la fórmula, cómo y dónde pueden aparecer los operadores de conjunción y disyunción, los cuantificadores y los literales (variables o variables negadas). Por ejemplo, ya hemos comentado las formas CNF y PCNF en las Secciones 3.1.1 y 3.1.3. Por poner unos casos, la fórmula ϕ_1 de la Sección 1.2 está en CNF, y las fórmulas ϕ_2 y ϕ_3 de la Sección 1.3 están en PCNF. En capítulos posteriores explicaremos otra forma, la forma normal inductiva (Sección 5.1), que es el centro de nuestra implementación.

En segundo lugar, con formato nos referimos a la estructura mediante la cual se codifica la fórmula. DIMACS [24] es el formato básico de las instancias de SAT, y su extensión cuantificada para instancias de QBF es QDIMACS [25]. Estos formatos solo permiten expresar fórmulas en forma CNF y PCNF, por eso muchas veces se mezclan forma y formato. Hay otros formatos más flexibles que permiten expresar fórmulas en cualquier forma como QCIR [26], y otras un poco menos intuitivas como QAIGER [27]. Ciertos resolvedores de QBF (Sección 3.4) se aprovechan de la estructura de estos formatos para hacer implementaciones más eficientes. En nuestro caso, nos hemos centrado exclusivamente en el formato básico QDIMACS.

Veamos un ejemplo de QDIMACS con la fórmula ϕ_2 de la Sección 1.3 (repetido antes del ejemplo por comodidad). Podemos tener comentarios con líneas que empiezan por 'c'. La línea de encabezado empieza con 'p', seguido de 'cnf' y dos números naturales: el número de variables y el número de cláusulas. Luego vienen los bloques de cuantificadores, 'a' para universal y 'e' para existencial, seguidos por los identificadores de las variables (números enteros mayores que 0) terminados en 0. Por último, tenemos las cláusulas como lista de literales terminadas en 0. Por el propio formato, implícitamente entre los literales de cada línea tenemos disyunción, y entre las cláusulas expresadas en distintas líneas tenemos conjunción. Por eso (Q)DIMACS solo sirve para expresar fórmulas en forma (P)CNF.

$$\phi_2 = \forall x \exists y((x \vee y) \wedge (\neg x \vee \neg y))$$

```
c Podemos tener comentarios, normalmente solo al principio
c x1 = x ; x2 = y
p cnf 2 2
a 1 0
e 2 0
1 2 0
-1 -2 0
```

Fichero 3.1: Ejemplo del fichero QDIMACS que codifica la fórmula ϕ_2.

3.3. Competiciones de SAT y QBF

Para impulsar la investigación y desarrollo de razonadores SAT y QBF se celebran competiciones donde investigadores de todo el mundo envían sus implementaciones y éstas se prueban en un entorno estandarizado; es decir, con unas instancias de los problemas (*benchmark*) iguales para todos, clasificados por ciertas características en común en varias categorías.

En cuanto a SAT, *SAT Competition* es una competición que viene celebrándose anualmente desde 2002 [1]. Las categorías típicas incluyen:

- Sin restricciones.
- Resolvedores ágiles con límites de tiempo pequeños.
- Resolvedores que devuelven certificados, útiles para que otros programas verifiquen la solución sin tener que ejecutar el resolvedor de nuevo:
  - De satisfacción, esto es, asignaciones que demuestran que la instancia es satisfactible.
  - De insatisfacción, pruebas lógicas que demuestran que ciertas cláusulas de la fórmula se simplifican a la cláusula vacía (equivalente a F).
- Computación paralela.
- Problemas incrementales o dinámicos.

Los *benchmark* provienen de:

- Problemas de verificación formal.
- Problemas generados automáticamente.

[1] https://satcompetition.github.io/

- Problemas combinatorios clásicos.
- Instancias industriales reales.

El formato de los mismos es DIMACS [24], que representa fórmulas en CNF (ver Sección 3.2). Por último, las métricas con las que se comparan los resolvedores son las siguientes:

- Número de instancias resueltas.
- Tiempo medio de resolución, normalmente con penalización (es decir, las instancias no resueltas valen dos veces el límite del tiempo establecido).
- Tiempo hasta la primera solución (si se da una).
- Número de fallos o errores.
- Tamaño de los certificados.

En cuanto a QBF, *QBFEval* [28] es su competición. Desgraciadamente, debido al gran coste computacional que requiere la resolución de instancias de QBF, no está tan en boga como SAT. La competición no mantiene una regularidad anual. Por ejemplo, en 2024 y 2025 no se celebraron competiciones (en agosto de 2025 hubo un workshop en Glasgow, Escocia [2]). Además, la página web oficial del evento está caída [3]. Afortunadamente, hay otras páginas (como las páginas de ediciones anuales concretas) donde los *benchmark* empleados están disponibles (por ejemplo, la del año 2023 [4]). Las categorías típicas incluyen: resolvedor de fórmulas en CNF prenexa; resolvedores que generan certificados; problemas reales; etc. La mayoría de los *benchmark* están en formato QDIMACS [25], una extensión a DIMACS que se limita a las fórmulas PCNF (ver Sección 3.2). Por último, tenemos métricas parecidas a las de SAT, haciendo mayor hincapié en la eficiencia en tiempo y memoria.

Como nota final, cabe mencionar que los certificados en el caso de los resolvedores QBF son mucho más complejos que con resolvedores SAT [29], y por tanto el tamaño de los certificados toma más importancia en dicha categoría. En nuestro caso, los resolvedores a implementar no devuelven certificados, por lo que dejamos a un lado estas complicaciones más avanzados.

3.4. Resolvedores QBF

Pese a no estar tan en auge como los resolvedores SAT, ya hay algunos resolvedores QBF bastante eficientes [30]. Se pueden clasificar según el tipo de algoritmo que usan para resolver las fórmulas. A grandes rasgos, tenemos los siguientes:

[2] https://qbf.pages.sai.jku.at/qbf25/
[3] *Not Found*: https://www.qbflib.org/qbfeval/
[4] https://qbf23.pages.sai.jku.at/gallery/

- Basados en búsqueda (QCDCL [31], *Quantified Conflict-Driven Clause Learning*): extensión de CDCL [32] para SAT con manejo de cuantificadores. Por ejemplo, DepQBF [5] [33], CAQE [6] [34], Cute [7] [35].
- Basados en expansión: elimina cuantificadores expandiendo todas las asignaciones de las variables cuantificadas universalmente, convirtiendo instancias QBF a SAT. Por ejemplo, RAReQS [8] [36], GhostQ [9] [37].
- Híbridos: combinan las dos aproximaciones anteriores. Por ejemplo, QuAbS [10] [38] (que utiliza el formato QCIR), CAQE (híbrido).
- Certificadores: aparte del método de resolución, devuelven certificados de satisfacción o insatisfacción. Por ejemplo, DepQBF (con certificados), CAQE, Rev-Qfun [11] [39].

De los comentados, DepQBF es el razonador con el que comparamos los resolvedores que implementamos en el proyecto.

[5] https://lonsing.github.io/depqbf/
[6] https://finkbeiner.groups.cispa.de/tools/caqe/
[7] https://www.ac.tuwien.ac.at/research/qute/
[8] https://sat.inesc-id.pt/ mikolas/sw/areqs/
[9] https://www.wklieber.com/ghostq/
[10] https://finkbeiner.groups.cispa.de/tools/quabs/
[11] https://github.com/MikolasJanota/qfun

CAPÍTULO 4

Algoritmo Naive

Para familiarizarnos con el problema QBF, hemos empezado por examinar el algoritmo más sencillo posible para plantear el resolvedor. Lo podemos encontrar en la página de Wikipedia dedicada al problema QBF [16], en el apartado *QBF Solvers: Naive*.

4.1. Definición del algoritmo naive

Dada una fórmula QBF

$$\overline{Q}\overline{v}\ \phi$$

donde $\overline{v} = \{v_1, \ldots, v_n\}$:

- Si $n = 0$, es decir, $\phi = T$ o bien $\phi = F$, devolvemos ϕ.
- Si $n > 0$, es decir, sí quedan variables y cuantificadores, tomamos la primera variable cuantificada y nos queda:

  $$Q_1 v_1 \overline{Q}' \overline{v}' \phi$$

  donde $\overline{v}' = \overline{v} \setminus \{v_1\}$. A continuación, simplificamos la fórmula sustituyendo a v_1 tanto por T como por F

  $$A = \overline{Q}' \overline{v}' \phi[v_1 \leftarrow T]$$

  $$B = \overline{Q}' \overline{v}' \phi[v_1 \leftarrow F]$$

  Entonces, se resuelven A y B. Si $Q_1 = \exists$, devolvemos $A \vee B$. En cambio, si $Q_1 = \forall$, devolvemos $A \wedge B$.

En cuanto al tiempo de computación, por cada variable en el problema original se hacen dos llamadas recursivas con un tamaño linealmente menor (solo una variable menos), lo cual significa que su complejidad es exponencial ($O(2^n)$).

En cuanto al espacio de memoria, cada llamada recursiva necesita guardar los resultados de los subproblemas A y B, aparte de plantear las fórmulas simplificadas asignando a la primera variable T o F. Cada llamada recursiva quita un cuantificador, por lo que la profundidad de la pila de llamadas es n (el número de variables de la fórmula). Considerando que trabajamos con matrices en CNF y que m es el número de cláusulas, cada registro de la pila de llamadas necesita un espacio lineal de orden $O(m \times n)$, dado que necesitamos guardar la fórmula de entrada (que cómo máximo tiene n variables y m cláusulas, que se van reduciendo conforme profundizamos en la recursión) y A y B son dos valores booleanos. Se puede usar el mismo bloque de cuantificadores (de orden $O(n)$) en todas las llamadas, fijándonos en la variable que toca en cada momento según la profundidad de llamada. Por lo tanto, el coste espacial es $O(n \times (m \times n) + n) = O(m \times n^2)$, lo que implica que QBF $\in$ PSPACE.

4.2. Ejemplos

Vamos a ver dos pequeños ejemplos de ejecución del algoritmo naive usando las fórmulas presentadas en la Sección 1.3:

$$\phi_2 = \forall x \exists y((x \vee y) \wedge (\neg x \vee \neg y))$$

- $(A1)$ $\phi_2[x \leftarrow T] = \exists y((T \vee y) \wedge (\neg T \vee \neg y)) = \exists y(\neg y) = \psi_{2,1}$
  - $(A1.A)$ $\psi_{2,1}[y \leftarrow T] = \neg T = F$
  - $(A1.B)$ $\psi_{2,1}[y \leftarrow F] = \neg F = T$
  - $(A1,\ \exists y)$: $\psi_{2,1} = F \vee T = T$
- $(B1)$ $\phi_2[x \leftarrow F] = \exists y((F \vee y) \wedge (\neg F \vee \neg y)) = \exists y(y) = \psi_{2,2}$
  - $(B1.A)$ $\psi_{2,2}[y \leftarrow T] = T$
  - $(B1.B)$ $\psi_{2,2}[y \leftarrow F] = F$
  - $(B1,\ \exists y)$: $\psi_{2,2} = T \vee F = T$
- $(\phi_2,\ \forall x)$: $\phi_2 = T \wedge T = T$

Por lo que ϕ_2 es satisfactible.

Ahora, con ϕ_3:

$$\phi_3 = \exists x \forall y((x \vee y) \wedge (\neg x \vee \neg y))$$

- $(A1)$ $\phi_3[x \leftarrow T] = \forall y((T \vee y) \wedge (\neg T \vee \neg y)) = \forall y(\neg y) = \psi_{3,1}$
  - $(A1.A)$ $\psi_{3,1}[y \leftarrow T] = \neg T = F$
  - $(A1.B)$ $\psi_{3,1}[y \leftarrow F] = \neg F = T$
  - $(A1,\ \forall y)$: $\psi_{3,1} = F \wedge T = F$
- $(B1)$ $\phi_3[x \leftarrow F] = \forall y((F \vee y) \wedge (\neg F \vee \neg y)) = \forall y(y) = \psi_{3,2}$
  - $(B1.A)$ $\psi_{2,2}[y \leftarrow T] = T$
  - $(B1.B)$ $\psi_{3,2}[y \leftarrow F] = F$
  - $(B1,\ \forall y)$: $\psi_{3,2} = T \wedge F = F$
- $(\phi_3,\ \exists x)$: $\phi_3 = F \vee F = F$

Por lo tanto, ϕ_3 es insatisfactible.

4.3. Variaciones del algoritmo naive

Este algoritmo *naive* puede tener variaciones. Por ejemplo, en vez de que la recursión se haga variable a variable, lo podemos hacer por bloques de cuantificadores, disminuyendo la profundidad de la recursión pero teniendo que iterar sobre todas las combinaciones de las variables cuantificadas en cada bloque y simplificando la fórmula con varias asignaciones a la vez, siendo el resultado la conjunción (en caso de bloque universal) o disyunción (en caso de bloque existencial) de todos los booleanos conseguidos a partir de las llamadas recursivas con las múltiples asignaciones. Otra simplificación es llamar a un resolvedor de SAT optimizado en el caso donde sólo quede un bloque existencial. También se puede usar un resolvedor de SAT en el caso de que sólo quede un bloque universal para conseguir todos los modelos de la matriz ϕ y ver si se obtienen 2^m modelos (donde $m = |vars(\phi)|$). Además, en cada llamada recursiva se puede llamar al resolvedor de SAT con la matriz de la fórmula, y en caso de que la propia matriz sea insatisfactible, la fórmula entera, incluyendo los cuantificadores, será también insatisfactible. Por último, se pueden evitar llamadas recursivas para calcular el valor de más asignaciones si conseguimos T en el caso del bloque existencial y F en el caso de bloque universal, aprovechando que son los elementos dominantes de la disyunción y conjunción (propiedades 5 y 6).

Como veremos en la Sección 6.3 del Capítulo 6, la versión que hemos implementado hace recursión bloque a bloque, utiliza el SAT solver por defecto de PySAT [1] en el caso en el que queda un único bloque existencial, y evita llamadas recursivas innecesarias comprobando si

[1] https://pysathq.github.io/

se ha obtenido el elemento dominante de la disyunción o conjunción (dependiendo de si el bloque actual es existencial o universal). No implementa ninguna otra idea de las variantes mencionadas.

CAPÍTULO 5

Algoritmo basado en la Forma Normal Inductiva

Una vez visto el algoritmo más sencillo posible, en este capítulo explicaremos el núcleo del algoritmo más sofisticado que planteamos en este trabajo. Primero, veremos en qué consiste la forma normal inductiva (FNI) en la que transformaremos las instancias del problema QBF. Segundo, describiremos operaciones booleanas entre fórmulas proposicionales en FNI. Tercero, analizaremos cómo se aplica la eliminación de cuantificadores a este tipo de fórmulas. Y por último, uniremos todas las piezas para plantear el algoritmo resolvedor de QBF completo.

5.1. Forma Normal Inductiva

La forma normal inductiva (FNI) es un tipo de fórmula proposicional cuantificada. En concreto, es un tipo de fórmula QBF en forma normal prenexa (ver Sección 3.1.3) que permite una representación más regular de la matriz. Antes de entrar con los bloques de cuantificadores, vamos a introducir las variantes regulares de DNF y CNF, que tendrán que cumplir las matrices de las fórmulas en FNI.

Comenzamos con la variante regular de la DNF.

Definición 4. *(DNF regular o R-DNF). Dado un conjunto ordenado $\overline{v} = \{v_1, ..., v_n\} \subset V$ de variables, una fórmula proposicional ϕ tal que $vars(\phi) = \overline{v}$ es una DNF regular, abreviado como R-DNF, si cumple alguna de estas dos condiciones:*

a) $n = 0$, *y por tanto,* $\overline{v} = \emptyset$ *y* $\phi = False$ *o* $\phi = True$.

b) $n > 0$ *y* ϕ *es de la forma*

$$(\neg v_n \wedge \phi_1) \vee (v_n \wedge \phi_2) \vee \phi_3$$

para tres fórmulas también en R-DNF ϕ_1, ϕ_2 y ϕ_3 tales que $vars(\phi_1) \cup vars(\phi_2) \cup vars(\phi_3) \subseteq \overline{v} \setminus \{v_n\}$ (es decir, las subfórmulas no contienen a v_n pero pueden contener el resto de variables de ϕ) y:

- *$\phi_3 \neq True$ (sino, $\phi = True$, porque éste es el elemento dominante de la disyunción, propiedad 5).*
- *$\phi_1 \neq True \vee \phi_2 \neq True$ (sino, $\phi = True$ por tautología, propiedad 18, y por dominación de $True$, propiedad 5, ya que tendríamos $\phi = \neg v_n \vee v_n \vee \phi_3$,).*
- *$\phi_1 \neq False \vee \phi_2 \neq False \vee \phi_3 \neq False$ (sino, $\phi = False$, ya que tendríamos disyunción de tres $False$ debido a que éste es el elemento dominante de la conjunción, propiedad 6).*

De forma similar, la variante regular de la CNF se define de la siguiente manera.

Definición 5. *(CNF regular o R-CNF). Dado un conjunto ordenado $\overline{v} = \{v_1, ..., v_n\} \subset V$ de variables, una fórmula proposicional ϕ tal que $vars(\phi) = \overline{v}$ es una CNF regular, abreviado como R-CNF, si cumple alguna de estas dos condiciones:*

a) $n = 0$, *y por tanto,* $\overline{v} = \emptyset$ *y* $\phi = False$ *o* $\phi = True$.

b) $n > 0$ *y* ϕ *es de la forma*

$$(\neg v_n \vee \phi_1) \wedge (v_n \vee \phi_2) \wedge \phi_3$$

para tres fórmulas también en R-CNF ϕ_1, ϕ_2 y ϕ_3 tales que $vars(\phi_1) \cup vars(\phi_2) \cup vars(\phi_3) \subseteq \overline{v} \setminus \{v_n\}$ (es decir, las subfórmulas no contienen a v_n pero pueden contener el resto de variables de ϕ) y:

- *$\phi_3 \neq False$ (sino, $\phi = False$, porque éste es el elemento dominante de la conjunción, propiedad 6).*
- *$\phi_1 \neq False \vee \phi_2 \neq False$ (sino, $\phi = False$ por contradicción, propiedad 16, y por dominación de $False$, propiedad 6, ya que tendríamos $\phi = \neg v_n \wedge v_n \wedge \phi_3$,).*
- *$\phi_1 \neq True \vee \phi_2 \neq True \vee \phi_3 \neq True$ (sino, $\phi = True$, ya que tendríamos conjunción de tres $True$ debido a que éste es el elemento dominante de la disyunción, propiedad 5).*

La negación de una fórmula R-DNF es una fórmula R-CNF y viceversa (demostrable por inducción y con las leyes de De Morgan 20 y 21).

Estas fórmulas regulares se pueden representar como árboles ternarios de profundidad n, con un nivel por cada variable. Esto es matizable, ya que no es necesario tener un árbol completo. Es decir, si tenemos una fórmula en R-CNF

$$\phi = (\neg v_n \vee \phi_1) \wedge (v_n \vee \phi_2) \wedge \phi_3$$

no es necesario que cada una de las subfórmulas ϕ_i, $1 \leq i \leq 3$, sea de la forma

$$\phi_i = (\neg v_{n-1} \vee \phi_{i,1}) \wedge (v_{n-1} \vee \phi_{i,2}) \wedge \phi_{i,3}$$

ni tampoco que sea $True$ o $False$. Esto es, puede que no contenga la siguiente variable v_{n-1}, sino otra variable como v_{n-2}, v_{n-3}, o directamente 1. Por tanto, estrictamente hablando, en el siguiente nivel habría distintas variables según la rama, no sólo v_{n-1}.

La transformación de fórmulas DNF/CNF a R-DNF/R-CNF (árbol ternario con $True$ y $False$ en las hojas e identificadores de variables en el resto de nodos) se puede hacer de una manera eficiente:

1. Dada cualquier fórmula proposicional ϕ tal que $vars(\phi) = \overline{v} = \{v_1, ..., v_n\}$, asumimos que $(\neg)v_i < (\neg)v_j$ para todo $1 \leq i < j \leq n$ (es decir, ordenamos los literales según su variable, sin fijarnos en su polaridad), y ordenamos cada uno de los términos (DNF) o cláusulas (CNF) en orden descendente.

2. Ordenamos ascendentemente los términos y cláusulas entre sí asumiendo adicionalmente que $\neg v_i < v_i$ para todo $1 \leq i \leq n$, es decir, dando mayor prioridad a los términos (DNF) o a las cláusulas (CNF) que contienen los literales con polaridad negativa.

3. Finalmente, por distribución inversa (propiedades 11 y 12), agrupamos todos los términos o cláusulas que empiezan por $\neg v_n$, por v_n y aquellos donde v_n no aparece (los tres subconjuntos están contiguos), y procedemos de la misma manera con la siguiente variable v_{n-1} con cada una de las tres agrupaciones (después de haberles eliminado v_n).

El coste temporal de la transformación anterior para una fórmula DNF/CNF con n variables y m términos o cláusulas es de orden $O(m\,(n \log n)) + O(n\,(m \log m)) + O(nm) = O(nm\,(\log n + \log m)$. La primera parte es simplemente ordenar m cláusulas independientemente. La segunda parte es ordenar las m cláusulas entre sí, teniendo en cuenta que la comparación entre cláusulas recorre todos los literales de la cláusula más pequeña (que en el peor de los casos sigue teniendo n literales). La tercera parte se deduce así: en cada llamada al paso 3, el coste principal es recorrer las m cláusulas y clasificarlas en tres grupos mirando el primer elemento, lo cual sería de orden $O(m)$. En las llamadas recursivas, en cada nivel del árbol de llamadas tenemos la misma cantidad de m cláusulas (repartidos entre los 3^{depth} registros de llamadas), por lo que el coste de cada nivel sigue siendo $O(m)$. Y como mucho, la profundidad de todo el árbol de llamadas será n, el número de variables, por lo que tenemos n niveles.

En cuanto al espacio en memoria, es complicado hacer una comparación *a priori* para todos los casos. En el caso de fórmulas en CNF con m cláusulas y n variables, el tamaño puede llegar a ser de $m \times n$ si todas las cláusulas tienen n literales (uno por variable). Además, con n variables podemos tener 3^n cláusulas distintas, teniendo en cuenta que cada variable puede no aparecer, aparecer como literal positivo o como literal negado. Por tanto, el tamaño de una fórmula en CNF es de orden $O(n \times 3^n)$. Pero hay que tener en cuenta que las cláusulas que no contienen ciertas variables tienen un tamaño menor a n, además de que en la práctica la

mayoría de las fórmulas tienen cláusulas de menor tamaño y una cantidad de cláusulas mucho menor al límite exponencial. Por otro lado, para una fórmula en R-CNF, que es equiparable a un árbol ternario con profundidad n (un nivel por cada variable), el número de nodos es de orden $O(3^n)$, teniendo en cuenta que podemos tener un árbol completo. Como con las CNF, en la práctica las R-CNF distan de ser árboles completos. Por tanto, teóricamente, podemos decir que las fórmulas en R-CNF son más compactas que las CNF, pero en la práctica no está nada claro. Veremos una pequeña comparación práctica en la Sección 7.6.

Vamos a ver un ejemplo de transformación de CNF a R-CNF con ϕ_4, una fórmula sencilla muy parecida a ϕ_1 de la Sección 1.2 (pero cambiando la variable z por otra aparición de y para que no quede un árbol tan profundo). Hemos puesto cada paso del algoritmo en una línea y los hemos numerado. El paso 3 es recursivo, por eso tiene varias líneas:

$$\phi_4 = (x \vee \neg y) \wedge (\neg x \vee y)$$

$$(1)\ \phi_4 \equiv (\neg y \vee x) \wedge (y \vee \neg x)$$

$$(2)\ \phi_4 \equiv (\neg y \vee x) \wedge (y \vee \neg x)$$

$$(3)\ \phi_{4,1} = x,\ \phi_{4,2} = \neg x,\ \phi_{4,3} = T$$

$$(3{,}1)\ \phi_{4,1} = x \implies \phi_{4,1,1} = T,\ \phi_{4,1,2} = F,\ \phi_{4,1,3} = T$$

$$(3{,}2)\ \phi_{4,2} = \neg x \implies \phi_{4,2,1} = F,\ \phi_{4,2,2} = T,\ \phi_{4,2,3} = T$$

Por lo tanto, la fórmula ϕ_4 queda de la siguiente manera en R-CNF:

$$\phi_4 = (\neg y \vee ((\neg x \vee T) \wedge (x \vee F) \wedge (T))) \wedge (y \vee ((\neg x \vee F) \wedge (x \vee T) \wedge T)) \wedge T$$

Vemos que en este pequeño ejemplo en particular, la CNF es mucho más sencilla y compacta que la R-CNF, a diferencia de la comparación teórica *a priori* de los peores casos. Simplificando los T y F vemos que recuperamos la fórmula inicial, tras cambiar el orden de los literales por la propiedad conmutativa de la disyunción (7):

$$\phi_1 = (\neg y \vee x) \wedge (y \vee \neg x)$$

Lo podemos visualizar como un árbol en la Figura 5.1.

Ahora sí, basándonos en las formas regulares, podemos definir la forma normal inductiva:

Definición 6. *(Forma Normal Inductiva, FNI). Una fórmula proposicional QBF en forma normal prenexa ϕ está en forma normal inductiva, abreviado como FNI, si cumple alguna de estas condiciones:*

a) $\phi = False$ *o* $\phi = True$.

b) $\phi = \overline{Q}\overline{v}\ \exists v\ \psi$ *para una fórmula* ψ *en R-DNF tal que* $vars(\psi) = \overline{v} \cup \{v\}$.

c) $\phi = \overline{Q}\overline{v}\ \forall v\ \psi$ *para una fórmula* ψ *en R-CNF tal que* $vars(\psi) = \overline{v} \cup \{v\}$.

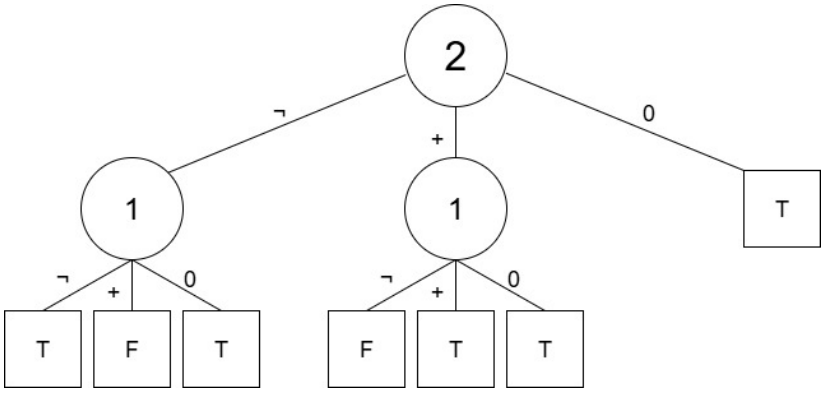

Figura 5.1: Visualización de la fórmula ϕ_4 en R-CNF.

La concordancia entre la forma regular de la matriz de la fórmula en FNI (R-DNF o R-CNF) y el tipo de cuantificador de la variable más interna (existencial o universal) deriva de la idea de eliminación de cuantificadores (Sección 3.1.4). Así, la eliminación de la variable más interna de una fórmula en FNI es más sencilla de realizar que cuando no hay esta concordancia.

Ya hemos visto cómo transformar una fórmula CNF o DNF a su forma regular. La transformación de una fórmula QBF a una QBF en forma normal prenexa equivalente es trivial por distribución inversa (propiedades 24 y 25). Después, la transformación de la matriz a CNF o DNF es sistemática ([23], Sección *Conversion to CNF*), pero no siempre trivial, ya que el tamaño de la fórmula puede crecer exponencialmente. Con estas herramientas, tenemos lo necesario para conseguir fórmulas en FNI a partir de cualquier fórmula cuantificada. En la práctica, vamos a tratar directamente con instancias QBF en forma normal prenexa con matrices en CNF, por lo que la transformación que realmente nos interesa es de CNF a su forma regular R-CNF.

Por ejemplo, tomando la fórmula ϕ_4 en R-CNF y cuantificando sus variables x e y tenemos una fórmula en FNI:

$$\phi_5 = \exists x \forall y \, \phi_4 = \exists x \forall y \, (\neg y \vee ((\neg x \vee T) \wedge (x \vee F) \wedge (T))) \wedge (y \vee ((\neg x \vee F) \wedge (x \vee T) \wedge T)) \wedge T$$

La variable más interna y tiene que estar cuantificada universalmente para que esté en FNI, de acuerdo a la Definición 6. En cambio, podríamos haber cuantificado x universalmente en vez de existencialmente, y seguiría estando en FNI.

5.2. Operaciones booleanas

En esta sección vamos a definir la conjunción y disyunción entre fórmulas en R-CNF. Utilizaremos estas operaciones en la Sección 5.3 para definir la eliminación de cuantificadores para el caso de fórmulas QBF en forma normal prenexa con matrices en R-CNF. Como en la implementación del resolvedor no utilizamos operaciones entre fórmulas en R-DNF, nos

centramos únicamente en las fórmulas en R-CNF. En el Apéndice A añadimos los teoremas duales explicando la disyunción y conjunción entre fórmulas en R-DNF.

Vamos a describir primero el caso de la conjunción de fórmulas en R-CNF.

Teorema 1. *(Conjunción de fórmulas R-CNF) La conjunción de dos fórmulas R-CNF se puede transformar a otra R-CNF equivalente.*

Demostración. Tenemos que demostrar que para dos fórmulas ϕ_1 y ϕ_2 en R-CNF donde $vars(\phi_1) \cup vars(\phi_2) = \{v_1, ..., v_n\} = \overline{v}$, existe una fórmula ϕ en R-CNF tal que $vars(\phi) = \overline{v}$ y $\phi \equiv \phi_1 \wedge \phi_2$.

Para ello, vamos a plantear una inducción sobre n.

Cuando $n = 0$ (ϕ_1 y ϕ_2 son T o F) los casos son triviales por identidad y dominación (propiedades 2 y 6).

Para el caso inductivo ($n > 0$), asumimos que, para cualquier par de fórmulas ψ_1 y ψ_2 en R-CNF, siempre existe una fórmula ψ en R-CNF tal que $\psi \equiv \psi_1 \wedge \psi_2$ con $vars(\psi) = vars(\psi_1) \cup vars(\psi_2) = \overline{v}$. Esta es nuestra hipótesis de inducción (H.I.).

Tenemos que demostrar que para cualquier par de fórmulas ϕ_1 y ϕ_2 en R-CNF, tales que $vars(\phi_1) \cup vars(\phi_2) = \overline{v} \cup \{v_{n+1}\}$, existe una fórmula ϕ en R-CNF tal que $vars(\phi) = \overline{v} \cup \{v_{n+1}\}$ y $\phi \equiv \phi_1 \wedge \phi_2$.

Consideramos que $vars(\phi_1) = \{v_{11}, ..., v_{1m_1}\}$ y $vars(\phi_2) = \{v_{21}, ..., v_{2m_2}\}$. Los casos donde $m_1 = 0$ o $m_2 = 0$ (alguna de las dos fórmulas es igual a T o F) son triviales por identidad y dominación (propiedades 2 y 6). Para el resto de casos y sin perder generalidad, por conmutatividad (propiedad 8), consideramos dos opciones: a) $v_{1m_1} = v_{2m_2} = v_n$ y b) $v_{1m_1} = v_n > v_{2m_2}$.

En el caso a), por la Definición 5 tenemos que

$$\phi_1 \wedge \phi_2 \;=\; ((\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi_{13}) \wedge ((\neg v_n \vee \phi_{21}) \wedge (v_n \vee \phi_{22}) \wedge \phi_{23})$$

para ciertas fórmulas en R-CNF tales que $\bigcup_{i=1}^{2} \bigcup_{j=1}^{3} vars(\phi_{ij}) = \overline{v}$. Por tanto:

$$\begin{aligned}
\phi_1 \wedge \phi_2 \quad &= \quad ((\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi_{13}) \wedge ((\neg v_n \vee \phi_{21}) \wedge (v_n \vee \phi_{22}) \wedge \phi_{23}) \\
&\overset{(8,10)}{\equiv} \quad ((\neg v_n \vee \phi_{11}) \wedge (\neg v_n \vee \phi_{21})) \wedge ((v_n \vee \phi_{12}) \wedge (v_n \vee \phi_{22})) \wedge (\phi_{13} \wedge \phi_{23}) \\
&\overset{(11)}{\equiv} \quad (\neg v_n \vee (\phi_{11} \wedge \phi_{21})) \wedge (v_n \vee (\phi_{12} \wedge \phi_{22})) \wedge (\phi_{13} \wedge \phi_{23}) \\
&\overset{(H.I.)}{\equiv} \quad (\neg v_n \vee \phi'_1) \wedge (v_n \vee \phi'_2) \wedge \phi'_3 \\
&\overset{(Def.5)}{=} \quad \phi
\end{aligned}$$

En el caso b), también por la Definición 5 sabemos que

$$\phi_1 \wedge \phi_2 \;=\; ((\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi_{13}) \wedge \phi_2$$

para unas fórmulas en R-CNF tales que $(vars(\phi_2) \cup \bigcup_{i=1}^{3} vars(\phi_{1i})) = \overline{v}$. Por ende:

$$\begin{array}{rcl} \phi_1 \wedge \phi_2 & = & ((\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi_{13}) \wedge \phi_2 \\ & \overset{(10)}{\equiv} & (\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge (\phi_{13} \wedge \phi_2) \\ & \overset{(H.I.)}{\equiv} & (\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi' \\ & \overset{(Def.5)}{=} & \phi \end{array}$$

Alternativamente:

$$\begin{array}{rcl} \phi_1 \wedge \phi_2 & = & ((\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi_{13}) \wedge \phi_2 \\ & \overset{(4)}{\equiv} & ((\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi_{13}) \wedge \phi_2 \wedge \phi_2 \\ & \overset{(1)}{\equiv} & ((\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi_{13}) \wedge \phi_2 \wedge (\phi_2 \vee false) \\ & \overset{(16)}{\equiv} & ((\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi_{13}) \wedge \phi_2 \wedge (\phi_2 \vee (\neg v_n \wedge v_n)) \\ & \overset{(11)}{\equiv} & ((\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi_{13}) \wedge \phi_2 \wedge (\phi_2 \vee \neg v_n) \wedge (\phi_2 \vee v_n) \\ & \overset{(7,8,10)}{\equiv} & ((\neg v_n \vee \phi_{11}) \wedge (\neg v_n \vee \phi_2)) \wedge ((v_n \vee \phi_{12}) \wedge (v_n \vee \phi_2)) \wedge (\phi_{13} \wedge \phi_2) \\ & \overset{(11)}{\equiv} & (\neg v_n \vee (\phi_{11} \wedge \phi_2)) \wedge (v_n \vee (\phi_{12} \wedge \phi_2)) \wedge (\phi_{13} \wedge \phi_2) \\ & \overset{(H.I.)}{\equiv} & (\neg v_n \vee \phi'_1) \wedge (v_n \vee \phi'_2) \wedge \phi'_3 \\ & \overset{(Def.5)}{=} & \phi \end{array}$$

□

En la Sección 7.2 veremos qué variante es experimentalmente más eficiente. Presumimos que será la primera, ya que aplicamos una única conjunción en vez de tres.

La disyunción de fórmulas en R-DNF la encontramos en el Teorema 7 de la Sección A.1

Ahora, vamos a tratar con la disyunción de fórmulas en R-CNF.

Teorema 2. *(Disyunción de fórmulas R-CNF) La disyunción de dos fórmulas R-CNF se puede transformar a otra fórmula R-CNF equivalente.*

Demostración. Tenemos que demostrar que para cualquier par de fórmulas ϕ_1 y ϕ_2 en R-CNF, tales que $vars(\phi_1) \cup vars(\phi_2) = \{v_1, ..., v_n\} = \overline{v}$, existe una fórmula ϕ en R-CNF tal que $vars(\phi) = \overline{v}$ y $\phi \equiv \phi_1 \vee \phi_2$.

Para esto, procederemos por inducción sobre n.

Cuando $n = 0$ (ϕ_1 y ϕ_2 son T o F) los casos son triviales por identidad y dominación (propiedades 1 y 5).

Para el caso inductivo ($n > 0$), asumimos que, para cualquier par de fórmulas ψ_1 y ψ_2 en R-CNF, siempre existe una fórmula ψ en R-CNF tal que $\psi \equiv \psi_1 \vee \psi_2$ con $vars(\psi) = vars(\psi_1) \cup vars(\psi_2) = \overline{v}$. Esta es nuestra hipótesis de inducción (H.I.).

Tenemos que demostrar que para cualquier par de fórmulas ϕ_1 y ϕ_2 en R-CNF, tales que $vars(\phi_1) \cup vars(\phi_2) = \overline{v} \cup \{v_{n+1}\}$, existe una fórmula ϕ en R-CNF tal que $vars(\phi) = \overline{v} \cup \{v_{n+1}\}$ y $\phi \equiv \phi_1 \vee \phi_2$.

Consideramos que $vars(\phi_1) = \{v_{11}, ..., v_{1m_1}\}$ y $vars(\phi_2) = \{v_{21}, ..., v_{2m_2}\}$. Los casos donde $m_1 = 0$ o $m_2 = 0$ (alguna de las dos fórmulas es igual a T o F) son triviales por identidad y dominación (propiedades 1 y 5). Para el resto de casos y sin perder generalidad, por conmutatividad (propiedad 7), consideramos dos opciones: a) $v_{1m_1} = v_{2m_2} = v_n$ y b) $v_{1m_1} = v_n > v_{2m_2}$.

En el caso a), por Definición 5 tenemos que

$$\phi_1 \vee \phi_2 \;=\; ((\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi_{13}) \vee ((\neg v_n \vee \phi_{21}) \wedge (v_n \vee \phi_{22}) \wedge \phi_{23})$$

para unas fórmulas en R-CNF tales que $\bigcup_{i=1}^{2} \bigcup_{j=1}^{3} vars(\phi_{ij}) = \overline{v}$. Por tanto:

$$\begin{aligned}
\phi_1 \vee \phi_2 \quad &= \quad ((\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi_{13}) \vee ((\neg v_n \vee \phi_{21}) \wedge (v_n \vee \phi_{22}) \wedge \phi_{23}) \\
&\overset{(11)}{\equiv} \quad (\neg v_n \vee \phi_{11} \vee ((\neg v_n \vee \phi_{21}) \wedge (v_n \vee \phi_{22}) \wedge \phi_{23})) \wedge \\
&\qquad (v_n \vee \phi_{12} \vee ((\neg v_n \vee \phi_{21}) \wedge (v_n \vee \phi_{22}) \wedge \phi_{23})) \wedge \\
&\qquad (\phi_{13} \vee ((\neg v_n \vee \phi_{21}) \wedge (v_n \vee \phi_{22}) \wedge \phi_{23})) \\
&\overset{(11)}{\equiv} \quad (\neg v_n \vee \phi_{11} \vee \neg v_n \vee \phi_{21}) \wedge (\neg v_n \vee \phi_{11} \vee v_n \vee \phi_{22}) \wedge \\
&\qquad (\neg v_n \vee \phi_{11} \vee \phi_{23}) \wedge (v_n \vee \phi_{12} \vee \neg v_n \vee \phi_{21}) \wedge \\
&\qquad (v_n \vee \phi_{12} \vee v_n \vee \phi_{22}) \wedge (v_n \vee \phi_{12} \vee \phi_{23}) \wedge (\phi_{13} \vee \neg v_n \vee \phi_{21}) \wedge \\
&\qquad (\phi_{13} \vee v_n \vee \phi_{22}) \wedge (\phi_{13} \vee \phi_{23}) \\
&\overset{(7)}{\equiv} \quad (\neg v_n \vee \neg v_n \vee \phi_{11} \vee \phi_{21}) \wedge (\neg v_n \vee v_n \vee \phi_{11} \vee \phi_{22}) \wedge \\
&\qquad (\neg v_n \vee \phi_{11} \vee \phi_{23}) \wedge (v_n \vee \neg v_n \vee \phi_{12} \vee \phi_{21}) \wedge \\
&\qquad (v_n \vee v_n \vee \phi_{12} \vee \phi_{22}) \wedge (v_n \vee \phi_{12} \vee \phi_{23}) \wedge (\neg v_n \vee \phi_{13} \vee \phi_{21}) \wedge \\
&\qquad (v_n \vee \phi_{13} \vee \phi_{22}) \wedge (\phi_{13} \vee \phi_{23}) \\
&\overset{(3,18)}{\equiv} \quad (\neg v_n \vee \phi_{11} \vee \phi_{21}) \wedge (true \vee \phi_{11} \vee \phi_{22}) \wedge (\neg v_n \vee \phi_{11} \vee \phi_{23}) \wedge \\
&\qquad (true \vee \phi_{12} \vee \phi_{21}) \wedge (v_n \vee \phi_{12} \vee \phi_{22}) \wedge (v_n \vee \phi_{12} \vee \phi_{23}) \wedge \\
&\qquad (\neg v_n \vee \phi_{13} \vee \phi_{21}) \wedge (v_n \vee \phi_{13} \vee \phi_{22}) \wedge (\phi_{13} \vee \phi_{23}) \\
&\overset{(5,2)}{\equiv} \quad (\neg v_n \vee \phi_{11} \vee \phi_{21}) \wedge (\neg v_n \vee \phi_{11} \vee \phi_{23}) \wedge \\
&\qquad (v_n \vee \phi_{12} \vee \phi_{22}) \wedge (v_n \vee \phi_{12} \vee \phi_{23}) \wedge \\
&\qquad (\neg v_n \vee \phi_{13} \vee \phi_{21}) \wedge (v_n \vee \phi_{13} \vee \phi_{22}) \wedge (\phi_{13} \vee \phi_{23})
\end{aligned}$$

$$
\begin{aligned}
&\overset{(8)}{\equiv} && (\neg v_n \vee \phi_{11} \vee \phi_{21}) \wedge (\neg v_n \vee \phi_{11} \vee \phi_{23}) \wedge (\neg v_n \vee \phi_{13} \vee \phi_{21}) \wedge \\
& && (v_n \vee \phi_{12} \vee \phi_{22}) \wedge (v_n \vee \phi_{12} \vee \phi_{23}) \wedge (v_n \vee \phi_{13} \vee \phi_{22}) \wedge \\
& && (\phi_{13} \vee \phi_{23}) \\
&\overset{(11)}{\equiv} && (\neg v_n \vee ((\phi_{11} \vee \phi_{21}) \wedge (\phi_{11} \vee \phi_{23}) \wedge (\phi_{13} \vee \phi_{21}))) \wedge \\
& && (v_n \vee ((\phi_{12} \vee \phi_{22}) \wedge (\phi_{12} \vee \phi_{23}) \wedge (\phi_{13} \vee \phi_{22}))) \wedge \\
& && (\phi_{13} \vee \phi_{23}) \\
&\overset{(11)}{\equiv} && (\neg v_n \vee ((\phi_{11} \vee (\phi_{21} \wedge \phi_{23})) \wedge (\phi_{13} \vee \phi_{21}))) \wedge \\
& && (v_n \vee ((\phi_{12} \vee (\phi_{22} \wedge \phi_{23})) \wedge (\phi_{13} \vee \phi_{22}))) \wedge \\
& && (\phi_{13} \vee \phi_{23}) \\
&\overset{(Teo.1)}{\equiv} && (\neg v_n \vee ((\phi_{11} \vee \phi'_{11}) \wedge (\phi_{13} \vee \phi_{21}))) \wedge \\
& && (v_n \vee ((\phi_{12} \vee \phi'_{21}) \wedge (\phi_{13} \vee \phi_{22}))) \wedge \\
& && (\phi_{13} \vee \phi_{23}) \\
&\overset{(H.I.)}{\equiv} && (\neg v_n \vee (\phi'_{12} \wedge \phi'_{13})) \wedge (v_n \vee (\phi'_{22} \wedge \phi'_{23})) \wedge \phi'_3 \\
&\overset{(Teo.1)}{\equiv} && (\neg v_n \vee \phi'_{14}) \wedge (v_n \vee \phi'_{24}) \wedge \phi'_3 \\
&\overset{(Def.5)}{=} && \phi
\end{aligned}
$$

En el caso b), también por Definición 5 sabemos que

$$\phi_1 \vee \phi_2 \;=\; ((\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi_{13}) \vee \phi_2$$

para unas fórmulas R-CNF tales que $(vars(\phi_2) \cup \bigcup_{i=1}^{3} vars(\phi_{1i})) = \overline{v}$. Por ende:

$$
\begin{aligned}
\phi_1 \vee \phi_2 \quad &= && ((\neg v_n \vee \phi_{11}) \wedge (v_n \vee \phi_{12}) \wedge \phi_{13}) \vee \phi_2 \\
&\overset{(11)}{\equiv} && (\neg v_n \vee \phi_{11} \vee \phi_2) \wedge (v_n \vee \phi_{12} \vee \phi_2) \wedge (\phi_{13} \vee \phi_2) \\
&\overset{(H.I.)}{\equiv} && (\neg v_n \vee \phi'_1) \wedge (v_n \vee \phi'_2) \wedge \phi'_3 \\
&\overset{(Def.5)}{=} && \phi
\end{aligned}
$$

□

La conjunción de fórmulas en R-DNF está en el Teorema 8 de la Sección A.1

5.3. Eliminación de cuantificadores en fórmulas en FNI

En esta sección, vamos a describir las transformaciones que eliminan el cuantificador más interno. Al igual que en la Sección 5.2, nos vamos a centrar únicamente en fórmulas QBF en forma normal prenexa que tengan la matriz en R-CNF (tanto si están en FNI, en caso de que

el cuantificador más interno sea universal, como si no, en caso de que el cuantificador más interno sea existencial). En el Apéndice A incluimos los teoremas duales de eliminación de cuantificadores para fórmulas con matriz en R-DNF.

Antes que nada, vamos a introducir dos resultados auxiliares. En primer lugar, consideramos que el cuantificador más interno es existencial.

Teorema 3. *Dado cualquier $v \in V$, cualquier bloque de cuantificadores $\overline{Q}\overline{v}$ y cualquier fórmula proposicional ϕ tal que $\overline{v} \subset V, v \notin \overline{v}$ y $vars(\phi) = \overline{v}$, la fórmula*

$$\overline{Q}\overline{v}\ \exists v(\psi \wedge \phi)$$

donde ψ es $\neg v$ o v es lógicamente equivalente a:

$$\overline{Q}\overline{v}\ \phi$$

Demostración. Primero, consideramos que $\psi = \neg v$. Entonces:

$$\begin{array}{lcl}
\overline{Q}\overline{v}\ \exists v\ (\ \psi \wedge \phi\ ) & = & \overline{Q}\overline{v}\ \exists v\ (\ \neg v \wedge \phi\ ) \\
 & \overset{(Def.3)}{\equiv} & \overline{Q}\overline{v}\ (\ (\neg v \wedge \phi)[v \leftarrow false] \vee (\neg v \wedge \phi)[v \leftarrow true]\ ) \\
 & \equiv & \overline{Q}\overline{v}\ (\ (\neg false \wedge \phi) \vee (\neg true \wedge \phi)\ ) \\
 & \overset{(17,15)}{\equiv} & \overline{Q}\overline{v}\ (\ (true \wedge \phi) \vee (false \wedge \phi)\ ) \\
 & \overset{(2,6,1)}{\equiv} & \overline{Q}\overline{v}\ \phi
\end{array}$$

Similarmente, si $\psi = v$ tenemos:

$$\begin{array}{lcl}
\overline{Q}\overline{v}\ \exists v\ (\ \psi \wedge \phi\ ) & = & \overline{Q}\overline{v}\ \exists v\ (\ v \wedge \phi\ ) \\
 & \overset{(Def.3)}{\equiv} & \overline{Q}\overline{v}\ (\ (v \wedge \phi)[v \leftarrow false] \vee (v \wedge \phi)[v \leftarrow true]\ ) \\
 & \equiv & \overline{Q}\overline{v}\ (\ (false \wedge \phi) \vee (true \wedge \phi)\ ) \\
 & \overset{(6,1,2)}{\equiv} & \overline{Q}\overline{v}\ \phi
\end{array}$$

□

Por otro lado, consideramos el caso en el que la variable más interna es universal.

Teorema 4. *Dada cualquier $v \in V$, cualquier bloque de cuantificadores $\overline{Q}\overline{v}$ y cualquier fórmula proposicional ϕ tal que $\overline{v} \subset V, v \notin \overline{v}$ y $vars(\phi) = \overline{v}$, la fórmula*

$$\overline{Q}\overline{v}\ \forall v(\psi \vee \phi)$$

donde ψ es $\neg v$ o v es lógicamente equivalente a:

$$\overline{Q}\overline{v}\ \phi$$

Demostración. Primero, consideramos el caso en el que $\psi = \neg v$. Entonces:

$$\begin{array}{rcl}
\overline{Q}\overline{v}\,\forall v\ (\ \psi \vee \phi\ ) & = & \overline{Q}\overline{v}\,\forall v\ (\ \neg v \vee \phi\ ) \\
& \overset{(Def.3)}{\equiv} & \overline{Q}\overline{v}\ (\ (\neg v \vee \phi)[v \leftarrow false] \wedge (\neg v \vee \phi)[v \leftarrow true]\ ) \\
& \equiv & \overline{Q}\overline{v}\ (\ (\neg false \vee \phi) \wedge (\neg true \vee \phi)\ ) \\
& \overset{(17,15)}{\equiv} & \overline{Q}\overline{v}\ (\ (true \vee \phi) \wedge (false \vee \phi)\ ) \\
& \overset{(5,2,1)}{\equiv} & \overline{Q}\overline{v}\ \phi
\end{array}$$

Similarmente, si $\psi = v$ tenemos:

$$\begin{array}{rcl}
\overline{Q}\overline{v}\,\forall v\ (\ \psi \vee \phi\ ) & = & \overline{Q}\overline{v}\,\forall v\ (\ v \vee \phi\ ) \\
& \overset{(Def.3)}{\equiv} & \overline{Q}\overline{v}\ (\ (v \vee \phi)[v \leftarrow false] \wedge (v \vee \phi)[v \leftarrow true]\ ) \\
& \equiv & \overline{Q}\overline{v}\ (\ (false \vee \phi) \wedge (true \vee \phi)\ ) \\
& \overset{(1,6,2)}{\equiv} & \overline{Q}\overline{v}\ \phi
\end{array}$$

□

Usando los resultados auxiliares de los Teoremas 3 y 4 distinguimos dos casos. Primero, el caso en el que la fórmula está en FNI, donde la matriz está en R-CNF y la variable más interna es universal.

Teorema 5. *La fórmula en FNI*

$$\overline{Q}\overline{v}\ \forall v\ \phi$$

puede transformarse a una fórmula proposicional cuantificada equivalente de la forma

$$\overline{Q}\overline{v}\ \psi$$

donde ψ está en R-CNF con $vars(\psi) = \overline{v}$.

Demostración. Por la Definición 6, tenemos que

$$\phi\ =\ (\neg v \vee \phi_1) \wedge (v \vee \phi_2) \wedge \phi_3$$

para ϕ_1, ϕ_2 y ϕ_3 en R-CNF tales que $vars(\phi_1) \cup vars(\phi_2) \cup vars(\phi_3) = \overline{v}$. Por ello:

$$\begin{array}{rcl}
\overline{Q}\overline{v}\,\forall v\ \phi & = & \overline{Q}\overline{v}\,\forall v\ (\ (\neg v \vee \phi_1) \wedge (v \vee \phi_2) \wedge \phi_3\ ) \\
& \overset{(25)}{\equiv} & \overline{Q}\overline{v}\ (\ \forall v\ (\neg v \vee \phi_1) \wedge \forall v\ (v \vee \phi_2) \wedge \forall v\ \phi_3\ ) \\
& \overset{(Teo.4)}{\equiv} & \overline{Q}\overline{v}\ (\ \phi_1 \wedge \phi_2 \wedge \forall v\ \phi_3\ ) \\
& \overset{(27)}{\equiv} & \overline{Q}\overline{v}\ (\ \phi_1 \wedge \phi_2 \wedge \phi_3\ ) \\
& \overset{(Teo.1)}{\equiv} & \overline{Q}\overline{v}\ \psi
\end{array}$$

□

El Teorema 9 de la eliminación del cuantificador más interno existencial con fórmulas en FNI con matriz en R-DNF está en la Sección A.1.

Ahora, consideramos el caso donde la fórmula QBF en forma normal prenexa no está en FNI por la discordancia entre el cuantificador más interno y la forma regular de la matriz; es decir, R-CNF con cuantificador existencial.

Teorema 6. *Dada una fórmula proposicional en R-CNF ϕ tal que $vars(\phi) = \overline{v} \cup \{v\}$, la fórmula QBF en forma normal prenexa*

$$\overline{Q}\overline{v}\ \exists v\ \phi$$

puede ser transformada a una fórmula proposicional cuantificada equivalente de la forma

$$\overline{Q}\overline{v}\ \psi$$

donde ψ está en R-CNF y $vars(\psi) = \overline{v}$.

Demostración. Por la Definición 5, tenemos que

$$\phi \ = \ (\neg v \vee \phi_1) \wedge (v \vee \phi_2) \wedge \phi_3$$

para unas fórmulas ϕ_1, ϕ_2 y ϕ_3 en R-CNF tales que $vars(\phi_1) \cup vars(\phi_2) \cup vars(\phi_3) = \overline{v}$. Por tanto:

$$\begin{aligned}
\overline{Q}\overline{v}\ \exists v\ \phi \quad &= \quad \overline{Q}\overline{v}\ \exists v\ (\ (\neg v \vee \phi_1) \wedge (v \vee \phi_2) \wedge \phi_3\ ) \\
&\overset{(2)}{\equiv} \quad \overline{Q}\overline{v}\ \exists v\ (\ true \wedge (\neg v \vee \phi_1) \wedge true \wedge (v \vee \phi_2) \wedge \phi_3\ ) \\
&\overset{(18)}{\equiv} \quad \overline{Q}\overline{v}\ \exists v\ (\ (\neg v \vee v) \wedge (\neg v \vee \phi_1) \wedge (\neg v \vee v) \wedge (v \vee \phi_2) \wedge \phi_3\ ) \\
&\overset{(12)}{\equiv} \quad \overline{Q}\overline{v}\ \exists v\ (\ ((\neg v \wedge (\neg v \vee \phi_1)) \vee (v \wedge (\neg v \vee \phi_1))) \wedge \\
&\qquad\qquad ((\neg v \wedge (v \vee \phi_2)) \vee (v \wedge (v \vee \phi_2))) \wedge \phi_3\ ) \\
&\overset{(14)}{\equiv} \quad \overline{Q}\overline{v}\ \exists v\ (\ (\neg v \vee (v \wedge (\neg v \vee \phi_1))) \wedge \\
&\qquad\qquad ((\neg v \wedge (v \vee \phi_2)) \vee v)) \wedge \phi_3\ ) \\
&\overset{(12)}{\equiv} \quad \overline{Q}\overline{v}\ \exists v\ (\ (\neg v \vee (v \wedge \neg v) \vee (v \wedge \phi_1)) \wedge \\
&\qquad\qquad ((\neg v \wedge v) \vee (\neg v \wedge \phi_2) \vee v) \wedge \phi_3\ ) \\
&\overset{(16,1)}{\equiv} \quad \overline{Q}\overline{v}\ \exists v\ (\ (\neg v \vee (v \wedge \phi_1)) \wedge ((\neg v \wedge \phi_2) \vee v) \wedge \phi_3\ ) \\
&\overset{(12)}{\equiv} \quad \overline{Q}\overline{v}\ \exists v\ (\ (\neg v \wedge ((\neg v \wedge \phi_2) \vee v) \wedge \phi_3) \vee (v \wedge \phi_1 \wedge \\
&\qquad\qquad ((\neg v \wedge \phi_2) \vee v) \wedge \phi_3)\ ) \\
&\overset{(12)}{\equiv} \quad \overline{Q}\overline{v}\ \exists v\ (\ (((\neg v \wedge \neg v \wedge \phi_2) \vee (\neg v \wedge v)) \wedge \phi_3) \vee \\
&\qquad\qquad (((v \wedge \phi_1 \wedge \neg v \wedge \phi_2) \vee (v \wedge \phi_1 \wedge v)) \wedge \phi_3)\ ) \\
&\overset{(16,6,1)}{\equiv} \quad \overline{Q}\overline{v}\ \exists v\ (\ (\neg v \wedge \neg v \wedge \phi_2 \wedge \phi_3) \vee (v \wedge \phi_1 \wedge v \wedge \phi_3)\ ) \\
&\overset{(4)}{\equiv} \quad \overline{Q}\overline{v}\ \exists v\ (\ (\neg v \wedge \phi_2 \wedge \phi_3) \vee (v \wedge \phi_1 \wedge \phi_3)\ )
\end{aligned}$$

Para terminar de eliminar v tenemos dos opciones. Primera opción, aplicar primero dos conjunciones entre R-CNFs y después distribuir el cuantificador existencial eliminando la variable más interna:

$$\begin{aligned}
&\overset{(Teo.1)}{\equiv} && Q\overline{v}\,\exists v\ (\ (\neg v \wedge \psi_1) \vee (v \wedge \psi_2)\ ) \\
&\overset{(24)}{\equiv} && Q\overline{v}\ (\ \exists v\ (\neg v \wedge \psi_1) \vee \exists v\ (v \wedge \psi_2)\ ) \\
&\overset{(Teo.3)}{\equiv} && Q\overline{v}\ (\ \psi_1 \vee \psi_2\ ) \\
&\overset{(Teo.2)}{\equiv} && Q\overline{v}\ \psi
\end{aligned}$$

Y segunda opción, aplicar primero la distribución del cuantificador y la eliminación de la variable, para después ahorrarnos una conjunción con la propiedad distributiva inversa con el ϕ_3 común:

$$\begin{aligned}
&\overset{(24)}{\equiv} && Q\overline{v}\,\exists v\ (\ \exists v(\neg v \wedge \phi_2 \wedge \phi_3) \vee \exists (v \wedge \phi_1 \wedge \phi_3)\ ) \\
&\overset{(Teo.3)}{\equiv} && Q\overline{v}\ (\ (\phi_1 \wedge \phi_3) \vee (\phi_2 \wedge \phi_3)\ ) \\
&\overset{(12,8)}{\equiv} && Q\overline{v}\ (\ (\phi_1 \vee \phi_2) \wedge \phi_3\ ) \\
&\overset{(Teo.2)}{\equiv} && Q\overline{v}\ (\psi_1 \wedge \phi_3) \\
&\overset{(Teo.1)}{\equiv} && Q\overline{v}\ \psi
\end{aligned}$$

□

Veremos en la Sección 7.2 cuál de las dos variantes es más eficiente en la práctica. A *priori*, la segunda opción requiere una conjunción menos, por lo que parece más interesante.

El Teorema 9 de la eliminación del cuantificador más interno universal para el caso de fórmulas QBF en forma normal prenexa y matriz en R-DNF (por tanto no en FNI) está en la Sección A.2.

5.4. Uniendo las piezas

Para terminar, en esta sección plantearemos el algoritmo basado en la FNI en general, sin entrar en detalles de implementación, usando las operaciones definidas en las Secciones 5.1, 5.2 y 5.3.

El primer paso es identificar la lista de bloques de cuantificadores (duplas cuantificador-lista de variables) y la matriz de la fórmula (que asumimos que está en CNF).

El siguiente paso es convertir la matriz en CNF a R-CNF.

Por último, una vez la matriz está en R-CNF, empezamos a aplicar las operaciones de eliminación de cuantificadores descritas en las demostraciones de los Teoremas 5 y 6. La

matriz de la fórmula siempre se mantiene en R-CNF. Así, cuando el bloque de cuantificadores más interno es universal ($\forall$), la fórmula está en FNI y se aplica la eliminación del Teorema 5 (que a su vez utiliza la operación booleana de conjunción entre fórmulas en R-CNF del Teorema 1). En cambio, cuando el bloque más interno es existencial ($\exists$), la fórmula no está en FNI, pero tenemos definida la eliminación en el Teorema 6 para este caso (que a su vez utiliza las operaciones booleanas de conjunción y disyunción de fórmulas en R-CNF de los Teoremas 1 y 2). De esta forma, eliminando sucesivamente todas las variables cuantificadas del más interno al más externo (del n-ésimo al primero), llegará un punto (como más tarde cuando hayamos eliminado todas las variables) en el que la fórmula se habrá simplificado a T o F; es decir, habremos resuelto que la fórmula es satisfactible o insatisfactible, respectivamente.

Vamos a ejemplificar los pasos con

$$\phi_6 = \forall x \exists y \; \phi_4 = \forall x \exists y \; ((x \vee \neg y) \wedge (\neg x \vee y))$$

que tiene como matriz la fórmula ϕ_4 que ya hemos convertido a R-CNF en la Sección 5.1 (ver la Figura 5.1 para visualizar la fórmula). Es una variación de ϕ_5 cambiando los tipos de cuantificadores, haciendo que la fórmula no esté en FNI. Así, ejemplificaremos la eliminación más complicada del Teorema 6, la que requiere la operación de disyunción.

Habiendo identificado la matriz ϕ_4 y los cuantificadores, con la regularización de ϕ_4 (ver Sección 5.1, Figura 5.1) obtenemos la siguiente equivalencia:

$$\phi_4 \equiv (\neg y \vee \underbrace{((\neg x \vee T) \wedge (x \vee F) \wedge T)}_{\phi_{41}}) \wedge (y \vee \underbrace{((\neg x \vee F) \wedge (x \vee T) \wedge T)}_{\phi_{42}}) \wedge \underbrace{T}_{\phi_{43}}$$

Después, pasamos a la parte más costosa del algoritmo, a la simplificación sucesiva de ϕ_6 hasta llegar a $True$ o $False$. El bloque más interno es existencial, por lo que empezamos aplicando el Teorema 6, es decir, calculando la disyunción de las subfórmulas ϕ_{41} y ϕ_{42} (Teorema 2) seguido de la conjunción con ϕ_{43} (Teorema 1):

$$\phi_{41} = (\neg x \vee \underbrace{T}_{\phi_{41,1}}) \wedge (x \vee \underbrace{F}_{\phi_{41,2}}) \wedge \underbrace{T}_{\phi_{41,3}}$$

$$\phi_{42} = (\neg x \vee \underbrace{F}_{\phi_{42,1}}) \wedge (x \vee \underbrace{T}_{\phi_{42,2}}) \wedge \underbrace{T}_{\phi_{42,3}}$$

Para calcular la disyunción entre dos fórmulas con la misma variable en la raíz (en este caso, tienen una única variable, x) tenemos que aplicar varias operaciones de conjunción e incluso operaciones recursivas de disyunción a las subfórmulas:

$$\phi'_{11} = \phi_{42,1} \wedge \phi_{42,3} = F \wedge T = F$$

$$\phi'_{21} = \phi_{42,2} \wedge \phi_{42,3} = T \wedge T = T$$

$$\phi'_{12} = \phi_{41,1} \vee \phi'_{11} = T \vee F = T$$

$$\phi'_{13} = \phi_{41,3} \vee \phi_{42,1} = T \vee F = T$$

$$\phi_{22}^{'} = \phi_{41,2} \vee \phi_{21}^{'} = F \vee T = T$$
$$\phi_{23}^{'} = \phi_{41,3} \vee \phi_{42,2} = T \vee T = T$$
$$\phi_{3}^{'} = \phi_{41,3} \vee \phi_{42,3} = T \vee T = T$$
$$\phi_{14}^{'} = \phi_{12}^{'} \wedge \phi_{13}^{'} = T \wedge T = T$$
$$\phi_{24}^{'} = \phi_{22}^{'} \wedge \phi_{23}^{'} = T \wedge T = T$$

Para finalizar:

$$\psi_1 = \phi_{41} \vee \phi_{42} = (\neg x \vee \phi_{14}^{'}) \wedge (x \vee \phi_{24}^{'}) \wedge \phi_{3}^{'} = (\neg x \vee T) \wedge (x \vee T) \wedge T$$

$$\psi = \psi_1 \wedge \phi_{43} = \psi_1 \wedge T = \psi_1 = (\neg x \vee T) \wedge (x \vee T) \wedge T = T$$

En este caso, como tenemos que las tres subfórmulas de ψ son T, ψ_6 se simplifica directamente a T. Es decir, la fórmula es satisfactible.

Para ilustrar cómo hubiéramos aplicado la eliminación de una variable existencial supongamos que tenemos esta fórmula cambiando ligeramente a ψ:

$$\phi_7 = \forall x(\psi') = \forall x((\neg x \vee F) \wedge (x \vee T) \wedge T)$$

Así, sólo queda aplicar una iteración. Tenemos que la matriz ψ' está en R-CNF y que el cuantificador que queda es universal. Por tanto, la fórmula está en FNI, por lo que aplicamos el Teorema 5. En este caso, sencillamente aplicamos la conjunción (Teorema 1) entre las tres subfórmulas:

$$\psi_1' = F \wedge \psi_2' = \psi_3' = T \implies \bigwedge_{i=1}^{3} \psi_i = F$$

Por lo tanto, tenemos que $\phi_7 \equiv F$. Es decir, esta otra fórmula es insatisfactible.

Como hemos visto, hacer a mano incluso el ejemplo sencillo con ϕ_6 es tedioso. El ordenador también tiene que hacer todas esas operaciones para llegar al resultado. Los resolvedores basados en eliminación de variables más sencillos tienen el inconveniente de necesitar transformar entre CNF y DNF para realizar operaciones booleanas con los que eliminar ambos tipos de cuantificadores. Por eso, muchos de los resolvedores modernos tratan otras aproximaciones. En nuestro caso, seguimos empleando principalmente eliminación de variables, pero las matrices en R-CNF nos permiten evitar tener que transformarlas completamente a DNF cuando el cuantificador es existencial. Es de destacar que la transformación que realizamos es parcial, solo se le aplica al primer nivel de la fórmula (Teorema 6). Además, hay variantes y muchas simplificaciones que optimizan la implementación: evaluar de manera perezosa ($lazy$) para evitar operaciones innecesarias al conseguir T o F antes de lo esperado; simplificaciones que se pueden aplicar a las fórmulas después de cada operación booleana; simplificaciones a la matriz en CNF antes de convertirlo a R-CNF; etc. Veremos en detalle todo esto y más (por ejemplo, optimizaciones especificas del lenguaje de implementación) en el Capítulo 6. En el Capítulo 7 veremos si las optimizaciones implementadas y el algoritmo basado en la FNI es suficiente para superar las limitaciones de la eliminación de variables.

CAPÍTULO 6

Implementación

En este capítulo hablaremos sobre la implementación del *parser* de QDIMACS, los resolvedores y el preprocesador de fórmulas en CNF cuantificadas que hemos realizado.

6.1. Repositorio

En esta sección explicamos el lenguaje de programación elegido para desarrollar el proyecto. Además, describimos la estructura de la parte principal de nuestro repositorio que contiene el código de los resolvedores.

Lenguaje

El lenguaje de implementación que hemos utilizado ha sido Python [1]. Es el lenguaje de programación principal en la rama de Computación de la carrera de Ingeniería Informática al que pertenece este proyecto. Por ese mismo motivo, es el lenguaje que conocemos mejor. Por último, es un lenguaje interpretado y dinámico que permite periodos de desarrollo más breves, y por tanto es muy bueno para prototipar software.

Directorios

En la Figura 6.1 encontramos los subdirectorios en el repositorio del proyecto con el código fuente y los casos de prueba básicos planteados durante el desarrollo.

La parte central de la implementación está en la carpeta `src`. En él, encontramos estos grupos de archivos:

[1] https://www.python.org/

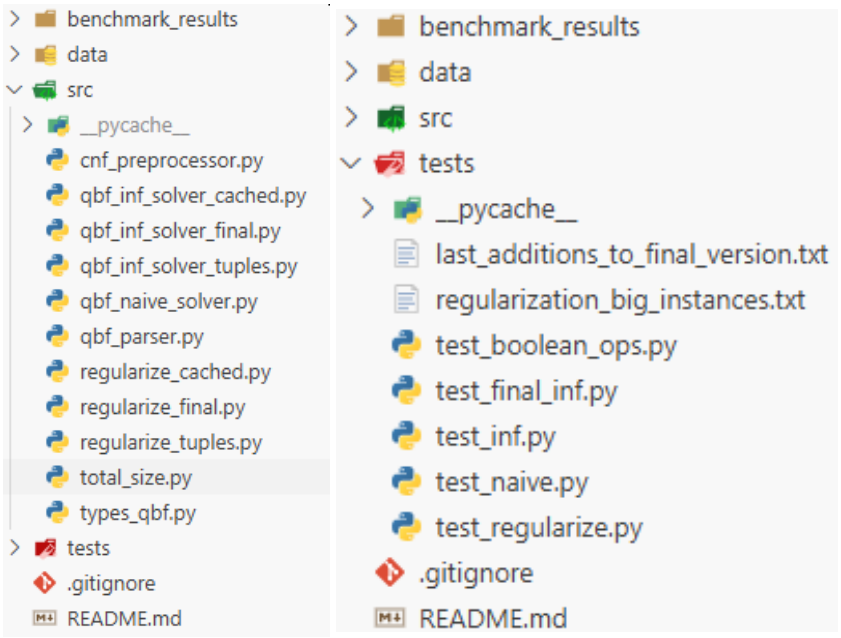

Figura 6.1: Directorios del código fuente y de las pruebas en el repositorio del proyecto.

1. `qbf_parser.py`: el *script* con las implementaciones del *parser* de QDIMACS.
2. `types_qbf.py`: un *script* simple que recoge los *type hints* para poner en las definiciones de las funciones y documentar así el propio código fuente.
3. `cnf_preprocessor.py`: el *script* con la implementación del preprocesador para fórmulas en CNF cuantificadas en forma normal prenexa, y otros preprocesamientos para la transformación de CNF a R-CNF.
4. `qbf_naive_solver.py`: el *script* con la implementación del resolvedor Naive comentado en el Capítulo 4.
5. `regularize`: estos *scripts* contienen distintas versiones de definiciones de la transformación de CNF a R-CNF y las operaciones booleanas de fórmulas en R-CNF; es decir, las operaciones auxiliares para nuestro resolvedor de QBF.
6. `qbf_inf_solver`: estos *scripts* contienen distintas versiones (cada una con su correspondiente `regularize`) del resolvedor QBF basado en la FNI presentado en el Capítulo 5, con la función principal que hace uso de la función de transformación a R-CNF y la definición de la función de eliminación de cuantificadores que emplea las operaciones booleanas.
7. `total_size.py`: un *script* auxiliar que contiene una función que calcula el espacio en memoria que ocupa cualquier objeto de Python que puede tener anidados otros objetos, sumando el espacio de un mismo objeto una sola vez.

Por otro lado, en la carpeta `tests` hemos dejado las pruebas más relevantes y visuales hechas durante el desarrollo, que sirven para comprobar rápidamente si las implementaciones son válidas.

El repositorio del proyecto está disponible en la página de GitHub [40].

En las siguientes secciones comentaremos más en profundidad el contenido de los conjuntos de ficheros enumerados previamente.

6.2. Parser

En esta sección detallamos la interfaz que cumplen los *parsers* del formato QDIMACS y las implementaciones que hemos realizado.

Interfaz general de parsers del formato QDIMACS

En general, un *parser* de archivos QDIMACS, que codifican fórmulas en forma normal prenexa con matriz en CNF, tiene que tomar el contenido del archivo y devolver como resultado dos elementos:

1. La matriz en CNF como una lista de listas de números enteros distintos de cero.
2. Los bloques de cuantificadores de la fórmula como una lista de duplas de la forma $(Q, [v_1, \ldots, v_n])$ donde $Q \in \{\exists, \forall\}$ y $v_i \in \mathbb{Z}^+$, correspondiendo a los literales indicados en la matriz. Cabe mencionar que otra posibilidad para los bloques de cuantificadores sería devolver una sencilla lista de duplas donde cada variable tiene emparejada a su cuantificador. Nosotros hemos usado la primera versión porque es más compacta.

He aquí un ejemplo con el archivo ya mostrado en la Sección 3.2:

$$\phi_2 = \forall x \exists y((x \vee y) \wedge (\neg x \vee \neg y))$$

```
c Podemos tener comentarios, normalmente solo al principio
c x1 = x ; x2 = y
p cnf 2 2
a 1 0
e 2 0
1 2 0
-1 -2 0
```

Fichero 6.1: Ejemplo del fichero QDIMACS que codifica la fórmula ϕ_2 (el mismo Fichero 3.1).

El *parser* sobre ese archivo QDIMACS da como resultado:

$$\text{Matriz} : [[1, 2], [-1, -2]]$$

$$\text{Bloques de cuantificadores} : [(\forall, [1]), (\exists, [2])]$$

Parsers implementados

En nuestro caso, en el fichero `qbf_parser.py` hemos definido distintos tipos de *parsers* para el formato QDIMACS como funciones. Todas ellas empiezan por `read_qdimacs`, y terminan con un sufijo que especifica su tipo. Se pueden clasificar según dos características:

1. Tipo de lectura:
   a) String (`str`). Leen todo el contenido del fichero a un string en memoria y operan sobre el string.
   b) Fichero (`file`). Van construyendo el resultado mientras van leyendo del fichero. Más eficientes en cuanto a memoria.
2. Comprobaciones:
   a) Comprobadores (`checked`). Validan que se respete el formato QDIMACS a rajatabla, como por ejemplo que empiece por la línea de cabecera, que los números de variables y cláusulas indicadas en la misma correspondan con la fórmula codificada, que las variables usadas estén en el intervalo $[1, num\_variables]$, etc. En la descripción de la función `read_qdimacs_from_file` está la lista exhaustiva de las comprobaciones que realizan.
   b) No comprobadores (`unchecked`). Asumen que el archivo de entrada es correcto, y por tanto son mucho más simples y eficientes.

En resumen, hemos usado `read_qdimacs_from_file_unchecked` en la experimentación, por ser más el más simple y eficiente. Eso sí, hemos comprobado y corregido las instancias de prueba empleadas.

Otros formatos

Además de dichos *parsers* de QDIMACS, en `qbf_parser.py` tenemos otras funciones para *parsear* ficheros QAIGER [27] y QCIR [26]. No son implementaciones desde cero, sino que presuponen que en el sistema están instaladas herramientas para transformar archivos de estos formatos a QDIMACS y después emplean el *parser* de QDIMACS. En la experimentación hemos trabajado directamente con instancias QDIMACS, así que no hemos utilizado ni testeado estas funciones.

6.3. Resolvedor Naive

En esta sección describimos la interfaz general de los resolvedores de QBF y, en particular, explicamos el resolvedor Naive realizado.

Interfaz de resolvedores de QBF

En general, un resolvedor de QBF recibe la codificación de la fórmula cuantificada como entrada y devuelve un booleano como salida: $True$ si es satisfactible y $False$ si es insatisfactible. En nuestro caso, la codificación de las fórmulas que reciben los resolvedores que hemos implementado es el par de la matriz en CNF y la lista de los bloques de cuantificadores explicados en la Sección 6.2. Por tanto, en particular, la interfaz del resolvedor Naive es ese mismo.

Particularidades de nuestro resolvedor Naive

La idea fundamental del algoritmo de este resolvedor está en el Capítulo 4. Tal y como mencionamos en la Sección 4.3 de ese mismo capítulo, éstas son las particularidades de la implementación final de nuestro resolvedor Naive:

- Se aprovechan las propiedades de dominación 5 y 6 para evitar llamadas recursivas innecesarias. Si conseguimos $True$ en un registro de llamada que corresponde a un bloque existencial, devolvemos $True$ directamente. Lo mismo con $False$ y bloques universales.

- La recursión se hace por bloques de cuantificadores, no variable a variable. En cada llamada recursiva se toma una asignación para el conjunto de las variables cuantificadas en el primer bloque de cuantificadores (es decir, en el más externo) de ese registro de llamada. Con esa asignación se simplifica la fórmula y se realiza la llamada recursiva. Los casos base se dan cuando solo queda un bloque de cuantificadores, ya sea existencial o universal. En caso de que no se apliquen las propiedades de dominación, se seguiría probando con la siguiente asignación para el mismo conjunto de variables en cada registro de llamadas.

- Cuando solo queda un bloque de cuantificadores existencial, como este tipo de QBFs son equivalentes a instancias SAT, llamamos a un resolvedor de SAT eficiente implementado en PySAT (el resolvedor por defecto). Cuando el último bloque de cuantificadores es universal, no queda otra que probar con todas las asignaciones posibles de las variables en dicho bloque y ver si todas satisfacen a la matriz.

Otras versiones

Por último, cabe mencionar que en `qbf_naive_solver.py` están implementadas otras funciones con distintas estrategias naive. La variante principal que hemos descrito antes sería la versión `v1`. Tenemos otras: la versión `v1.2` es igual a la versión `v1` pero en cada llamada recursiva lo primero que se hace es resolver la matriz de la fórmula con PySAT, y en caso de que sea insatisfactible, se devuelve $False$ directamente. Esto es posible porque la cuantificación universal es más estricta que la existencial. Por tanto, si cuantificando todas las variables

existencialmente no hay ni una sola asignación que haga cierta a la fórmula, entonces si alguna de las variables estuviera cuantificada universalmente dicha fórmula también sería insatisfactible con más razón. Por otro lado, la versión `v3` hace recursión variable a variable. Y la versión `v3.2` es para `v3` lo que `v1.2` es para `v1`. Ninguna de estas versiones adicionales se ha usado en la experimentación ni se ha testeado de manera exhaustiva. En unas primeras pruebas iniciales vimos que `v1` era más eficiente que `v3`.

Como curiosidad, después de `v1` viene la versión `v3` porque descartamos otra versión (`v2`), dado que requería programar operaciones como conjunción y disyunción de fórmulas en CNF. Como ya teníamos la versión `v1` para comparar con el resolvedor basado en la FNI, que incluso es más sencilla que `v2`, decidimos no invertir tiempo en este trabajo que no aportaría nada al siguiente paso de comparar resolvedores.

6.4. Resolvedor basado en la Forma Normal Inductiva

En esta sección veremos detenidamente la implementación central del proyecto: el resolvedor de QBF basado en la FNI, cuyo algoritmo fue explicado en el Capítulo 5.

6.4.1. Representación de fórmulas en R-CNF

La primera tarea ha sido decidir cómo representar en Python fórmulas en R-CNF. Hemos utilizado dos representaciones, además de estrategias adicionales para tratar de reducir el gasto de memoria.

Tuplas como nodos

La representación más natural es como un árbol ternario. Empezamos por emplear las *clases* de Python, pero vimos que el gasto de memoria era demasiado elevado, ya que Python implementa las clases definidas por usuarios como diccionarios (`dicts` en Python, es decir, tablas *hash*), no como `structs` compactos de C. Por ello, probamos después con otras herramientas de Python como los `dataclasses`, pero vimos que incluso estos tipos de clases están representados mediante diccionarios. Por ello, pasamos de las clases definidas por usuario y empezamos a emplear tipos de datos nativos en Python: listas o tuplas. Ambos eran factibles, ya que son mucho más eficientes en memoria que las aproximaciones iniciales. Nos decantamos por las tuplas por dos motivos: primero, ocupan ligeramente menos memoria que las listas; y segundo, son inmutables, una propiedad que los nodos del árbol ternario en nuestro algoritmo cumplían de manera natural y que, al emplear tuplas, nos aseguramos de respetar gracias al propio lenguaje Python.

Es decir, la primera representación posible son 4-tuplas anidadas: el primer elemento es la variable correspondiente a ese nodo, y los tres últimos son referencias a otras tuplas que representan las subfórmulas que contienen a la variable negada, sin negar o no la contienen. Las hojas de esta estructura de árbol ternario son los booleanos (`bool` en Python)

`True` y `False`. Esta representación se emplea en los ficheros `qbf_inf_solver_tuples.py` y `regularize_tuples.py` (y también en la versión final).

Variables guardadas explícitamente en los nodos

Tener la variable como primer elemento de los nodos parecería innecesario si consideramos que la idea de R-CNF es tener un nivel por cada variable. No obstante, permite evitar la necesidad de tener un árbol completo, que requeriría una cantidad de memoria exponencial con respecto al número de variables, por lo que es un detalle clave.

Evitar repetición de nodos

Otra idea importante es que podemos evitar la repetición de los nodos iguales que representan las mismas subfórmulas. Para ello hemos usado el decorador `@lru_cache` de Python sobre la función de creación de nodos (`create_node`, que simplemente hace un *return* del constructor de 4-tuplas). Este decorador emplea un diccionario por debajo, por lo que tenemos un mapeo que va desde los cuatro componentes de cada nodo a la propia 4-tupla que lo representa. Así, cuando llamamos a `create_node` con los mismos argumentos por segundo vez, devolvemos exactamente el mismo nodo creado en la primera llamada. Gracias a esto, reducimos bastante el espacio en memoria empleado. Cabe mencionar que esta idea fue inspirada por los objetos `bool` de Python. `True` y `False` son objetos *singleton*, por lo que esta no repetición de nodos ya estaba ahí para los nodos hoja desde un principio. Por todo esto, con la reutilización de los nodos tenemos que un mismo nodo puede tener varios padres, por lo que ya no sería un árbol ternario estrictamente hablando, sino más bien un Grafo Acíclico Dirigido (DAG, [41]).

Ejemplo

Por ejemplo, la fórmula ϕ_4 de la Sección 5.1, que la visualizamos mejor en la figura 5.1, se vería así en está representación:

$$\phi_4 = (\neg y \vee ((\neg x \vee T) \wedge (x \vee F) \wedge (T))) \wedge (y \vee ((\neg x \vee F) \wedge (x \vee T) \wedge T)) \wedge T$$

$$\phi_4 = (2, (1, T, F, T), (1, F, T, T), T)$$

En este ejemplo no hay repetición de ninguna 4-tupla, así que tampoco hay reutilización.

Representación alternativa: nodos como identificadores

Otra representación posible que ideamos más adelante fue no utilizar tuplas anidadas, sino identificar cada nodo con un ID y tener 4-tuplas de la forma $(v_n, \mathrm{ID}_{neg}, \mathrm{ID}_{pos}, \mathrm{ID}_{abs})$ donde v_n es la variable en el propio nodo como en la primera representación, y los siguientes tres IDs identifican a los nodos que corresponden a las tres subfórmulas. Los IDs son enteros positivos

mayores que 1. Obviamos a 0 y 1 como posibles IDs ya que `bool` es un tipo especializado de `int` en Python, por lo que al hacer comparaciones estaríamos confundiendo `True` y `False` con los dos primeros nodos.

Por otro lado, es importante ver que tenemos dos mapeos o `dicts`. El primero mapea las 4-tuplas que contienen los IDs de las subfórmulas de los nodos con los IDs de los nodos, y le sirve a la versión de `create_node` de esta representación para comprobar si una 4-tupla ya tiene asignado un identificador (por lo que no tenemos repetición de nodos). El segundo asocia los IDs con las 4-tuplas. Este sirve para poder recorrer la estructura en forma de DAG.

La idea inicial de esta representación era hacer más eficientes las búsquedas en la tabla *hash* que *cachea* los nodos, ya que la función *hash* de las tuplas es recursiva, y por tanto potencialmente costosa con muchas tuplas anidadas. Mientras tanto, con esta representación manejamos simples `ints` o tuplas con cuatro `ints`. Por el mismo motivo, en esta versión hemos implementado una operación opcional para ir eliminando los nodos cacheados periódicamente, ya que con un recorrido del árbol es suficiente para calcular el conjunto de los IDs de los nodos alcanzables, que se usa después para filtrar los dos mapeos. Con la primera representación la operación de filtrado podría ser muy costosa ya que tendríamos que calcular los valores *hash* de todos los nodos *cacheados* (aunque habría estrategias para remediarlo, como usar las direcciones de memoria aprovechando que los nodos no se repiten). Los ficheros que usan esta representación son `qbf_inf_solver_cached.py` y `regularize_cached.py`. El ejemplo anterior se vería así:

$$\phi_4 = (\neg y \vee \underbrace{((\neg x \vee T) \wedge (x \vee F) \wedge (T))}_{\phi_{41}}) \wedge (y \vee \underbrace{((\neg x \vee F) \wedge (x \vee T) \wedge T)}_{\phi_{42}}) \wedge T$$

$$\phi_{41} = (1, T, F, T) \wedge \mathrm{ID}_{41} = 2$$

$$\phi_{42} = (1, T, F, T) \wedge \mathrm{ID}_{42} = 3$$

$$\phi_4 = (2, 2_{\mathrm{ID}}, 3_{\mathrm{ID}}, T) \wedge \mathrm{ID}_4 = 4$$

6.4.2. Transformación a R-CNF

El segundo paso ha sido implementar la transformación de CNF a R-CNF: la operación de *regularización*. El algoritmo se ha explicado en la Sección 5.1.

Los primeros dos pasos se hacen de manera fácil. El primero, en el que ordenamos las cláusulas teniendo en cuenta el valor absoluto de los literales, se limita a usar el método `sort` de cada cláusula teniendo como función *key* a `abs`. De esa forma, antes de la ordenación se calculan los valores absolutos de los literales en la cláusula y los elementos se ordenan en base a dichos valores. El segundo, en el que tenemos que ordenar las cláusulas entre sí siguiendo el orden lexicográfico (dando mayor prioridad a los literales negativos), no es tan fácil de implementar usando una función *key* que ya nos da Python, aparte de que es más fácil y natural implementar la función que compara dos cláusulas (con los literales ya ordenados por valor absoluto). Por ello, hemos implementado dicha función de comparación (`cmp_clauses` en `cnf_preprocessor.py`) y usado la utilidad `cmp_to_key` del módulo `functools`.

El paso tres es recursivo. La idea principal es clasificar las cláusulas que tenemos en tres grupos. Nos fijamos en la variable del primer literal de la primera cláusula (que será la variable más grande gracias a las ordenaciones de los primeros pasos). El primer conjunto de cláusulas serán aquellas que comienzan por el literal negado de dicha variable; el segundo conjunto será el de las clausulas que empiezan por el literal positivo; y el tercer y último conjunto será el de las cláusulas que no contienen a la variable. Gracias a la ordenación, los tres conjuntos están en ese orden, no están entremezclados. Una vez clasificados dichos conjuntos, se modifican eliminando la variable que se ha tenido en cuenta. Después, se hacen las llamadas recursivas. Puede ser que alguno o varios de estos conjuntos sea vacío. En ese caso, el caso base de fórmula vacía devuelve `True` (el elemento neutro de la conjunción, propiedad 2). El otro caso base es que entre las cláusulas tengamos una vacía, por lo que se devuelve `False` (elemento neutro de la disyunción, propiedad 1, y dominante de la conjunción, propiedad 6). Gracias a que las cláusulas están ordenadas según el orden lexicográfico, es suficiente con mirar la primera cláusula para detectar si hay alguna vacía, ya que éstas tienen longitud 0. Cuando conseguimos los tres resultados de las llamadas recursivas, llamamos a la función `create_node` con la variable que se ha usado en ese registro de llamadas para clasificar las cláusulas y los tres resultados. Opcionalmente, aplicamos simplificaciones a la fórmula, que se explicarán en la Sección 6.4.5.

6.4.3. Operaciones booleanas

Una vez hemos implementado la forma de tener la matriz de las fórmulas en R-CNF, el siguiente paso ha sido desarrollar sus operaciones de disyunción y conjunción. Estas operaciones se han explicado en la Sección 5.2. Las demostraciones de dicha sección nos muestran las formas de implementarlas, viendo cómo hay que operar recursivamente con las tres subfórmulas de cada una de las dos fórmulas iniciales para llegar a obtener las tres subfórmulas del resultado. Éste es el apartado que tiene el mayor número de variantes, ya que, aparte de las versiones en serie básicas, hemos intentado optimizar las operaciones de distintas maneras: con paralelización, memoizando resultados y planteando versiones acorde a las dos representaciones de matrices en R-CNF descritas en la Sección 6.4.1.

Versiones básicas en serie

Primero, hemos hecho versiones en serie básicas, que corresponden a las funciones `serial_basic` en los archivos `regularize_tuples.py`. Los detalles a considerar son los siguientes:

- Tratamos de ser perezosos lo máximo posible con *return* tempranos, en base a las restricciones impuestas por la Definición 5 de fórmulas en R-CNF. Por ello, primero operamos con las subfórmulas correspondientes a la tercera rama, la correspondiente a las subfórmulas que no cuentan con la variable mayor, para comprobar si el resultado que obtenemos es directamente `False`. Después operamos con las otras dos ramas, y

hacemos dos comprobaciones: si los dos primeros componentes del resultado son `False` o `True` a la vez, en cuyo caso devolvemos `False` o el tercer componente directamente.

- En los casos base consideramos los casos donde al menos uno de los dos operandos es booleano. En ese caso, aprovechamos las propiedades 5 y 6 de dominación o las 1 y 2 de identidad.
- Aprovechamos la propiedad conmutativa comprobando las variables en las raíces de los dos operandos, que acotan las máximas profundidades de las fórmulas. Así, guardamos en la primera variable la fórmula mayor y en una segunda variable la menor, si inicialmente la llamada se hubiera hecho con los operandos al revés.
- Al igual que vimos en la parte teórica en la demostración de conjunciones de fórmulas en R-CNF (Teorema 1), la conjunción tiene dos variantes en caso de tener fórmulas con distintas variables máximas.

Versiones paralelas

Ahora vamos a explicar cómo hemos tratado de optimizar las operaciones booleanas paralelizándolas. Empezamos comentando las limitaciones de Python para paralelizar programas, y después pasamos a las distintas formas en las que hemos paralelizado cada operación.

Limitaciones de Python

Después, hemos tratado de paralelizar las operaciones booleanas. Estas versiones corresponden a las funciones `parallel` en el mismo *script*. No obstante, en Python tenemos grandes limitaciones para paralelizar funciones (al menos en la implementación oficial CPython). La primera de ellas es el GIL (Global Interpreter Lock, [42]). Un proceso del intérprete de Python no permite que dos hilos [43] ejecuten a la vez código de Python, incluso en procesadores con múltiples *cores* [44]. Sirve para facilitar el manejo de memoria y la llamada a librerías implementadas en C. Para superar esta limitación, se suele hacer uso de múltiples procesos [45] mediante el módulo `subprocess` [2]. Cabe destacar que nuestra aplicación es intensiva en el uso de la CPU. La mayoría del tiempo la CPU está ocupada computando las operaciones booleanas de fórmulas en R-CNF. Los hilos son una buena elección cuando el cuello de botella de la aplicación es la lectura de disco, o la espera para la comunicación con otro ordenador (como un servidor web). En nuestro caso, crear subprocesos es la respuesta. No obstante, incluso con los subprocesos tenemos otra limitación: al crear un subproceso que ejecuta una función de nuestro código Python, dicha función no puede generar a su vez más subprocesos. Por ello, la paralelización que podemos tener está sólo en el primer nivel de llamadas. Esto facilita la implementación porque no tenemos que gestionar una pila de llamadas que potencialmente genera un número cada vez mayor de procesos, pero limita la paralelización. Es decir, al usar el módulo `subprocess`, los subprocesos que generamos resuelven versiones en serie de las operaciones.

[2] https://docs.python.org/3/library/subprocess.html

Formas de paralelizar

Dicho todo lo anterior, conviene comentar también que tenemos distintas formas de paralelizar cada operación:

- Conjunción. Podemos obviar los *return* tempranos y paralelizar las operaciones entre las tres subfórmulas. Por otro lado, podemos tratar de mantener el estilo perezoso de la versión en serie a costa de mayor comunicación entre los procesos, lo cual gestiona automáticamente el módulo `concurrent.futures` [3] (que por debajo usa `subprocess`).
- Disyunción. Podemos paralelizar las llamadas a las operaciones de conjunción que se realizan. Podemos también paralelizar las llamadas recursivas a disyunción. Estos dos planteamientos se hacen respetando las comprobaciones perezosas de la versión en serie. Por último, podemos plantear paralelizar todas las operaciones posibles, perdiendo *return* tempranos.

Memoización de resultados

Una vez implementadas las versiones en paralelo, hemos vuelto a versiones en serie con otra idea de optimización: memoizar [46] los resultados de las operaciones booleanas. Estas versiones están implementadas en las funciones `serial`, `serial_manual` y `serial_manual_ids`. Hay tres tipos de funciones por las distintas pruebas que hemos hecho para memoizar los resultados:

- Cache LRU (Last Recently Used [4]). Simplemente usando la etiqueta `@lru_cache` con el parámetro `max_size` a `None` sobre la versión en serie básica, para no limitar el número de resultados guardados.
- Diccionarios Python: gestión más manual en nuestro propio código. Permite aprovechar mejor la conmutatividad para no guardar dos entradas para la misma operación. También nos da la capacidad de ahorrarnos los casos básicos donde alguno de los operandos es directamente `True` o `False`.
- Diccionarios Python usando las direcciones de memoria de los operandos: esta versión solo es posible si la no repetición de los nodos está activada, gracias a lo cual cada fórmula se puede mapear biyectivamente con su dirección en memoria. La idea de esta versión es parecida a la segunda representación de fórmulas en R-CNF, donde en principio la operación *hash* sobre un `int` es más fácil que sobre unas tuplas anidadas.

Estas versiones con memoización son un tanto incompatibles con la paralelización debido al uso de subprocesos. Cada subproceso tiene su propio espacio en memoria aislado del de su padre. Por tanto, haría falta gestionar más comunicaciones entre procesos para mantener un estado coherente y completo de las *caches* de memoización.

[3] https://docs.python.org/3/library/concurrent.futures.html

[4] `https://docs.python.org/3/library/functools.html#functools.lru_cache`

Versiones con la representación mediante identificadores

Por último, están las versiones con la segunda representación de fórmulas en R-CNF. Estas funciones están en el *script* `regularize_cached.py`. No tienen tantas variantes ni tantas complicaciones. Simplemente hacemos memoización de los resultado, también usando el decorador `@lru_cache` o manualmente con diccionarios Python. La memoización de los resultados es una mejora muy natural en esta representación, en la que *cacheamos* simples `ints`. Esta representación también es bastante incompatible con la paralelización por subprocesos, por el mismo motivo que la mejora de memoización, ya que tenemos el par de mapeos entre tuplas e identificadores que habría que mantener coherentemente entre distintos procesos.

6.4.4. Eliminación de cuantificadores

Para terminar el resolvedor, el último paso ha sido implementar la eliminación de cuantificadores. Su base teórica ya se ha explicado en la Sección 5.3. Lo más importante es fijarse en las operaciones booleanas a emplear con las subfórmulas de la fórmula a resolver dependiendo del cuantificador de la variable más interna. Los *script* que implementan esta parte del algoritmo son los `qbf_inf_solver`, en las funciones `_eliminate_variables`.

Empezamos explicando las pocas variantes de eliminación de variables que hemos implementado (especialmente con variables cuantificadas existencialmente). Luego, mencionamos una estrategia que hemos empleado para tratar de usar menos memoria: liberar la memoria de los nodos que ya no son alcanzables. Al final, terminamos aclarando un par de detalles que hemos tenido que considerar para la corrección de la implementación.

Variantes

Comparado con las operaciones booleanas, no tenemos muchas variantes, ni mucho menos. Como vimos en la demostración del Teorema 6, tenemos dos variantes para eliminar variables existencialmente cuantificadas. Una de ellas hace menos operaciones: una única disyunción seguida de una conjunción. La otra hace tres operaciones: dos conjunciones y una disyunción. En esta segunda opción, las conjunciones se pueden paralelizar, pero como hemos explicado en la sección anterior, como solo se puede paralelizar un nivel, no hemos implementado esta paralelización, ya que es más interesante paralelizar cada una de las conjunciones (ya que internamente tienen más de dos llamadas recursivas paralelizables). Por otro lado, tenemos la versión recursiva y la versión iterativa de la eliminación de cuantificadores. Son prácticamente iguales salvo por un detalle que hace que a priori la versión iterativa sea claramente superior: la versión recursiva añade un registro de llamada extra por cada variable, lo cual resulta problemático debido a las limitaciones de la recursión en Python y que nuestro algoritmo ya es altamente recursivo.

Limpieza de nodos

Por otra parte, en la versión que representa las R-CNFs con IDs, hacemos periódicamente limpieza de nodos no alcanzables desde la raíz de la fórmula simplificada después de cada eliminación de variable, siempre y cuando se hayan creado más de 1000 nodos nuevos desde la limpieza anterior. Con la representación de tuplas anidadas no hacemos esta limpieza por el mayor coste que supone la función *hash* de éstas, como se ha explicado en la Sección 6.4.1. No obstante, aparte de las mediciones principales, hemos probado a implementar la limpieza con tuplas anidadas en una experimentación final pequeña, como veremos en la Sección 7.6.

Eliminación de variables adicionales

Un detalle de implementación curioso es que en un paso de eliminación de variables puede darse el caso de que se simplifique adicionalmente una o varias de las siguientes variables. Por ello, es necesario seguir iterando sobre los bloques de cuantificadores hasta dar con la variable que se encuentra en la raíz de la fórmula simplificada de la iteración actual.

Renombramiento inicial de las variables

Un último detalle interesante es que renombramos las variables para que en los bloques de cuantificadores estén ordenadas empezando desde 1 (la cuantificación más a la izquierda o externa) y terminando con n (la más a la derecha o interna). Así, respetamos la convención que hemos usado en las explicaciones teóricas del Capítulo 5 y nos aseguramos de cumplir las presuposiciones correspondientes al nombramiento de variables. Los literales de las cláusulas se modifican para que se respete la cuantificación de las variables y se mantenga la equivalencia con la fórmula inicial (que sigue siendo la misma, solamente con identificadores ajustados). Este renombramiento se hace antes incluso de llamar a la transformación de CNF a R-CNF, para hacerlo de manera más sencilla iterando sobre listas de listas.

6.4.5. Optimizando la última versión

El último paso para terminar la parte de implementación del proyecto ha sido tratar de optimizar todo lo posible el resolvedor basado en FNI. En las secciones anteriores ya hemos comentado las variantes principales de cada módulo del resolvedor, que se pueden considerar optimizaciones (que luego han sido confirmadas o refutadas en la experimentación): la segunda representación basado en IDs, no repetición de nodos pasando de árbol a DAG, paralelización o memoización de las operaciones booleanas, etc. A parte de las anteriores, hemos implementado unas cuantas más, con menor o mayor importancia, que se describen a continuación.

Comparación de booleanos por identidad

Primero, la comparación con booleanos con el operador `is` del lenguaje Python. Como ya hemos mencionado, los objetos `bool` `True` y `False` son *singleton* en Python, por lo que se puede usar su dirección de memoria directamente en vez de llamar a su operador de comparación (que es una función). No es una gran mejora, como mucho nos ahorramos algunas llamadas a función.

Comparación de nodos cacheados por identidad

Gracias a la no repetición de nodos, ya sea con una *cache* en el caso de tuplas o la propia representación de los IDs, nos ahorramos memoria. También permite realizar la comparación entre nodos con el operador `is` en el caso de tuplas anidadas. En el caso de los IDs, que son `ints`, hay que seguir con el operador de comparación (pero que es más sencillo que el de tuplas anidadas). Tampoco es un gran cambio, porque la comparación entre tuplas está muy optimizada. Además, la gran mayoría de las subfórmulas tienen estructuras distintas, por lo que no se tendrá que hacer la recursión completa. No obstante, podría llegar a ser interesante para muchas tuplas anidadas con estructuras similares. Lo verdaderamente importante de la no repetición de nodos es el ahorro de memoria en sí.

Desactivación de la recolección de basura generacional

Python es un lenguaje de alto nivel con manejo de memoria automático. Tiene dos estrategias de recolección de basura. La principal es el conteo de referencias. Cuando un objeto es referenciado por una variable, un parámetro de función, o el campo de otro objeto, su contador de referencias se incrementa. Cuando esas referencias dejan de apuntar al objeto (por ser variables locales y por terminar la llamada a la función, o porque pasa a referenciar a otro objeto), el contador de referencias se disminuye. Cuando el contador llega a cero, se libera el espacio de memoria en el *heap* [47] que ocupaba. Esta estrategia es fundamental y no se puede desactivar. Además, es eficiente con los nodos de una estructura tipo árbol o DAG. Su gran limitación es que no es capaz de detectar referencias cíclicas. Por ello, Python implementa también un recolector de basura generacional [48]. Este mecanismo añade una carga de computación innecesaria si sabemos que no tenemos referencias cíclicas en nuestro programa, que es nuestro caso. Además, el *overhead* en nuestro QBF solver es grande, ya que generamos muchos nodos (bastantes miles en función de la instancia), superando con facilidad los umbrales del número de instanciaciones generacionales, encareciendo el manejo de este mecanismo. Por eso, usamos el modulo `gc` [5] para desactivar este mecanismo. Esta mejora tampoco es fundamental.

[5] https://docs.python.org/3/library/gc.html

Delegación en un resolvedor de SAT

Si detectamos que tenemos un único bloque de cuantificador existencial, tenemos que nuestra instancia de QBF es en realidad de SAT, centrándonos exclusivamente en la matriz en CNF. En ese caso, lo más eficiente es llamar a un resolvedor de SAT optimizado, como el por defecto de PySAT. Esta mejora no es fundamental, pero las instancias de SAT dejan de ser una preocupación para nuestro algoritmo, que, al estar todas las variables cuantificadas existencialmente, requerirían una gran cantidad de disyunciones, lo cual es costoso y puede hacer explotar el tamaño de las fórmulas.

Simplificaciones a fórmulas en R-CNF

Esta siguiente optimización está más relacionada con nuestro algoritmo y es utilizada con mayor frecuencia. Se trata de simplificaciones a fórmulas en R-CNF, que aplicamos después de cada operación booleana, y también en la transformación de regularización. La idea es comparar las subfórmulas en R-CNF para comprobar si son iguales. Si algunas lo son, podemos ver que la fórmula completa es equivalente a una más simple. Supongamos que tenemos la siguiente fórmula en R-CNF:

$$\phi = (\neg v \vee \phi_1) \wedge (v \vee \phi_2) \wedge \phi_3$$

Entonces:

- Si $\phi_1 = \phi_2$, podemos aplicar la propiedad distributiva 11 obteniendo la expresión $(\neg v \vee v) \wedge \phi_1$, que por tautología (propiedad 18) y porque $True$ es elemento neutro de la conjunción (propiedad 2), tenemos como resultado que $\phi = \phi_1 \wedge \phi_3$, es decir, aplicamos otra conjunción.
- Si $\phi_1 = \phi_3$, simplemente sustituimos ϕ_1 por T, dando como resultado que $\phi = (\neg v \vee T) \wedge (v \vee \phi_2) \wedge \phi_3$. Es válido, primero porque seguimos manteniendo la forma R-CNF, y más fundamentalmente, porque la equivalencia es correcta por absorción (propiedad 14) y porque T es el elemento dominante de la disyunción y neutro de la conjunción (propiedades 2 y 5). Así, las dos expresiones de ϕ se simplifican al mismo $(v \vee \phi_2) \wedge \phi_3$, dejando clara la equivalencia. Por otro lado, es interesante porque obtenemos un T en vez de una subfórmula más complicada, haciendo más fácil que futuras llamadas a operaciones booleanas sean solucionadas por casos básicos.
- Si $\phi_2 = \phi_3$, es el mismo caso que el anterior, pero el papel de ϕ_1 pasa a cumplirlo ϕ_2.

Preprocesamiento de la matriz en CNF

Otra optimización que trabaja con conceptos de la lógica proposicional es que antes de transformar la CNF a R-CNF podemos aplicar preprocesamiento a la matriz original, teniendo en cuenta las cuantificaciones. Las tres estrategias que hemos empleado son las siguientes:

1. Polaridad única (*pure literal elimination* [49]). Si una variable siempre aparece con la misma polaridad (es decir, siempre como literal positivo o negativo), podemos: si es existencial, eliminar las cláusulas donde aparece la variable (debido a que habríamos asignado el valor que hace cierto a los literales); y si es universal, eliminamos los literales siguiendo la idea de la semántica de fórmulas QBF de la Definición 3: es decir, planteando la conjunción de la misma fórmula habiendo sustituido la variable por T y por F. Además, comprobamos si queda alguna cláusula vacía, en cuyo caso la fórmula es equivalente a $False$.

2. Propagación unitaria (*unit propagation* [50]) con variables existenciales. Si una cláusula tiene un único literal de una variable existencial, le asignamos el valor que hace cierto al literal. Además, simplificamos el resto de cláusulas, eliminando las que tienen el mismo literal y quitando los literales de polaridad contraria (y comprobando si queda alguna cláusula vacía).

3. Cláusulas universales (caso particular de $\forall$*-reduction* [51]). Si alguna cláusula está formada únicamente por literales de variables universales, la fórmula es equivalente a $False$, ya que siempre habrá alguna asignación que haga falsa a la cláusula, y por tanto a la fórmula en CNF.

Las primeras dos estrategias hay que aplicarlas hasta que no encuentren más simplificaciones, se retroalimentan. Una propagación unitaria o eliminación de variable de polaridad única pueden hacer que aparezcan más cláusulas unitarias o variables con polaridad única; las cláusulas unitarias por las eliminaciones de literales; y las variables con polaridad única por las eliminaciones de cláusulas que aparte de la variable que provoca la eliminación pueden contener otros literales.

Por otro lado, la comprobación de cláusulas universales no modifica la fórmula dando pie a mayores simplificaciones por las demás estrategias. Además, las propagaciones unitarias no eliminan cláusulas universales porque solo actúan sobre variables existenciales. En cuanto a la primera estrategia, si todas las variables de una cláusula universal tuvieran polaridad única, sí que se eliminaría dicha cláusula universal, pero nos quedaríamos con una cláusula vacía y por tanto devolveríamos $False$ igualmente. Por todo esto, dejamos la comprobación de cláusulas universales para el final. También es perfectamente válido hacer este tipo de comprobaciones después de cada propagación unitaria y simplificación de variables de polaridad única, y dependiendo de la instancia, podría ser más o menos eficiente. Nosotros nos hemos decantado por llamar a esta comprobación una sola vez.

Cabe mencionar que en la experimentación (ver Capítulo 7) también hemos usado el preprocesador con el resolvedor Naive, para hacer una comparación más justa con el resolvedor basado en la FNI. Las primeras simplificaciones sencillas de variables repetidas o cláusulas que contienen la misma variable con ambas polaridades se ha hecho de manera ligeramente distinta dependiendo del resolvedor, ya que con el Naive no ordenamos cada cláusula según el valor absoluto de los literales, mientras que con el otro resolvedor sí como primer paso para transformar la matriz a R-CNF.

Preprocesamiento extra con absorción

Otro preprocesamiento posible (que no ha sido aplicado en la experimentación iterativa, sino en una prueba pequeña posterior, por haberlo pensado después), es la aplicación de la propiedad 14 de absorción. Si consideramos a las cláusulas como conjuntos de literales, vemos por conmutatividad y asociatividad (propiedades 7 y 9) que las cláusulas que son subconjuntos de otras mayores absorben a estas últimas (*subsumed clauses elimination* [52]). Por lo tanto, nos podemos quedar únicamente con esa cláusula pequeña, que es igual a la intersección entre ella y todas las cláusulas que contienen al menos todos sus literales. Esta simplificación no depende de las cuantificaciones de las variables, y se puede hacer antes que el preprocesamiento descrito anteriormente, ya que puede provocar la aparición de variables con única polaridad.

Optimización de la transformación a R-CNF

Otra mejora que no hemos aplicado en la propia experimentación, sino en un pequeño testeo posterior, es que la transformación de regularización lo podíamos optimizar un poco más usando un parámetro índice extra en vez de usar la operación `pop(0)` de las listas. Esta operación provoca copias innecesarias de las referencias a los últimos elementos debido a que no podemos tener huecos en una lista, sino que tiene que estar totalmente compactado y contiguo. Es decir, con `pop(0)` el elemento del índice 1 (referencia a un objeto) se copia en el índice 0; el del índice 2 en el 1; y así sucesivamente hasta el índice n y $n-1$, siendo n el número de elementos de la lista.

Aumento del límite de recursión

Por otro lado, aunque no sea una optimización en sí, cabe destacar que hemos tenido que aumentar el límite de recursión de 1000 por defecto de Python mediante el módulo `sys` [6] para las instancias más grandes. Para la mayoría 5000 ha sido suficiente, pero para dos lo hemos tenido que subir a un valor altísimo de 100 000 para que no dieran problemas hasta llegar al límite de tiempo de la experimentación.

Versión final

En la versión final nos quedamos con la configuración más óptima lograda en la experimentación iterativa que se documenta en el Capítulo 7. Como adelanto, nos quedamos con la representación de tuplas anidadas, memoización manual de los resultados (sin usar direcciones de memoria de las tuplas), las versiones de las operaciones y eliminaciones que requieren menor cantidad de llamadas (como lo pensamos a priori) y todas las optimizaciones de las que hemos hablado, salvando las excepciones indicadas.

[6] https://docs.python.org/3/library/sys.html

CAPÍTULO 7

Experimentación

En este capítulo explicaremos los experimentos realizados y los resultados obtenidos. Podemos dividir la experimentación realizada en dos tareas: por una parte, hemos ido afinando el resolvedor basado en la FNI implementando iterativamente las mejoras comentadas en el Capítulo 6, hasta conseguir una versión final. Y por otra parte, hemos realizado un benchmarking general con DepQBF, el resolvedor Naive, el Naive usando el preprocesador de CNFs y la versión final del basado en la FNI.

7.1. Benchmarks

En esta sección vamos a explicar las instancias de QBF que hemos utilizado para hacer las mediciones. En concreto, empezaremos indicando el repositorio donde se encontraban los datos; seguiremos con una descripción más detallada de las instancias en su conjunto (número de variables y cláusulas, número y orden de los tipos de cuantificadores, etc.); terminaremos con la disposición de los datos y los resultados en el repositorio de nuestro proyecto.

Fuente de las instancias

Las instancias que hemos usado en la experimentación salen del repositorio GitHub del resolvedor Cadet [1]. En nuestro caso, hemos renombrado las instancias para identificarlas con mayor comodidad. Además, hemos corregido alguna instancia que no respetaba del todo el formato QDIMACS, ya que no cuantificaba explícitamente todas las variables, sino que tenía un único bloque existencial con la variable más grande (como tratando de indicar que todas las variables son existenciales).

[1] https://github.com/MarkusRabe/cadet/tree/master/integration-tests

Descripción de las instancias

Tenemos 163 instancias, de los cuales hay una minoría de instancias de SAT en formato DIMACS, pero el resto están en QDIMACS con al menos un cuantificador universal. La mayoría no tiene un tamaño considerable. El conjunto de instancias está pensado para valorar la corrección de la implementación de un resolvedor de QBF, más que para medir la eficiencia. No obstante, hay unas cuantas decenas de instancias con un tamaño medio o alto (por ejemplo, DepQBF no ha podido resolver 7, pero otras que sí siguen siendo grandes), por lo que tenemos suficientes datos para hacer una primera comparación entre los resolvedores comentados.

Para ver con más detalle las características de cada instancia podemos mirar en el archivo `instance_information.csv` creado a través del *script* `instance_info_script.py`. Resumiendo las características de estas instancias, tenemos lo siguiente:

- **Número de variables y cláusulas**. La mayoría son de un tamaño menor, menos de 100 variables y cláusulas. No obstante, hay una cantidad suficiente de instancias de un tamaño medio, con cientos de variables y cláusulas. Además, hay unas cuantas con un número de variables y cláusulas del orden de varios miles o incluso decenas de miles.
- **Tamaño de cláusulas**. En comparación con el tamaño máximo que podrían tener (el número de variables), las medias y medianas de los tamaños de las cláusulas queda muy por debajo. La inmensa mayoría no llega a 10 literales por cláusula.
- **Bloques de cuantificadores**. La mayoría de las instancias tiene pocos bloques, entre 2 y 7. Tres instancias sobresalen, uno con 17 bloques y los otros con 43. En el caso de las instancias SAT tenemos un único bloque existencial. Por otro lado, en cuanto al orden de los tipos de bloques y la cantidad de variables que contiene cada bloque, tenemos que la gran mayoría de las instancias concentra muchas de las variables en el último bloque que es siempre existencial. Es normal que la mayoría de las variables en QBF, y especialmente el último bloque de cuantificadores, sean existenciales (sobre todo en instancias que modelan problemas prácticos). Esto puede entenderse de dos maneras:
  1. Lógica. Si predominan los cuantificadores universales, muchas cláusulas podrían quedar formadas solo por variables universales, lo que simplifica la fórmula a $False$. Además, si el último bloque fuera universal, la matriz debería haberse simplificado ya a una tautología para que la fórmula resultara satisfactible.
  2. Juego adversarial. Con bloques existenciales finales, las variables existenciales representan una estrategia reactiva: pueden depender de lo que haga el entorno (las variables universales). Si fuera al revés, las existenciales tendrían que fijarse de antemano y funcionar contra todos los valores posibles de las universales, lo cual es mucho más restrictivo.

Por tanto, especialmente debido a la configuración de los bloques de cuantificadores, sospechamos que el resolvedor Naive parte con ventaja sobre el resolvedor basado en la FNI. Justamente, el resolvedor Naive es bueno si puede llegar rápidamente al caso base de un único bloque de

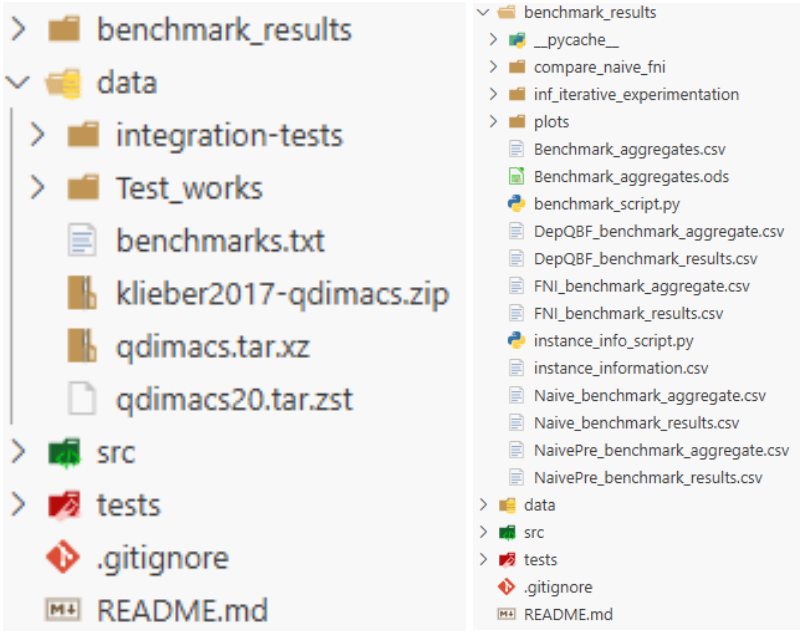

Figura 7.1: Directorios de las instancias y los resultados obtenidos en el repositorio del proyecto.

variables existenciales, donde llama a un resolvedor de SAT muy eficiente. Por el contrario, al resolvedor basado en la FNI le favorecen los casos donde hay más variables universales, ya que la fórmula estará en la FNI siempre que la variable más interna sea universal. Así, la eliminación de dichas variables es más eficiente que la de los existenciales: dos conjunciones en vez de una disyunción y una conjunción. Si está constantemente eliminando variables existenciales, ejecutará muchas disyunciones, con el riesgo de que el tamaño de la fórmula explote.

Repositorio del proyecto

En la Figura 7.1 vemos la estructura del directorio que contienen las instancias y el que guarda los resultados obtenidos durante la experimentación.

En el directorio `data` el subdirectorio `integration-tests` es el que contiene las instancias descritas previamente, las usadas en las pruebas. Además hemos incluido unas cuantas instancias más, junto con el archivo `benchmarks.txt` que explica de dónde salen, para posibles futuras experimentaciones. Por limitaciones de GitHub, que no deja subir archivos mayores de 100MB, hemos transferido los archivos comprimidos con dichas instancias extra a una carpeta de Google Drive, cuyo enlace hemos añadido a `benchmarks.txt`.

En el directorio `benchmark_results` tenemos los datos recopilados: información sobre las instancias; los resultados de cada resolvedor por cada instancia; resultados agregados por resolvedor; y un único archivo con los agregados más interesantes. Además, tenemos directorios adicionales con resultados de cada paso en la experimentación iterativa, los gráficos

realizados con los datos agregados, y una comparación sencilla entre el resolvedor Naive y el basado en la FNI. Aparte de los archivos `csv`, también dejamos los *scripts* de Python empleados para recopilar los resultados y la información sobre las instancias.

En las siguientes secciones explicaremos cada tarea realizada al recoger los datos, expondremos los resultados obtenidos, y reflexionaremos brevemente sobre ellos.

7.2. Experimentación iterativa

La experimentación iterativa consiste en que hemos empezado con una versión básica del resolvedor basado en la FNI que hemos ido afinando. Con cada una de las configuraciones, empezando desde la básica inicial, hemos recopilado el número de instancias resueltas y las medias del tiempo de CPU, el tiempo real y la cantidad de memoria utilizados. Si una configuración mejoraba la anterior, hemos mantenido dicha mejora en las siguientes configuraciones, salvo los casos de cambios incompatibles.

A continuación, vamos a explicar: las librerías de Python que hemos utilizado; estrategias concretas que hemos tomado para estabilizar los resultados y optimizar el tiempo de experimentación; los resultados con las configuraciones más destacables del resolvedor; y las conclusiones obtenidas.

Librerías de Python

El *script* principal que hemos usado en este apartado es `benchmark_script.py`. Para hacer las mediciones hemos utilizado las siguientes librerías:

- `psutil` [2]. Permite recopilar información de otro proceso mediante comunicación entre procesos que gestiona internamente. Nosotros como usuarios solamente tenemos que tomar periódicamente los datos que nos interesan con la interfaz que nos ofrece.
- `subprocess`. Para instanciar el proceso del resolvedor sobre la instancia elegida.
- `time`. Por un lado, hemos usado la función `sleep` para disminuir la frecuencia de recopilación de datos con `psutil`. Eso ha provocado que, especialmente para las instancias más pequeñas, el propio `sleep` incrementa artificialmente el tiempo real que utilizaría el resolvedor. Por tanto, hemos repetido cada medición usando la función homónima `time` para medir el tiempo real de resolución sin el coste de usar `psutil`.

Estabilización de los resultados

Además de lo anterior, hemos repetido la ejecución de cada resolvedor sobre cada instancia tres veces para hacer una media con los datos recogidos, para estabilizar un poco los resultados.

[2] https://psutil.readthedocs.io/en/latest/

Es decir, primero ejecutamos el resolvedor tres veces (un número no muy grande para el proceso iterativo) usando `psutil` para recopilar el tiempo de CPU y el gasto de memoria, y después otras tres veces usando solamente `time.time` para calcular el tiempo real.

Optimización de la experimentación

Hemos impuesto un tiempo límite de un minuto (pequeño para que la iteración no se eternizase). Si la primera (realmente cualquiera) de las ejecuciones da como resultado un error o que supera el tiempo límite, entonces evitamos hacer las siguientes ejecuciones para evitar perder tiempo con las instancias más complicadas.

Otra medida para aligerar el proceso de iteración ha sido limitar las instancias utilizadas. Para ello, antes de la experimentación iterativa hemos realizado la medición completa con DepQBF y el resolvedor Naive. Con dichos resultados más la información sobre los tamaños de las instancias hemos hecho el siguiente filtrado:

1. Hemos descartado las instancias que DepQBF no puede resolver, suponiendo que el nuestro tampoco podrá con ellas.
2. Hemos incluido las instancias que no han podido ser resueltas por el resolvedor Naive, independientemente de su tamaño.
3. Del resto, nos hemos quedado únicamente con las instancias que tenían más de 100 variables o más de 100 cláusulas.

En total, hemos utilizado 56 instancias de `integration-tests`, tratando de evitar instancias triviales o excesivamente difíciles.

Resultados de la experimentación iterativa

Después de explicar en qué ha consistido la experimentación iterativa, vamos a pasar a ver los resultados. En la Tabla 7.1 tenemos un resumen de los datos más interesantes. Primero, en cuanto a las columnas, mostramos el número de instancias resueltas y el número de instancias con los que se agotan los recursos (memoria o el límite de recursión) antes de llegar al tiempo límite. En todos los experimentos nos hemos asegurado de que el número de instancias que devuelven una respuesta incorrecta es 0, para estar seguros de la corrección de los algoritmos. Como sabemos que tenemos 56 instancias, el resto de ellas son las que han llegado al tiempo límite sin llegar a calcular un resultado. También tenemos el tiempo real (muy parecido al tiempo de CPU en todas las instancias) con penalización (es decir, cada instancia que no se ha podido resolver cuenta por dos veces el límite de tiempo disponible) y el máximo uso de memoria (o pico de memoria) de la ejecución. Por otro lado, en cuanto a las filas no hemos mostrado los resultados de todas las configuraciones en detalle. Hemos agrupado mejoras que no han dado gran resultado con otros que sí. Entre líneas horizontales tenemos los resultados que se centran únicamente sobre las 8 instancias de SAT (para testear la optimización del

uso del resolvedor de PySAT para este tipo de instancias). Las líneas cuya configuración va seguido por paréntesis conteniendo otras versiones indican que hemos considerado todas ellas, y que el de la izquierda es el mejor y cuyos datos estamos mostrando.

Configuración	Número de resueltos	Número de agostamiento de recursos	Tiempo real penalizado (s)	Pico de memoria (MB)
1	1	*4	117.86	1153.99
2-3	16	*4	86.09	472.35
4-5	18	*1	81.81	198.53
6-9	18	*1	81.80	93.49
**10 A	0/8	0	TIMEOUT	126.32
**10 B	8/8	0	2.34	41.35
11 A (B-C)	****24	*1	68.91	***85.65
12 B (D | A, C)	26	*0	65.20	***66.23
13 A (B)	26	0	65.34	***65.34
14	27	0	62.56	187.71
15	27	0	62.43	475.08
16	27	0	63.18	***300.11
18 (17)	27	0	62.41	617.94
19 (20)	27	0	62.49	184.04

Tabla 7.1: Resultados de la evaluación iterativa.

Notas:

* El número de agostamientos de recursos con esas configuraciones está un tanto viciado. A partir de las configuraciones 11, nos dimos cuenta que lo que fallaba era el límite de recursión. Al aumentarlo, a partir de la configuración 12A dejamos de tener errores.

** Como hemos comentado, estas mediciones están hechos con las 8 instancias de SAT de las instancias de integración (que resulta que también habían superado el filtrado para la experimentación iterativa), ya que nos centramos en la mejora del uso del resolvedor de PySAT para instancias SAT.

*** Los gastos de memoria de las configuraciones paralelas iniciales están viciadas, porque no estábamos sumando el uso de memoria de los subprocesos generados (a diferencia del tiempo de CPU). Vemos después en la configuración 16, que es la misma que la 14 pero añadiendo las operaciones paralelas más eficientes encontradas con las pruebas sobre 11, 12 y 13, que el pico de memoria pasa de 188MB a 300MB. Por tanto, por lógica y por esta última comparación cabe esperar que los gastos de memoria de las versiones paralelas sea realmente superior al de la versión en serie.

**** ¡Cuidado! A partir de esta configuración se utiliza el resolvedor de PySAT para las 8 instancias de SAT. Por tanto, para compararlo equitativamente con la versión en serie hasta la configuración 9, hay que restar 8 al número de instancias resueltas.

Descripción de las configuraciones

Vamos a describir brevemente cada una de las configuraciones y subrayar cómo se comparan con las configuraciones previas:

1 Versión básica, en serie, sin ninguna optimización especial (sí tiene *return* tempranos, pero poco más). Resultados muy malos, solo una instancia resuelta.

2-3 Pasamos a usar la versión de conjunción que en el caso de fórmulas con distinta profundidad ejecuta conjunción exclusivamente entre la fórmula pequeña y la tercera subfórmula de la grande. Además, la eliminación de las variables existenciales pasa a hacerse con una disyunción seguido de una conjunción. Mejora enorme, de una instancia resuelta a dos con el cambio en la conjunción y de dos a 16 con el cambio en la eliminación de variables existenciales.

4-5 Empleamos la simplificación de fórmulas y el preprocesador de CNFs cuantificadas. Mejora notable, cada una de ellas hace que resolvamos una instancia extra, y además reducimos el número de errores (es decir, la profundidad de la pila de llamadas).

6-9 *Cacheamos* nodos, usamos el operador `is` para comparar nodos y desactivamos la recolección de basura generacional (obviamos la configuración 8 que corresponde al uso del preprocesamiento de absorción incompleto). Pese a que las dos últimas mejoras tienen todo el sentido conceptual, la buena mejora en el gasto de memoria medido se debe exclusivamente al *cacheo* de los nodos.

10 En la configuración A usamos nuestro propio algoritmo basado en la FNI para tratar de resolver instancias SAT. No conseguimos resolver ninguna. En cambio, al delegar la labor al resolvedor de PySAT en la configuración B, pasamos a resolverlas todas. Es una mejora sustancial, limitada, eso sí, a este tipo de instancias.

11 Paralelizamos la operación de conjunción. La mejor opción ha sido olvidarnos de los *return* tempranos y paralelizar todas las operaciones posibles. No obstante, comparado con la configuración 9 en serie, y descontando el número de instancias resueltos por PySAT, vemos que es peor opción.

12 Paralelizamos la operación de disyunción. La mejor opción ha sido paralelizar únicamente la propia disyunción, no las llamadas a conjunción. La versión que se olvida de los *return* tempranos y paraleliza todo lo posible ha igualado a la versión anterior, pero requiere una mayor paralelización. Comparado con la versión en serie, tenemos resultados parecidos.

13 Tratamos de combinar las paralelizaciones de conjunción y disyunción. Obtenemos resultados idénticos a la configuración 12.

14 Volvemos a las versiones en serie, pero memoizando los resultados de las operaciones booleanas (manualmente con diccionarios Python). A costa de mayor uso de memoria, conseguimos resolver una instancia no SAT extra.

15 Memoizamos las direcciones de memoria de los nodos aprovechando que se cachean. Sorprendentemente, el coste de memoria aumenta considerablemente, sin que las demás métricas mejoren. Posiblemente porque necesitamos instanciar objetos `int` con valores muy grandes (ya que tienen valores de direcciones de memoria de 64 bits) que en la configuración anterior no hacen falta, sin que la mayor sencillez de las funciones *hash* de `int`s sobre tuplas se note.

16 Tratamos de combinar la paralelización de las disyunciones con la memoización de los resultados en los subprocesos. Gastamos más memoria sin mejorar los resultados de la configuración 14.

18 (17) Usamos el segundo tipo de representación de las fórmulas en R-CNF. No mejoramos el rendimiento, y gastamos bastante más memoria.

19 (20) Planteamos la versión final quitando todo el código *debug* y condicionales para probar las configuraciones. No mejoramos apenas la configuración 14 en la que se basa 19. Además, en la 20 probamos a usar las `lru_cache` de `functools` en vez de los diccionarios para *cachear* los resultados de las operaciones booleanas. Simplemente usa más memoria, por eso nos quedamos con la configuración 19.

Todos los resultados obtenidos y todas las configuraciones empleadas se pueden ver en el directorio `benchmark_results/inf_iterative_experimentation`.

Conclusiones de la experimentación

En conclusión, tenemos que la mejor versión alcanzada con la experimentación iterativa para nuestro resolvedor basado en la FNI tiene las siguientes características:

1. Es una versión en serie, no paralela. Utiliza *return* tempranos para evitar operaciones innecesarias.
2. Las versiones eficientes de conjunción y eliminación de variables existenciales son las esperadas, las que ejecutan menor cantidad de llamadas.

3. Simplifica las fórmulas después de cada operación booleana y utiliza el preprocesador de CNFs cuantificadas.

4. Evita la repetición de nodos.

5. Delega la resolución de instancias de SAT a PySAT.

6. Memoiza los resultados de las operaciones booleanas manualmente con diccionarios.

7. La representación de las matrices de las fórmulas en R-CNF es mediante tuplas anidadas, la más básica y natural.

7.3. Mediciones completas

En esta sección vamos a mostrar las mediciones completas que hemos realizado con DepQBF, el resolvedor Naive (con y sin preprocesamiento) y el resolvedor basado en la FNI (su versión final). Explicaremos los pocos cambios con respecto a la experimentación iterativa, los resultados obtenidos y las conclusiones alcanzadas.

Cambios a la experimentación iterativa

Las mediciones completas han sido prácticamente idénticas a las realizadas en la experimentación iterativa. Los cambios más importantes son tres: el límite de tiempo pasa a ser de 15 minutos; la ejecución con cada instancia se realiza 10 veces para estabilizar mejor los resultados; y se utilizan todas las instancias. Las demás estrategias de optimización de tiempo son iguales; principalmente, no repetimos ejecuciones sobre instancias que en un principio superan el límite de tiempo indicado. Incluso con esta optimización, cada una de las mediciones completas ha llevado unas cuantas horas. Por ejemplo, con el resolvedor FNI, la ejecución de `benchmark_script.py` casi alcanzó las 9 horas (con las demás no habíamos añadido todavía la medición del tiempo de ejecución de este *script)*. Por ello, resulta comprensible la dificultad de investigar el problema QBF y por qué su estudio progresa más lentamente que el de SAT. A parte de ello, en el directorio `benchmark_results` hemos guardado más datos por cada resolvedor que en la experimentación iterativa para cada configuración.

Resultados de las mediciones completas

En los archivos del directorio `benchmark_results` tenemos disponibles todos los datos recopilados: datos de la resolución de cada instancia por cada resolvedor; agregados por cada resolvedor; y una recopilación de los agregados más interesantes. En la Tabla 7.2 encontramos un resumen con los datos más subrayables.

Resolvedor	Número de resueltos	Porcentaje de resueltos	Tiempo real penalizado (s)	Pico de memoria (MB)
DepQBF	156	95.71 %	79.78	17.57
Naive	128	78.53 %	388.05	30.39
NaivePre	128	78.53 %	389.03	31.44
FNI	127	77.91 %	397.72	834.02

Tabla 7.2: Resultados de las mediciones completas.

Conclusiones de las mediciones completas

Como cabría esperar, ninguna de las implementaciones realizadas ha llegado al nivel de DepQBF, un resolvedor QBF *state-of-the-art*. En el caso del resolvedor Naive, el uso del preprocesador de CNFs no se ha notado en absoluto. Por último, el algoritmo central de este trabajo no ha llegado a igualar ni al algoritmo Naive, aunque tienen prácticamente el mismo rendimiento. En la Sección 7.5 compararemos más en detalle nuestras dos implementaciones.

7.4. Gráficos con los resultados finales

En esta sección vamos a visualizar los resultados obtenidos con gráficos. Vamos a empezar con los datos resumidos en la Tabla 7.2, y en el caso del uso de la memoria, con información adicional con el mínimo y máximo uso. Para ello, usamos simples gráficos de barras.

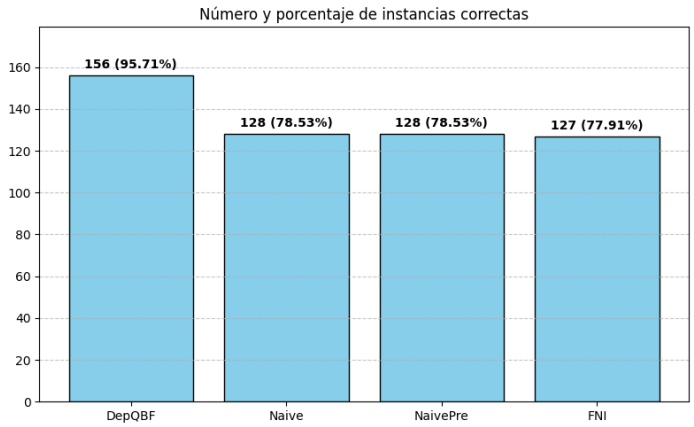

Figura 7.2: Número y porcentaje de instancias resueltas.

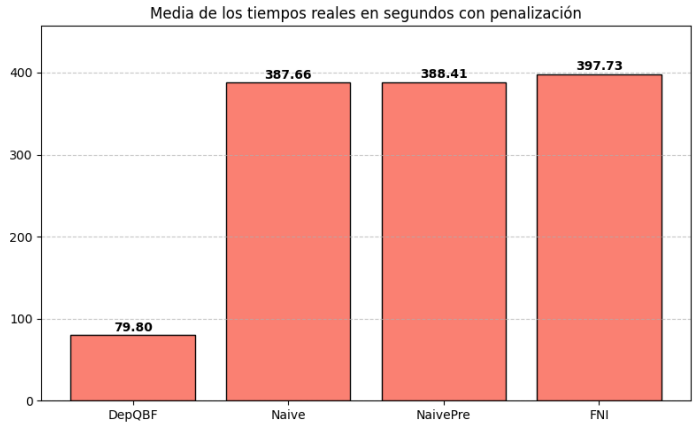

Figura 7.3: Media de los tiempos reales penalizados. Las instancias que no han sido resueltas valen por 2 veces el límite de tiempo, es decir, 1800 segundos o 30 minutos.

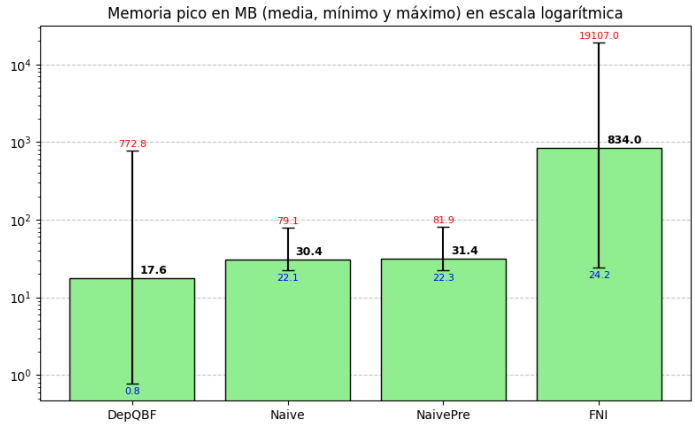

Figura 7.4: Media de los picos de uso de memoria, entre el mínimo y el máximo, y en escala logarítmica.

Gráfico de barras de los números de instancias resueltas

En la Figura 7.2 vemos que DepQBF está por encima de los otros resolvedores en cuanto el criterio más importante: el número de instancias resueltas. Los otros resolvedores que hemos implementado nosotros están a la par, aunque el Naive está una instancia por encima.

Gráfico de barras de los tiempos reales penalizados

En la Figura 7.3 tenemos un gráfico complementario de la Figura 7.2. En vez del número de instancias resueltas tenemos el tiempo real penalizado. Los tamaños relativos de las barras son inversos a los de la Figura 7.2, pero la conclusión es la misma.

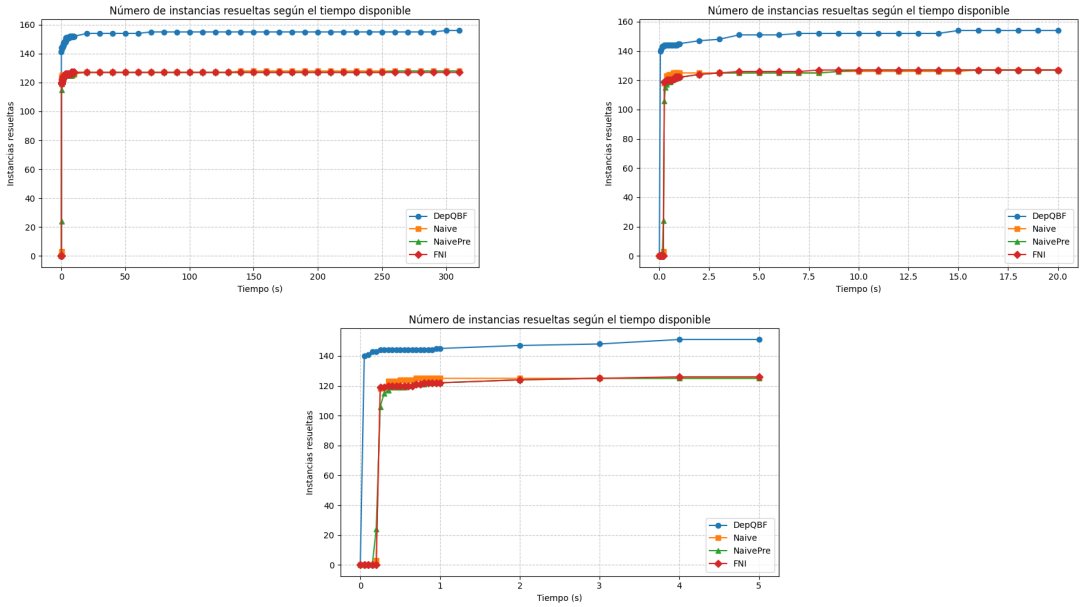

Figura 7.5: Número de instancias resueltas según el tiempo disponible. El tiempo máximo de resolución finalizada es de 300 segundos. Con cada subgráfico nos enfocamos cada vez más en los primeros segundos, donde terminan la mayoría de las resoluciones.

Gráfico de barras de los picos de memoria

En la Figura 7.4 vemos el consumo medio de memoria de cada resolvedor, encerrado entre el uso mínimo y máximo. Un detalle importante en el que fijarnos es que la escala que utiliza es logarítmica. Vemos que, también en este caso, DepQBF es la más optimizada, lo cual es esperable teniendo en cuenta que está implementado en C [53]. Nuestras implementaciones Naive están muy a la par de DepQBF, dado que son el algoritmo básico por el que vemos que el problema QBF está en PSPACE, como vimos en la Sección 4.1. Por el contrario, nuestro resolvedor basado en la FNI utiliza una cantidad de memoria exponencialmente superior, debido a que las fórmulas en R-CNF se codifican en estructuras tipo árbol o DAG, que con las operaciones de disyunción pueden explotar de tamaño.

Perfiles de rendimiento

Una vez acabado con los gráficos de barras, vamos con otro tipo de gráficos: los perfiles de rendimiento. En el eje horizontal tenemos el tiempo disponible del resolvedor. En el eje vertical tenemos el número de instancias resueltas. En la Figura 7.5 tenemos el perfil de rendimiento (tres vistas distintas, acercándonos cada vez más a los primeros segundos) de los resolvedores testeados con nuestras instancias de integración.

Como vemos, las instancias utilizadas no son las mejores para estimar la optimización y eficiencia de resolvedores de QBF. Tenemos pocas instancias de dificultad grande o moderada.

Todas los resolvedores solucionan la mayoría de las instancias con poco tiempo disponible. No obstante, se sigue notando la diferencia entre DepQBF y los demás, y que Naive y FNI están muy a la par.

7.5. Comparando los resolvedores Naive y FNI

Que el resolvedor Naive haya igualado o incluso superado ligeramente al resolvedor FNI es un poco decepcionante. Por eso, en esta sección vamos a profundizar un poco más en la comparación entre estos dos resolvedores. Para ello, en el directorio `compare_naive_fni` tenemos definido un *script* para identificar las instancias que Naive resuelve fácilmente y que FNI no puede resolver, y viceversa. En ese mismo directorio contamos con otro archivo de texto en el que aparecen las instancias encontradas junto con las siguiente reflexiones:

- Hemos visto que Naive resuelve una instancia más que FNI. No obstante, el conjunto de las instancias resueltas por Naive no incluye a todas las instancias resueltas por FNI. Esto es, difieren en el tipo de instancias que resuelven. Más concretamente, Naive resuelve cinco instancias que FNI no puede, y FNI resuelve cuatro que Naive no.
- El factor crítico que determina si uno de los dos resolvedores es más eficiente para una instancia no es el tamaño bruto de esa instancia (el número de variables y cláusulas) sino más bien el orden y el tipo de los cuantificadores.
    - El resolvedor Naive supera al FNI cuando hay pocos cuantificadores universales y éstos están lo más a la izquierda posible. Con este tipo de instancias nuestro resolvedor Naive puede llegar rápidamente al caso base de un único bloque existencial, y ahí llamar a un resolvedor de SAT eficiente. Así, la resolución de este tipo de instancias es como resolver varias instancias SAT.
    - Por otro lado, el resolvedor FNI no es tan bueno con estas instancias. Con este algoritmo, nos interesa que la fórmula esté precisamente en FNI la mayor cantidad de tiempo posible; es decir, estando la matriz siempre en R-CNF, nos interesa que el cuantificador más interno sea universal, ya que su eliminación son un par de conjunciones. En cambio, si la fórmula no está en FNI debido a que la cuantificación más interna es existencial, la eliminación de dicha variable implica realizar una disyunción seguido de una conjunción. Las disyunciones pueden hacer explotar el tamaño de la fórmula. Además, mientras estábamos desarrollando el algoritmo, con *logs* de *debugging* hemos visto que cuando una instancia queda atascada, la mayoría de las veces, por no decir siempre, es en una operación de disyunción. En resumen, las instancias donde el resolvedor FNI supera al Naive es donde hay una cantidad importante de variables cuantificadas universalmente.

Si queremos que el resolvedor FNI llegue a superar al Naive para la mayoría de las instancias, hay que encontrar alguna forma de optimizar la disyunción de fórmulas en R-CNF

o eliminar cuantificadores existenciales con fórmulas cuantificadas con matriz en R-CNF de una manera más eficiente. Lo segundo parece complicado, ya que la eliminación de variables existenciales ya es algo conceptualmente sencillo: una disyunción seguido de una conjunción. El problema está en el gran coste de la disyunción.

7.6. Últimas experimentaciones adicionales

Escribiendo la memoria y repasando el código nos dimos cuenta de que podíamos mejorar ciertas partes del código. Los problemas encontrados no parecían críticos, y los resultados obtenidos al introducir los cambios así lo demuestran. Concretamente, hemos tratado de optimizar la operación de regularización; usar el preprocesador de CNF cuantificadas de manera completa; añadir el preprocesamiento con la propiedad de absorción; y limpiar los nodos no alcanzables con la representación de tuplas anidadas de fórmulas en R-CNF.

Optimizando la operación de regularización

Primero, tratamos de mejorar la transformación de CNF a R-CNF para evitar el uso de `pop(0)` en las cláusulas. Lo intentamos de dos maneras, usando `deque` como cola [54] y usando un parámetro índice adicional. Elegimos las cuatro instancias mayores usadas en las mediciones completas. Los resultados están en `regularization_big_instances.txt`, conseguido a través del *script* `test_regularize.py`. En resumen, la versión más óptima es la que introduce el parámetro índice adicional. Eso sí, las mejoras en tiempo que conseguimos son de ciertas centésimas de segundo, que para estas instancias que no se han podido resolver en 15 minutos no arreglan nada.

Por otra parte, en el mismo archivo apreciamos también que al pasar de CNF a R-CNF, el tamaño en memoria de estas instancias más grandes se reduce considerablemente. No obstante, siguen ocupando cientos de MB, y el peligro de aumentar todavía más de tamaño sigue estando presente a causa de las disyunciones.

Preprocesador completo

En las pruebas de secciones anteriores no ejecutábamos todas las simplificaciones de variables de única polaridad y propagaciones unitarias posibles. Hacíamos una única ejecución de simplificación de variables de única polaridad, y después todas las propagaciones unitarias posibles, sin considerar que realmente se retroalimentan. Ajustando eso, vemos que los resultados que obtenemos en `test_final_inf.py` son los mismos (los encontramos en el archivo `last_additions_to_final_version.txt`). De hecho, el tiempo necesario para algunas instancias es marginalmente mayor, por unas décimas de segundo, lo cual no implica que no sea una versión mejor para otras instancias que se puedan simplificar todavía más gracias a esta retroalimentación entre dichas simplificaciones.

Preprocesamiento con absorción

En la experimentación iterativa descartamos el preprocesamiento de absorción, cuando lo que hacíamos era una versión limitada que sólo eliminaba cláusulas que tenían como prefijo otra cláusula de la fórmula. Usando plenamente la propiedad de absorción para eliminar todas las cláusulas que incluyen otras cláusulas de la fórmula independientemente del orden de los literales, vemos que obtenemos el mismo resultado (en el archivo `last_additions_to_final_version.txt`). En este caso los tiempos de ejecución también son ligeramente peores, por lo que descartamos esta adición definitivamente porque complica el algoritmo de manera innecesaria.

Limpieza de nodos

Para terminar, probamos a guardar los nodos manualmente en un `dict` de Python en vez de usar el decorador `lru_cache`, para poder hacer una limpieza periódica de los nodos que no se pueden alcanzar desde el nodo raíz de la matriz de la fórmula en R-CNF, al estilo de la segunda representación de fórmulas en R-CNF con identificadores. Como resultado, se reduce el gasto en memoria, pero se aumenta el tiempo de ejecución. De hecho, hemos tenido que aumentar el tiempo límite en `test_final_inf.py` de 10 a 15 segundos para poder seguir resolviendo la misma cantidad de instancias que podíamos resolver inicialmente, debido a una instancia de tamaño considerable.

CAPÍTULO 8

Seguimiento

En este capítulo resumimos el seguimiento hecho al proyecto, tocando los mismos puntos del Capítulo 2 de planificación una vez acabado el trabajo: objetivos realizados, tecnologías utilizadas, riesgos materializados, periodos de realización demorados y tiempo adicional invertido.

8.1. Objetivos

Como hemos comentado en las conclusiones (Sección 9.1), hemos cumplido los objetivos básicos propuestos al inicio del proyecto. Los resultados obtenidos no han sido ideales, aunque no dependían del todo de nosotros, sino también de las instancias de prueba elegidas y la eficacia del algoritmo basado en la FNI.

Por otra parte, el entregable del repositorio del código está bien documentado, con *type hints* en todas las principales funciones de la implementación y comentando con bastante profundidad cada una de dichas funciones.

8.2. Tecnologías

No hemos tenido que utilizar más sistemas de información o medios de comunicación que los planteados en la planificación (Sección 2.2.5). Lo mismo con el lenguaje de programación y las librerías que decidimos usar al principio.

8.3. Riesgos

Los principales riesgos que se han materializado en mayor o menor medida han sido estos:

- Planificación inicial optimista. Sobre todo en cuanto a los periodos de realización, así como el tiempo estimado de las tareas más importantes. Hemos ido replanificando qué tareas cumplir en cada semana. El orden de las tareas sí ha sido lo esperado.

- Tareas no consideradas en la planificación. Por ejemplo, la experimentación iterativa y todas las optimizaciones que hemos tratado de añadir al resolvedor basado en la FNI. Asimismo, la limpieza y reestructuración del repositorio escrito en Python ha sido más costoso de lo esperado.

- Obligaciones externas. Especialmente durante el periodo lectivo, con todos los trabajos y exámenes de la carrera. Arrastramos un poco de fatiga mental al verano, y procrastinamos un poco más de la cuenta. Podíamos haber adelantado más los periodos de realización, para ir con más tranquilidad. Afortunadamente, hemos terminado el proyecto con margen para la convocatoria de septiembre, salvo últimos apuros por no tener tan presente el plazo de solicitud de defensa de la matrícula.

8.4. Periodos de realización

Tenemos dos comentarios sobre los periodos de realización plasmados en el diagrama de Gantt 2.2:

1. El orden de las tareas realizadas ha sido lo esperado. Las relaciones entre los periodos de realización reales corresponden con los planeados.

2. Los periodos de realización se han alargado hasta la primera semana de septiembre, en concreto, los periodos de las tareas de seguimiento y control y la realización de los entregables. La implementación del resolvedor lo teníamos para la primera semana de agosto, pero la experimentación iterativa, las mediciones completas y la reestructuración y limpieza del repositorio han llevado más tiempo de lo esperado, de media una semana por cada una de dichas tareas. Esto ha provocado un mayor retraso en la escritura de la memoria y la realización del póster y las transparencias de presentación del proyecto.

8.5. Tiempo invertido

En la Tabla 8.1 tenemos la comparación entre el tiempo estimado y el tiempo invertido en cada tarea. El seguimiento y control, la planificación y la investigación han ido bien, hemos usado más o menos el tiempo esperado. Estas tareas no han sido los que han dado los mayores quebraderos de cabeza. En cuanto a los entregables, la memoria, las transparencias de presentación y el póster se han hecho dentro de un tiempo razonable. Si la realización de la memoria ha sido mentalmente pesado en algún punto, ha sido más por el periodo de realización que por el tiempo invertido. Por último, tenemos la parte más costosa del proyecto y en la que mayor desviación con respecto a la planificación inicial tiene: la implementación

del código (en el que incluimos la experimentación con los resolvedores). Al inicio de la implementación gastamos más tiempo de lo debido en el parser, teniendo en cuenta que luego nos hemos limitado a usar la versión sencilla que presupone la corrección de las instancias. El razonador Naive no supuso mayores complicaciones. Para terminar, el razonador FNI y su experimentación han sido con diferencia las tareas más costosas. Lo más cansino de la experimentación de resolvedores QBF es el tiempo que les lleva las instancias más difíciles. Por culpa de eso, a menos que tengamos tareas que hacer en paralelo, la experimentación se vuelve lenta. En la Tabla 8.1 no hemos incluido los tiempos de espera mientras hacíamos la medición de los resolvedores. Sino, el apartado de implementación tendría tiempos reales todavía más elevados.

Paquetes de trabajo			Estimadas (h)	Reales (h)
Gestión	*Planificación*		10	11
	Seguimiento y Control		20	19
Investigación	PSPACE y QBF		25	28
	Formatos: QDIMACS, QCIR...		15	13
	Razonadores: DepQBF, QuAbS...		20	12
	Benchmarks		20	15
Entregables	Memoria		60	70
	Presentación		20	17
	Póster		10	10
	Implementación	*Parser*	20	25
		Razonador Naive	30	32
		Razonador FNI	50	87
Total			**300**	**339**

Tabla 8.1: Estimación de tiempos frente al tiempo real invertido en cada tarea.

CAPÍTULO 9

Conclusiones

En este capítulo cerramos el proyecto haciendo unas reflexiones sobre el cumplimiento de los objetivos, los resultados finales obtenidos y el trabajo que ha quedado pendiente para el futuro.

9.1. Sobre los objetivos del proyecto

Hemos cumplido los objetivos básicos propuestos en la planificación (Sección 2.1.1):

- Nos hemos adentrado en el estudio del problema QBF, tanto investigando información como implementando un resolvedor Naive.
- Hemos implementado el resolvedor basado en la FNI, y hemos comprobado su corrección superando todas las instancias sencillas de las pruebas de integración. Hemos tratado de optimizarlo todo lo posible, aprendiendo en el proceso sobre el lenguaje de implementación Python y usando muchas configuraciones distintas hasta dar con una implementación final lo más eficiente posible.
- Hemos realizado las experimentaciones necesarias para sacar conclusiones claras comparando un resolvedor *state-of-the-art* (DepQBF) con las implementaciones realizadas, así como estas últimas entre sí.

9.2. Sobre los resultados finales

Como cabría esperar por el lenguaje de alto nivel (Python) usado para prototipar los resolvedores y la dificultad del problema QBF, no hemos podido igualar o acercarnos mucho al rendimiento de DepQBF. Por otro lado, hemos comprobado que el algoritmo basado en la FNI

no supera con claridad al algoritmo Naive. De hecho, en las instancias de prueba utilizadas, el resolvedor Naive consigue resolver una más. Además, por no dejar el proyecto simplemente con ese mal resultado, hemos comparado las instancias que Naive resuelve con facilidad y que FNI no puede, y viceversa. Así, hemos diagnosticado el problema del algoritmo FNI: la eliminación de bloques existenciales, que requiere el uso de disyunciones de fórmulas en R-CNF, que es el gran cuello de botella del algoritmo.

9.3. Trabajo futuro

Aquí está una lista de ideas para probar distintas implementaciones, o por donde deberían seguir las investigaciones para tratar de refinar el propio algoritmo:

- Probar con Prolog [1], Haskell [2] u otros lenguajes declarativos de alto nivel, que por su naturaleza ya aprovechan estructuras de datos inmutables, como hemos hecho nosotros al usar tuplas de Python como nodos de las codificaciones de fórmulas en R-CNF.
- Probar con más estrategias de preprocesamiento de las matrices en CNF antes de regularizarlos.
- Pensar en más simplificaciones de fórmulas en R-CNF: investigar más la estructura de los árboles de las instancias más costosas, reestructurar a árboles más fáciles de computar, inspirarnos por las operaciones de preprocesamiento de las fórmulas en CNF, etc. Especialmente, tratar de paliar el gran coste, tanto de tiempo como memoria, de las disyunciones.
- Investigar en profundidad los resolvedores de QBF *state-of-the-art*. Ver si las ideas de sus algoritmos se pueden incorporar al basado en la FNI, o si son claramente superiores al algoritmo que hemos tratado de desarrollar en este proyecto.
- Relacionado con el punto anterior, investigar sobre los formatos QAIGER y QCIR, a ver si dan pie a mayores optimizaciones. En dicho caso, implementar *parsers* de estos formatos.
- Una vez tengamos un prototipo que supere en todas las instancias al resolvedor Naive y se acerque más al rendimiento de DepQBF, probar a implementar el algoritmo en un lenguaje de bajo nivel como C o C++ [3], para ver hasta que punto llega a optimizarse y si consigue rivalizar con los mejores.

[1] https://www.swi-prolog.org/
[2] https://www.haskell.org/
[3] https://cplusplus.com/

APÉNDICE A

Operaciones duales de fórmulas en R-DNF

A.1. Conjunción y disyunción de fórmulas en R-DNF

Teorema 7. *(Disyunción de fórmulas R-DNF) La disyunción de dos fórmulas R-DNF se puede transformar a otra R-DNF equivalente.*

Demostración. Tenemos que demostrar que para dos fórmulas ϕ_1 y ϕ_2 en R-DNF donde $vars(\phi_1) \cup vars(\phi_2) = \{v_1, ..., v_n\} = \overline{v}$, existe una fórmula ϕ en R-DNF tal que $vars(\phi) = \overline{v}$ y $\phi \equiv \phi_1 \vee \phi_2$.

Para ello, vamos a plantear una inducción sobre n.

Cuando $n = 0$ (ϕ_1 y ϕ_2 son T o F) los casos son triviales por identidad y dominación (propiedades 1 y 5).

Para el caso inductivo ($n > 0$), asumimos que, para cualquier par de fórmulas ψ_1 y ψ_2 en R-DNF, siempre existe una fórmula ψ en R-DNF tal que $\psi \equiv \psi_1 \vee \psi_2$ con $vars(\psi) = vars(\psi_1) \cup vars(\psi_2) = \overline{v}$. Esta es nuestra hipótesis de inducción (H.I.).

Tenemos que demostrar que para cualquier par de fórmulas ϕ_1 y ϕ_2 en R-DNF, tales que $vars(\phi_1) \cup vars(\phi_2) = \overline{v} \cup \{v_{n+1}\}$, existe una fórmula ϕ en R-DNF tal que $vars(\phi) = \overline{v} \cup \{v_{n+1}\}$ y $\phi \equiv \phi_1 \vee \phi_2$.

Consideramos que $vars(\phi_1) = \{v_{11}, ..., v_{1m_1}\}$ y $vars(\phi_2) = \{v_{21}, ..., v_{2m_2}\}$. Los casos donde $m_1 = 0$ o $m_2 = 0$ (alguna de las dos fórmulas es igual a T o F) son triviales por identidad y dominación (propiedades 1 y 5). Para el resto de casos y sin perder generalidad por conmutatividad (propiedad 7) consideramos dos opciones: a) $v_{1m_1} = v_{2m_2} = v_n$ y b) $v_{1m_1} = v_n > v_{2m_2}$.

En el caso a), por la Definición 4 tenemos que

$$\phi_1 \vee \phi_2 \;=\; ((\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi_{13}) \vee ((\neg v_n \wedge \phi_{21}) \vee (v_n \wedge \phi_{22}) \vee \phi_{23})$$

para ciertas fórmulas en R-DNF tales que $\bigcup_{i=1}^{2} \bigcup_{j=1}^{3} vars(\phi_{ij}) = \overline{v}$. Por tanto:

$$\begin{array}{rcl}
\phi_1 \vee \phi_2 & = & ((\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi_{13}) \vee ((\neg v_n \wedge \phi_{21}) \vee (v_n \wedge \phi_{22}) \vee \phi_{23}) \\
& \overset{(7;9)}{\equiv} & ((\neg v_n \wedge \phi_{11}) \vee (\neg v_n \wedge \phi_{21})) \vee ((v_n \wedge \phi_{12}) \vee (v_n \wedge \phi_{22})) \vee (\phi_{13} \vee \phi_{23}) \\
& \overset{(12)}{\equiv} & (\neg v_n \wedge (\phi_{11} \vee \phi_{21})) \vee (v_n \wedge (\phi_{12} \vee \phi_{22})) \vee (\phi_{13} \vee \phi_{23}) \\
& \overset{(H.I.)}{\equiv} & (\neg v_n \wedge \phi_1') \vee (v_n \wedge \phi_2') \vee \phi_3' \\
& \overset{(Def.4)}{=} & \phi
\end{array}$$

En el caso b), también por la Definición 4 sabemos que

$$\phi_1 \vee \phi_2 \;=\; ((\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi_{13}) \vee \phi_2$$

para unas fórmulas en R-DNF tales que $(vars(\phi_2) \cup \bigcup_{i=1}^{3} vars(\phi_{1i})) = \overline{v}$. Por ende:

$$\begin{array}{rcl}
\phi_1 \vee \phi_2 & = & ((\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi_{13}) \vee \phi_2 \\
& \overset{(9)}{\equiv} & (\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee (\phi_{13} \vee \phi_2) \\
& \overset{(H.I.)}{\equiv} & (\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi' \\
& \overset{(Def.4)}{=} & \phi
\end{array}$$

Alternativamente:

$$\begin{array}{rcl}
\phi_1 \vee \phi_2 & = & ((\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi_{13}) \vee \phi_2 \\
& \overset{(3)}{\equiv} & ((\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi_{13}) \vee \phi_2 \vee \phi_2 \\
& \overset{(2)}{\equiv} & ((\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi_{13}) \vee \phi_2 \vee (\phi_2 \wedge true) \\
& \overset{(18)}{\equiv} & ((\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi_{13}) \vee \phi_2 \vee (\phi_2 \wedge (\neg v_n \vee v_n)) \\
& \overset{(12)}{\equiv} & ((\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi_{13}) \vee \phi_2 \vee (\phi_2 \wedge \neg v_n) \vee (\phi_2 \wedge v_n) \\
& \overset{(7,8,9)}{\equiv} & ((\neg v_n \wedge \phi_{11}) \vee (\neg v_n \wedge \phi_2)) \vee ((v_n \wedge \phi_{12}) \vee (v_n \wedge \phi_2)) \vee (\phi_{13} \vee \phi_2) \\
& \overset{(12)}{\equiv} & (\neg v_n \wedge (\phi_{11} \vee \phi_2)) \vee (v_n \wedge (\phi_{12} \vee \phi_2)) \vee (\phi_{13} \vee \phi_2) \\
& \overset{(H.I.)}{\equiv} & (\neg v_n \wedge \phi_1') \vee (v_n \wedge \phi_2') \vee \phi_3' \\
& \overset{(Def.4)}{=} & \phi
\end{array}$$

□

Teorema 8. *(Conjunción de fórmulas R-DNF) La conjunción de dos fórmulas R-DNF se puede transformar a otra fórmula R-DNF equivalente.*

Demostración. Tenemos que demostrar que para cualquier par de fórmulas ϕ_1 y ϕ_2 en R-DNF, tales que $vars(\phi_1) \cup vars(\phi_2) = \{v_1, ..., v_n\} = \overline{v}$, existe una fórmula ϕ en R-DNF tal que $vars(\phi) = \overline{v}$ y $\phi \equiv \phi_1 \wedge \phi_2$.

Para esto, procederemos por inducción sobre n.

Cuando $n = 0$ (ϕ_1 y ϕ_2 son T o F) los casos son triviales por identidad y dominación (propiedades 2 y 6).

Para el caso inductivo ($n > 0$), asumimos que, para cualquier par de fórmulas ψ_1 y ψ_2 en R-DNF, siempre existe una fórmula ψ en R-DNF tal que $\psi \equiv \psi_1 \wedge \psi_2$ con $vars(\psi) = vars(\psi_1) \cup vars(\psi_2) = \overline{v}$. Esta es nuestra hipótesis de inducción (H.I.).

Tenemos que demostrar que para cualquier par de fórmulas ϕ_1 y ϕ_2 en R-DNF, tales que $vars(\phi_1) \cup vars(\phi_2) = \overline{v} \cup \{v_{n+1}\}$, existe una fórmula ϕ en R-DNF tal que $vars(\phi) = \overline{v} \cup \{v_{n+1}\}$ y $\phi \equiv \phi_1 \wedge \phi_2$.

Consideramos que $vars(\phi_1) = \{v_{11}, ..., v_{1m_1}\}$ y $vars(\phi_2) = \{v_{21}, ..., v_{2m_2}\}$. Los casos donde $m_1 = 0$ o $m_2 = 0$ (alguna de las dos fórmulas es igual a T o F) son triviales por identidad y dominación (propiedades 2 y 6). Para el resto de casos y sin perder generalidad por conmutatividad (propiedad 8) consideramos dos opciones: a) $v_{1m_1} = v_{2m_2} = v_n$ y b) $v_{1m_1} = v_n > v_{2m_2}$.

En el caso a), por Definición 4 tenemos que

$$\phi_1 \wedge \phi_2 \;=\; ((\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi_{13}) \wedge ((\neg v_n \wedge \phi_{21}) \vee (v_n \wedge \phi_{22}) \vee \phi_{23})$$

para unas fórmulas en R-DNF tales que $\bigcup_{i=1}^{2} \bigcup_{j=1}^{3} vars(\phi_{ij}) = \overline{v}$. Por tanto:

$$\begin{aligned}
\phi_1 \wedge \phi_2 \quad &= \quad ((\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi_{13}) \wedge ((\neg v_n \wedge \phi_{21}) \vee (v_n \wedge \phi_{22}) \vee \phi_{23}) \\
&\overset{(12)}{\equiv} \quad (\neg v_n \wedge \phi_{11} \wedge ((\neg v_n \wedge \phi_{21}) \vee (v_n \wedge \phi_{22}) \vee \phi_{23})) \vee \\
&\qquad (v_n \wedge \phi_{12} \wedge ((\neg v_n \wedge \phi_{21}) \vee (v_n \wedge \phi_{22}) \vee \phi_{23})) \vee \\
&\qquad (\phi_{13} \wedge ((\neg v_n \wedge \phi_{21}) \vee (v_n \wedge \phi_{22}) \vee \phi_{23})) \\
&\overset{(12)}{\equiv} \quad (\neg v_n \wedge \phi_{11} \wedge \neg v_n \wedge \phi_{21}) \vee (\neg v_n \wedge \phi_{11} \wedge v_n \wedge \phi_{22}) \vee \\
&\qquad (\neg v_n \wedge \phi_{11} \wedge \phi_{23}) \vee (v_n \wedge \phi_{12} \wedge \neg v_n \wedge \phi_{21}) \vee \\
&\qquad (v_n \wedge \phi_{12} \wedge v_n \wedge \phi_{22}) \vee (v_n \wedge \phi_{12} \wedge \phi_{23}) \vee (\phi_{13} \wedge \neg v_n \wedge \phi_{21}) \vee \\
&\qquad (\phi_{13} \wedge v_n \wedge \phi_{22}) \vee (\phi_{13} \wedge \phi_{23}) \\
&\overset{(8)}{\equiv} \quad (\neg v_n \wedge \neg v_n \wedge \phi_{11} \wedge \phi_{21}) \vee (\neg v_n \wedge v_n \wedge \phi_{11} \wedge \phi_{22}) \vee \\
&\qquad (\neg v_n \wedge \phi_{11} \wedge \phi_{23}) \vee (v_n \wedge \neg v_n \wedge \phi_{12} \wedge \phi_{21}) \vee \\
&\qquad (v_n \wedge v_n \wedge \phi_{12} \wedge \phi_{22}) \vee (v_n \wedge \phi_{12} \wedge \phi_{23}) \vee (\neg v_n \wedge \phi_{13} \wedge \phi_{21}) \vee
\end{aligned}$$

$$
\begin{aligned}
& (v_n \wedge \phi_{13} \wedge \phi_{22}) \vee (\phi_{13} \wedge \phi_{23}) \\
\overset{(4,16)}{\equiv} \quad & (\neg v_n \wedge \phi_{11} \wedge \phi_{21}) \vee (false \wedge \phi_{11} \wedge \phi_{22}) \vee (\neg v_n \wedge \phi_{11} \wedge \phi_{23}) \vee \\
& (false \wedge \phi_{12} \wedge \phi_{21}) \vee (v_n \wedge \phi_{12} \wedge \phi_{22}) \vee (v_n \wedge \phi_{12} \wedge \phi_{23}) \vee \\
& (\neg v_n \wedge \phi_{13} \wedge \phi_{21}) \vee (v_n \wedge \phi_{13} \wedge \phi_{22}) \vee (\phi_{13} \wedge \phi_{23}) \\
\overset{(6,1)}{\equiv} \quad & (\neg v_n \wedge \phi_{11} \wedge \phi_{21}) \vee (\neg v_n \wedge \phi_{11} \wedge \phi_{23}) \vee \\
& (v_n \wedge \phi_{12} \wedge \phi_{22}) \vee (v_n \wedge \phi_{12} \wedge \phi_{23}) \vee \\
& (\neg v_n \wedge \phi_{13} \wedge \phi_{21}) \vee (v_n \wedge \phi_{13} \wedge \phi_{22}) \vee (\phi_{13} \wedge \phi_{23}) \\
\overset{(7)}{\equiv} \quad & (\neg v_n \wedge \phi_{11} \wedge \phi_{21}) \vee (\neg v_n \wedge \phi_{11} \wedge \phi_{23}) \vee (\neg v_n \wedge \phi_{13} \wedge \phi_{21}) \vee \\
& (v_n \wedge \phi_{12} \wedge \phi_{22}) \vee (v_n \wedge \phi_{12} \wedge \phi_{23}) \vee (v_n \wedge \phi_{13} \wedge \phi_{22}) \vee \\
& (\phi_{13} \wedge \phi_{23}) \\
\overset{(12)}{\equiv} \quad & (\neg v_n \wedge ((\phi_{11} \wedge \phi_{21}) \vee (\phi_{11} \wedge \phi_{23}) \vee (\phi_{13} \wedge \phi_{21}))) \vee \\
& (v_n \wedge ((\phi_{12} \wedge \phi_{22}) \vee (\phi_{12} \wedge \phi_{23}) \vee (\phi_{13} \wedge \phi_{22}))) \vee \\
& (\phi_{13} \wedge \phi_{23}) \\
\overset{(12)}{\equiv} \quad & (\neg v_n \wedge ((\phi_{11} \wedge (\phi_{21} \vee \phi_{23})) \vee (\phi_{13} \wedge \phi_{21}))) \vee \\
& (v_n \wedge ((\phi_{12} \wedge (\phi_{22} \vee \phi_{23})) \vee (\phi_{13} \wedge \phi_{22}))) \vee \\
& (\phi_{13} \wedge \phi_{23}) \\
\overset{(Teo.7)}{\equiv} \quad & (\neg v_n \wedge ((\phi_{11} \wedge \phi'_{11}) \vee (\phi_{13} \wedge \phi_{21}))) \vee \\
& (v_n \wedge ((\phi_{12} \wedge \phi'_{21}) \vee (\phi_{13} \wedge \phi_{22}))) \vee \\
& (\phi_{13} \wedge \phi_{23}) \\
\overset{(H.I.)}{\equiv} \quad & (\neg v_n \wedge (\phi'_{12} \vee \phi'_{13})) \vee (v_n \wedge (\phi'_{22} \vee \phi'_{23})) \vee \phi'_3 \\
\overset{(Teo.7)}{\equiv} \quad & (\neg v_n \wedge \phi'_{14}) \vee (v_n \wedge \phi'_{24}) \vee \phi'_3 \\
\overset{(Def.4)}{=} \quad & \phi
\end{aligned}
$$

En el caso b), también por Definición 4 sabemos que

$$\phi_1 \wedge \phi_2 \;=\; ((\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi_{13}) \wedge \phi_2$$

para unas fórmulas R-DNF tales que $(vars(\phi_2) \cup \bigcup_{i=1}^{3} vars(\phi_{1i})) = \overline{v}$. Por ende:

$$
\begin{aligned}
\phi_1 \wedge \phi_2 \quad = \quad & ((\neg v_n \wedge \phi_{11}) \vee (v_n \wedge \phi_{12}) \vee \phi_{13}) \wedge \phi_2 \\
\overset{(12)}{\equiv} \quad & (\neg v_n \wedge \phi_{11} \wedge \phi_2) \vee (v_n \wedge \phi_{12} \wedge \phi_2) \vee (\phi_{13} \wedge \phi_2) \\
\overset{(H.I.)}{\equiv} \quad & (\neg v_n \wedge \phi'_1) \vee (v_n \wedge \phi'_2) \vee \phi'_3 \\
\overset{(Def.4)}{=} \quad & \phi
\end{aligned}
$$

□

A.2. Eliminación de variables con matrices en R-DNF

Teorema 9. *La fórmula en FNI*

$$\overline{Q}\overline{v}\ \exists v\ \phi$$

puede transformarse a una fórmula proposicional cuantificada equivalente de la forma

$$\overline{Q}\overline{v}\ \psi$$

donde ψ está en R-DNF con $vars(\psi) = \overline{v}$.

Demostración. Por la Definición 6, tenemos que

$$\phi\ =\ (\neg v \wedge \phi_1) \vee (v \wedge \phi_2) \vee \phi_3$$

para algunas fórmulas ϕ_1, ϕ_2 y ϕ_3 en R-DNF tales que $vars(\phi_1) = vars(\phi_2) = vars(\phi_3) = \overline{v}$. Por ello:

$$\begin{array}{lcl}
\overline{Q}\overline{v}\ \exists v\ \phi & = & \overline{Q}\overline{v}\ \exists v\ (\ (\neg v \wedge \phi_1) \vee (v \wedge \phi_2) \vee \phi_3\ ) \\
& \overset{(24)}{\equiv} & \overline{Q}\overline{v}\ (\ \exists v\ (\neg v \wedge \phi_1) \vee \exists v\ (v \wedge \phi_2) \vee \exists v\ \phi_3\ ) \\
& \overset{(Teo.3)}{\equiv} & \overline{Q}\overline{v}\ (\ \phi_1 \vee \phi_2 \vee \exists v\ \phi_3\ ) \\
& \overset{(26)}{\equiv} & \overline{Q}\overline{v}\ (\ \phi_1 \vee \phi_2 \vee \phi_3\ ) \\
& \overset{(Teo.7)}{\equiv} & \overline{Q}\overline{v}\ \psi
\end{array}$$

□

Teorema 10. *Dada una fórmula proposicional en R-DNF ϕ tal que $vars(\phi) = \overline{v} \cup \{v\}$, la fórmula QBF en forma normal prenexa*

$$\overline{Q}\overline{v}\ \forall v\ \phi$$

puede ser transformada a una fórmula proposicional cuantificada equivalente de la forma

$$\overline{Q}\overline{v}\ \psi$$

donde ψ está en R-DNF y $vars(\psi) = \overline{v}$.

Demostración. Por la definición 4, tenemos que

$$\phi\ =\ (\neg v \wedge \phi_1) \vee (v \wedge \phi_2) \vee \phi_3$$

para unas fórmulas ϕ_1, ϕ_2 y ϕ_3 en R-DNF tales que $vars(\phi_1) = vars(\phi_2) = vars(\phi_3) = \overline{v}$. Por tanto:

$$
\begin{aligned}
\overline{Q}\overline{v}\,\forall v\ \phi \quad &= \quad \overline{Q}\overline{v}\,\forall v\ (\ (\neg v \wedge \phi_1) \vee (v \wedge \phi_2) \vee \phi_3\ ) \\
&\overset{(1)}{\equiv} \quad \overline{Q}\overline{v}\,\forall v\ (\ false \vee (\neg v \wedge \phi_1) \vee false \vee (v \wedge \phi_2) \vee \phi_3\ ) \\
&\overset{(16)}{\equiv} \quad \overline{Q}\overline{v}\,\forall v\ (\ (\neg v \wedge v) \vee (\neg v \wedge \phi_1) \vee (\neg v \wedge v) \vee (v \wedge \phi_2) \vee \phi_3\ ) \\
&\overset{(11)}{\equiv} \quad \overline{Q}\overline{v}\,\forall v\ (\ ((\neg v \vee (\neg v \wedge \phi_1)) \wedge (v \vee (\neg v \wedge \phi_1))) \vee \\
&\qquad\qquad ((\neg v \vee (v \wedge \phi_2)) \wedge (v \vee (v \wedge \phi_2))) \vee \phi_3\ ) \\
&\overset{(13)}{\equiv} \quad \overline{Q}\overline{v}\,\forall v\ (\ (\neg v \wedge (v \vee (\neg v \wedge \phi_1))) \vee \\
&\qquad\qquad ((\neg v \vee (v \wedge \phi_2)) \wedge v)) \vee \phi_3\ ) \\
&\overset{(11)}{\equiv} \quad \overline{Q}\overline{v}\,\forall v\ (\ (\neg v \wedge (v \vee \neg v) \wedge (v \vee \phi_1)) \vee \\
&\qquad\qquad ((\neg v \vee v) \wedge (\neg v \vee \phi_2) \wedge v) \vee \phi_3\ ) \\
&\overset{(18,2)}{\equiv} \quad \overline{Q}\overline{v}\,\forall v\ (\ (\neg v \wedge (v \vee \phi_1)) \vee ((\neg v \vee \phi_2) \wedge v) \vee \phi_3\ ) \\
&\overset{(11)}{\equiv} \quad \overline{Q}\overline{v}\,\forall v\ (\ (\neg v \vee ((\neg v \vee \phi_2) \wedge v) \vee \phi_3) \wedge (v \vee \phi_1 \vee \\
&\qquad\qquad ((\neg v \vee \phi_2) \wedge v) \vee \phi_3)\ ) \\
&\overset{(11)}{\equiv} \quad \overline{Q}\overline{v}\,\forall v\ (\ (((\neg v \vee \neg v \vee \phi_2) \wedge (\neg v \vee v)) \vee \phi_3) \wedge \\
&\qquad\qquad (((v \vee \phi_1 \vee \neg v \vee \phi_2) \wedge (v \vee \phi_1 \vee v)) \vee \phi_3)\ ) \\
&\overset{(18,5,2)}{\equiv} \quad \overline{Q}\overline{v}\,\forall v\ (\ (\neg v \vee \neg v \vee \phi_2 \vee \phi_3) \wedge (v \vee \phi_1 \vee v \vee \phi_3)\ ) \\
&\overset{(3)}{\equiv} \quad \overline{Q}\overline{v}\,\forall v\ (\ (\neg v \vee \phi_2 \vee \phi_3) \wedge (v \vee \phi_1 \vee \phi_3)\ )
\end{aligned}
$$

Primera opción, aplicar primero dos disyunciones entre R-DNFs y después distribuir el cuantificador universal eliminando la variable más interna:

$$
\begin{aligned}
&\overset{(Teo.7)}{\equiv} \quad \overline{Q}\overline{v}\,\forall v\ (\ (\neg v \vee \psi_1) \wedge (v \vee \psi_2)\ ) \\
&\overset{(25)}{\equiv} \quad \overline{Q}\overline{v}\ (\ \forall v\ (\neg v \vee \psi_1) \wedge \forall v\ (v \vee \psi_2)\ ) \\
&\overset{(Teo.4)}{\equiv} \quad \overline{Q}\overline{v}\ (\ \psi_1 \wedge \psi_2\ ) \\
&\overset{(Teo.8)}{\equiv} \quad \overline{Q}\overline{v}\ \psi
\end{aligned}
$$

Y segunda opción, aplicar primero la distribución del cuantificador y la eliminación de la variable, para después ahorrarnos una disyunción con la propiedad distributiva inversa con el

ϕ_3 común:

$$
\begin{aligned}
&\overset{(25)}{\equiv} && \overline{Q}\overline{v}\,\forall v\ (\ \forall v(\neg v \vee \phi_2 \vee \phi_3) \wedge \forall(v \vee \phi_1 \vee \phi_3)\ ) \\
&\overset{(Teo.4)}{\equiv} && \overline{Q}\overline{v}\ (\ (\phi_1 \vee \phi_3) \wedge (\phi_2 \vee \phi_3)\ ) \\
&\overset{(11,7)}{\equiv} && \overline{Q}\overline{v}\ (\ (\phi_1 \wedge \phi_2) \vee \phi_3\ ) \\
&\overset{(Teo.8)}{\equiv} && \overline{Q}\overline{v}\ (\psi_1 \vee \phi_3) \\
&\overset{(Teo.7)}{\equiv} && \overline{Q}\overline{v}\ \psi
\end{aligned}
$$

□

Bibliografía

[1] P (clase de complejidad). *Wikipedia*.

[2] Clases de complejidad P y NP. *Wikipedia*.

[3] Rodolfo Nieves. La complejidad del problema del milenio P vs NP y un algoritmo para su solución. *Niböe*, 2022.

[4] Lane A Hemaspaandra. Sigact news complexity theory column 36. *ACM SIGACT News*, 33(2):34–47, 2002.

[5] Bilal Aamir. P Vs NP Problem In A Nutshell. *Medium*, 2019.

[6] Boolean satisfiability problem. *Wikipedia*.

[7] Lógica proposicional. *Wikipedia*.

[8] Weiwei Gong and Xu Zhou. A survey of SAT solver. In *AIP Conference Proceedings*, volume 1836. AIP Publishing, 2017.

[9] Jussi Rintanen. Planning with specialized SAT solvers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 25, pages 1563–1566, 2011.

[10] Malay Ganai and Aarti Gupta. *SAT-based scalable formal verification solutions*. Springer, 2007.

[11] Artur Niewiadomski, Piotr Switalski, Teofil Sidoruk, and Wojciech Penczek. Applying modern SAT-solvers to solving hard problems. *Fundamenta Informaticae*, 165(3-4):321–344, 2019.

[12] Curtis Bright, Jürgen Gerhard, Ilias Kotsireas, and Vijay Ganesh. Effective problem solving using SAT solvers. In *Maple Conference*, pages 205–219. Springer, 2019.

[13] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 244–257. Springer, 2009.

[14] Steve Dai, Gai Liu, and Zhiru Zhang. A scalable approach to exact resource-constrained scheduling based on a joint SDC and SAT formulation. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 137–146, 2018.

[15] Ian Finlayson. Space complexity. *Ian Finlayson's blog*, 2025.

[16] True quantified Boolean formula. *Wikipedia*.

[17] Valentin Mayer-Eichberger and Abdallah Saffidine. Positional games and QBF: the corrective encoding. In *Theory and Applications of Satisfiability Testing–SAT 2020: 23rd International Conference, Alghero, Italy, July 3–10, 2020, Proceedings 23*, pages 447–463. Springer, 2020.

[18] Irfansha Shaik and Jaco van de Pol. Classical planning as QBF without grounding. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, pages 329–337, 2022.

[19] Shuyuan Zhang, Abdulrahman Mahmoud, Sharad Malik, and Sanjai Narain. Verification and synthesis of firewalls using SAT and QBF. In *2012 20th IEEE International Conference on Network Protocols (ICNP)*, pages 1–6. IEEE, 2012.

[20] Marco Benedetti and Hratch Mangassarian. QBF-based formal verification: Experience and perspectives. *Journal on Satisfiability, Boolean Modelling and Computation*, 5(1-4):133–191, 2009.

[21] Autenticación de múltiples factores. *Wikipedia*.

[22] Disyunctive Normal Form. *Wikipedia*.

[23] Conjunctive Normal Form. *Wikipedia*.

[24] Varisat Developers. *Varisat Manual*. Varisat Project, 2025. Accessed: July 10th, 2025.

[25] Hansjörg Schurr. README.md of QBFToys: A Collection of QBF Instances and Tools. `https://github.com/hansjoergschurr/QBFToys/blob/master/README.md`, 2025. Accessed: July 10, 2025.

[26] Charles Jordan, Will Klieber, and Martina Seidl. Non-CNF QBF Solving with QCIR. In *AAAI Workshop: Beyond NP*, volume 16, page 05, 2016.

[27] Leander Tentrup. QAIGER: AIGER format for Quantified Boolean Formulas. `https://github.com/ltentrup/QAIGER`, 2025. Repositorio de GitHub, consultado el 24 de julio de 2025.

[28] Luca Pulina and Martina Seidl. The 2016 and 2017 QBF solvers evaluations (QBFEVAL'16 and QBFEVAL'17). *Artificial Intelligence*, 274:224–248, 2019.

[29] Valeriy Balabanov and Jie-Hong R Jiang. Unified QBF certification and its applications. *Formal Methods in System Design*, 41(1):45–65, 2012.

[30] Florian Lonsing. An Overview of QBF Reasoning Techniques. 2016.

[31] Benjamin Böhm, Tomáš Peitl, and Olaf Beyersdorff. QCDCL with cube learning or pure literal elimination–What is best? *Artificial Intelligence*, 336:104194, 2024.

[32] Conflict-driven clause learning. *Wikipedia*.

[33] Florian Lonsing and Uwe Egly. DepQBF 6.0: A search-based QBF solver beyond traditional QCDCL. In *International Conference on Automated Deduction*, pages 371–384. Springer, 2017.

[34] Markus N Rabe and Leander Tentrup. CAQE: a certifying QBF solver. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pages 136–143. IEEE, 2015.

[35] Tomáš Peitl, Friedrich Slivovsky, and Stefan Szeider. Dependency learning for QBF. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491, pages 298–313, 2017.

[36] Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. Solving QBF with counterexample guided refinement. *Artificial Intelligence*, 234:1–25, 2016.

[37] William Klieber. GhostQ: System Description. *Journal on Satisfiability, Boolean Modelling and Computation*, 11(1):65–72, 2019.

[38] Jesko Hecking-Harbusch and Leander Tentrup. Solving QBF by abstraction. *arXiv preprint arXiv:1604.06752*, 2016.

[39] Mikoláš Janota. QFUN: towards machine learning in QBF. *arXiv preprint arXiv:1710.02198*, 2017.

[40] Julen Beristain. QBF Solver based on Inductive Normal Form – Código del Trabajo de Fin de Grado. `https://github.com/JulenBeristain/qbf-inf-solver`, 2025.

[41] Grafo acíclico dirigido. *Wikipedia*.

[42] Python Software Foundation. GlobalInterpreterLock. `https://wiki.python.org/moin/GlobalInterpreterLock`. Consultado el 24/08/2025.

[43] Thread (computing). *Wikipedia*.

[44] Multi-core processor. *Wikipedia*.

[45] Process (computing). *Wikipedia*.

[46] Memoización. *Wikipedia*.

[47] Memory management. *Wikipedia*.

[48] Garbage Collection in Python. *GeeksForGeeks*, 2025. Consultado el 24 de agosto de 2025.

[49] DPLL algorithm. *Wikipedia*.

[50] Unit propagation. *Wikipedia*.

[51] Ralf Wimmer and Ming-Yi Hu. Using Unit Propagation with Universal Reduction in DQBF Preprocessing. *arXiv preprint arXiv:2303.14446*, 2023.

[52] Marijn Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for CNF formulas. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 357–371. Springer, 2010.

[53] C (lenguaje de programación). *Wikipedia*.

[54] Cola (informática). *Wikipedia*.