E.T.S. INGENIERÍA INFORMÁTICA

UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

UNIVERSIDAD DE MÁLAGA

Graduada en Ingeniería Informática

Herramienta para la visualización de reducciones entre el problema SAT y otros problemas NPC

◊

Tool for visualization of reductions between the SAT problem and other NPC problems

Realizado por

Paula del Carmen Guzmán Enríquez

Tutorizado por

Rafael Morales Bueno

José del Campo Ávila

Departamento

Lenguajes y Ciencias de la Computación

MÁLAGA, JUNIO 2025

UNIVERSIDAD DE MÁLAGA

Resumen

El problema de la satisfacibilidad boolena (SAT) fue el primer problema identificado como perteneciente a la clase de complejidad NP-completo. Este problema, aunque no inicialmente ligado a la informática, guarda una estrecha relación con ella gracias a la teoría de la complejidad computacional.

Se tiene como propósito crear un material interactivo que facilite la comprensión de las reducibilidades entre problemas NP-completos. Se consigue mediante la demostración de cómo a partir de SAT, es posible reducir polinómicamente otros problemas NP-completos relacionados con la informática. Para ello, se desarrolla una aplicación web que describe el proceso de dichas reducciones. Este contempla tres etapas principales:

1. Transformación de una fórmula booleana dada por el usuario a una 3FNC (forma normal conjuntiva de 3 literales por cláusula). Se usan mapas de Karnaugh para una simplificación más eficiente de la fórmula.
2. Reducción del problema 3SAT (que expresa fórmulas booleanas 3FNC) a otro problema NP-completo que dispone de una representación visual (como CLIQUE o HAMPATH).
3. Identificación de posibles soluciones para cada problema individual. Se da la posibilidad al usuario de elegir una solución a mostrar entre todas las posibles. La solución elegida se representa visualmente.

Las reducciones se detallan para 4 problemas NP-completos: CLIQUE, VERTEX-COVER, HAMPATH y SUBSET-SUM.

Asimismo, cada uno de los problemas se complementa con una explicación detallada de las reducciones realizadas, con el fin de facilitar la comprensión por parte del usuario.

Palabras clave: Complejidad computacional, NP-completitud, problema SAT, reducibilidad entre problemas, representación visual

Abstract

The Boolean satisfiability problem (SAT) was the first problem identified as belonging to the class of NP-complete complexity. Although this problem was not initially linked to computer science, it has a close relationship with it, thanks to the theory of computational complexity.

The goal is to create an interactive material that facilitates the understanding of reducibilities between NP-complete problems. This is achieved by demonstrating how, starting from SAT, it is possible to polynomially reduce other NP-complete problems related to computer science. To this end, a web application is developed that performs this reduction. The process involves three main stages:

1. Transformation of a Boolean formula provided by the user into a 3CNF (conjunctive normal form with 3 literals per clause). Karnaugh maps are used for more efficient simplification of the formula.
2. Reduction of the 3SAT problem (expressing Boolean 3FNC formulas) to another NP-complete problem with a visual representation (such as CLIQUE or HAMPATH).
3. Identification of possible solutions for each individual problem. The user has the option to choose one solution to display among all possible ones. The chosen solution is visually represented.

This process is carried out with 4 NP-complete problems: Clique, Vertex-Cover, HAMPATH, and SUBSET-SUM.

In addition, each of the problems is complemented with a detailed explanation of the reductions performed, in order to facilitate the user's understanding.

Keywords: Computational complexity, NP-completeness, SAT problem, reducibility between problems, visual representation

Índice general

CAPÍTULO

Introducción 1

1.1 Motivación

La teoría de la computación comprende el estudio de qué problemas pueden computarse y cuáles no, con qué rapidez, cuánta memoria se requiere y en qué tipo de modelo computacional [1, p. xi]. La asignatura *Algoritmia y Complejidad* del tercer curso de la *Mención Computación* estudia, entre otros, una de las ramas primordiales de la teoría de la computación: la teoría de la complejidad computacional. Este campo se centra en clasificar los problemas computacionales según los recursos necesarios para resolverlos, como el tiempo, la memoria u otros que sean requeridos [1, p. 275]. Los dos primeros, tiempo y espacio, son de los más importantes en este análisis, tomando especial relevancia el tiempo [2, p. 402]. Igualmente, esta materia aborda la demostración de límites superiores e inferiores de la complejidad de los problemas [2, p. 401].

La clasificación de los problemas según su complejidad es fundamental porque permite evaluar la eficiencia de los algoritmos que los resuelven y determinar qué tan viable es su implementación en la práctica. La complejidad influye directamente en la eficiencia del algoritmo, y esta, a su vez, está estrechamente relacionada con el grado de precisión requerido en las soluciones. Como regla general, mientras más precisa sea la solución que se busca, mayor será el tiempo que demande el algoritmo. Por el contrario, si se desea reducir el tiempo de ejecución, a menudo es necesario sacrificar parte de esa precisión (se realiza una aproximación). Por ello, comprender la complejidad de un problema es esencial para encontrar un equilibrio adecuado entre tiempo y precisión, según las necesidades específicas de cada caso.

La página web desarrollada se enfoca justo en esta rama tan necesaria, aunque con frecuencia relegada, ya que suele resultar especialmente desafiante para los estudiantes debido a su naturaleza altamente abstracta, formal y a menudo alejada de aplicaciones inmediatas.

Con el fin de hacer más accesible este conocimiento, la aplicación propone un enfoque pedagógico que combina representaciones visuales, explicaciones detalladas paso a paso y una

interfaz interactiva e intuitiva. Todo ello está diseñado para acompañar al usuario a lo largo del proceso de reducción entre problemas NP-completos, facilitando no solo la comprensión teórica, sino también el aprendizaje activo a través de la exploración y la experimentación directa con ejemplos concretos.

1.2 Objetivos

El principal objetivo de este Trabajo de Fin de Grado (TFG) es acercar al estudiante a la teoría de la complejidad computacional. Con este fin, se ha realizado la reducción de 3SAT a cuatro problemas diferentes: CLIQUE, VERTEX-COVER, HAMPATH y SUBSET-SUM. La explicación de cada una de estas reducciones está organizada en una serie de pasos, los más relevantes, para facilitar el aprendizaje del usuario. Son los siguientes:

1. **Introducción de fórmula booleana y posterior transformación a 3FNC.** Se permite al usuario escribir cualquier fórmula usando los operadores de conjunción (and), disyunción (or) y negación (not). A partir de esta entrada, mediante el uso de mapas de Karnaugh, se obtiene otra fórmula expresada en forma normal conjuntiva (FNC), con el propósito de facilitar la posterior conversión a FNC de 3 literales (3FNC). Es importante señalar que la búsqueda de grupos óptimos en los mapas de Karnaugh constituye un problema NP en sí mismo, por lo que se ha implementado un algoritmo voraz para abordar este problema de manera más eficiente. Cabe destacar que al usar los mapas de Karnaugh, no se está buscando necesariamente una fórmula mínima, sino una expresión equivalente que, en lo posible, tenga menos cláusulas que la original. Este aspecto será explicado con más profundidad a lo largo de la memoria.

2. **Visualización del problema mediante la construcción de su gadget.** A partir de la fórmula 3FNC obtenida, se construye un componente formado por subestructuras o elementos auxiliares, que comúnmente se denomina *gadget*. Este componente suele tener una representación gráfica, que permite observar cómo se traduce la lógica booleana. Para generar los grafos asociados a la reducción de CLIQUE, VERTEX-COVER y HAMPATH se emplea la biblioteca D3.js, que ofrece visualizaciones dinámicas e interactivas. Para el caso de SUBSET-SUM, se utiliza una tabla.

3. **Identificación y visualización de las soluciones**. Se solicita al usuario que seleccione los valores a asignar a cada una de las variables booleanas. A partir de esta asignación, se genera y muestra el *gadget*, marcando la solución que se corresponde con los valores asignados. Si la evaluación de la fórmula resulta verdadera, la representación gráfica mostrará una solución válida; en caso contrario, el *gadget* no contendrá ninguna solución.

Todos estos pasos se acompañan de explicaciones específicas para cada uno de los problemas considerados. Asimismo, se incluye una breve introducción que describe en qué consiste cada uno de ellos, con el objetivo de facilitar la comprensión del proceso completo de reducción.

1.3 Metodología de trabajo empleada

La metodología de desarrollo iterativo e incremental (IID) es un enfoque evolutivo para construir software que evita la rigidez del modelo en cascada tradicional. Consiste en dividir el proyecto en pequeñas partes funcionales y ejecutables llamadas *slices*, las cuales se desarrollan en ciclos breves que incluyen análisis, diseño, implementación, prueba y retroalimentación. Cada iteración busca entregar un producto parcial que funcione según los requisitos definidos, permitiendo a los desarrolladores aprender de la experiencia real del sistema en uso y adaptarse a cambios continuos [3]. Históricamente, IID se ha aplicado con éxito desde los años 50 en proyectos complejos como el software del transbordador espacial, y ha sido promovido por referentes como Tom Gilb, Barry Boehm y Frederick Brooks. Su valor reside en ofrecer resultados tempranos y medibles, facilitar la detección de errores de forma temprana y permitir decisiones de gestión basadas en datos empíricos acumulados durante el proceso [4].

Durante el desarrollo de este proyecto, se ha seguido esta metodología para construir una single-page application (SPA). Esta estrategia permitió dividir el sistema en partes funcionales pequeñas que fueron diseñadas, implementadas y validadas de forma individual en ciclos breves. Gracias a este enfoque, fue posible integrar mejoras constantes basadas en la retroalimentación obtenida y ofrecer entregables funcionales en cada etapa del proceso.

Etapa 1. Creación del esqueleto de la aplicación. Se inició con una versión mínima del sistema, una single web application con navegación básica entre las secciones de inicio, CLIQUE, VERTEX-COVER, HAMPATH y SUBSET-SUM. Además, se incluye una breve presentación en el inicio y se implementa el modo oscuro y claro.

Etapa 2. Introducción de fórmulas booleanas y validación. Se permite al usuario introducir fórmulas booleanas utilizando los operadores *and*, *or* y *not*. Se implementa una función que verifica si es una fórmula bien formada. En caso contrario, se muestra un mensaje de error indicando que la fórmula introducida no es válida.

Etapa 3. Conversión a 3FNC con fórmulas de 4 variables o menos. Primeramente, se crea una función que internaliza las negaciones para así simplificar las fórmulas. Posteriormente, se realiza el mapa de Karnaugh para obtener la FNC, que luego se transforma en su equivalente 3FNC.

Etapa 4. Mapas de Karnaugh para más de 4 variables. Para fórmulas con más de 4 variables, se desarrollan funciones que comparan agrupaciones entre diferentes mapas. Se emplea un algoritmo voraz para seleccionar, en cada iteración, la agrupación más óptima.

Etapa 5. Visualización de grafos y tablas. Se crean visualizaciones interactivas de los problemas CLIQUE, VERTEX-COVER y HAMPATH utilizando la librería D3.js. Cada grafo se adapta en estructura y tamaño según las necesidades. Para SUBSET-SUM, se implementa un modelo de tabla específico.

Etapa 6. Tratamiento de tautologías y contradicciones. Si la fórmula insertada resulta ser una tautología o una contradicción, se devuelve el valor Verdadero o Falso, respectivamente. En estos casos, se genera un modelo 3FNC que incluye todas las variables proporcionadas por el usuario, junto con explicaciones adicionales.

Etapa 7. Asignación de valores a variables. Se incorpora una funcionalidad que permite al usuario asignar visualmente valores a las variables, mediante mapas. Si existen más de 4 tablas, se brinda la opción de introducir los valores uno a uno o desplegar tablas adicionales.

Etapa 8. Construcción de las estructuras a partir de la asignación. A partir de los valores asignados por el usuario, se generan los grafos correspondientes. En el caso de Subset-Sum, se crea una tabla con los valores aportados.

Etapa 9. Explicaciones. En cada vista se añade un texto explicativo que describe el problema abordado y cómo se realiza la reducción correspondiente.

1.4 Estructura de la memoria

La estructura seguida en la memoria es la siguiente:

Capítulo 1. Se presenta una introducción general, incluyendo la motivación que dio origen al proyecto, los objetivos que se persiguen, la metodología de trabajo empleada y una descripción de cómo se organiza el contenido del documento.

Capítulo 2. Describe las tecnologías empleadas en el desarrollo de la aplicación, incluyendo herramientas como React, JavaScript, HTML y Docker.

Capítulo 3. Ofrece una visión general de la aplicación desarrollada. Se explica cómo está estructurada la página web, incluyendo el punto de entrada, la navegación y las distintas vistas disponibles. Se detalla el apartado visual e interactivo del proyecto.

Capítulo 4. Se profundiza en el proceso de conversión de una fórmula booleana arbitraria a su Forma Normal Conjuntiva (FNC), y posteriormente a 3FNC. Se detallan los algoritmos y procedimientos empleados (mapas de Karnaugh).

Capítulo 5. Recoge las reducciones de fórmulas en 3FNC a distintos problemas clásicos NP-completos. Concretamente, se presentarán las reducciones del problema 3SAT a CLIQUE, VERTEX-COVER, HAMPATH y SUBSET-SUM. Se explica el razonamiento teórico detrás de cada reducción, así como su implementación visual.

Capítulo 6. Expone las conclusiones obtenidas a partir del trabajo realizado y presenta diversas líneas de mejora y ampliación del proyecto.

CAPÍTULO

Tecnologías usadas 2

2.1 React y JSX

React es una biblioteca de JavaScript desarrollada por Meta (anteriormente Facebook) que se utiliza para construir interfaces de usuario de manera eficiente y modular. Su característica principal es el uso de componentes reutilizables, que permiten dividir una aplicación en piezas independientes y manejables [5, pp. 1–3]. En este proyecto, React se ha utilizado junto con JSX (JavaScript XML), una extensión de sintaxis que permite escribir marcado similar a HTML dentro de un archivo JavaScript. JSX facilita el desarrollo de código, ya que combina tanto la lógica de renderizado como la estructura del marcado dentro de un mismo componente [5, p. 71].

Dentro de la aplicación web desarrollada, React ha sido empleado principalmente para la creación de componentes reutilizables que facilitan la consistencia y el mantenimiento del código. Se han desarrollado componentes específicos para la creación de botones con diferentes funciones, la visualización dinámica de los grafos generados para las reducciones, así como para la representación de tablas. Esta organización por componentes ha permitido mantener una interfaz coherente, reducir la duplicación de código y mejorar la escalabilidad del proyecto.

2.2 JavaScript y Node.js

JavaScript es un lenguaje de programación interpretado, ampliamente utilizado en el desarrollo web para dotar de interactividad y dinamismo a las páginas. Es un lenguaje versátil, orientado a objetos y basado en eventos, que permite manipular el contenido del navegador, responder a acciones del usuario y gestionar la lógica del lado del cliente. Junto con HTML y CSS, forma parte del núcleo de las tecnologías web, y es compatible con múltiples bibliotecas y frameworks, como React [6, p. 1–2].

En este proyecto también se ha utilizado Node.js, un entorno de ejecución para JavaScript del lado del servidor, creado por Ryan Dahl en 2009. Dahl, inicialmente interesado en la programación web, encontró que las tecnologías existentes no podían manejar eficientemente las solicitudes concurrentes. Después de explorar diversas soluciones, como Ruby y C, descubrió que JavaScript era ideal para este propósito debido a su naturaleza no bloqueante, su amplia adopción y la flexibilidad de su arquitectura. Node.js utiliza el motor V8 de Google para ejecutar el código JavaScript, optimizando la ejecución con compilación Just-In-Time (JIT) y mejorando el rendimiento general. Además, el proyecto es de código abierto y se gestiona en GitHub, con colaboradores que mejoran continuamente su desarrollo [7].

En el contexto de esta aplicación web, JavaScript ha sido esencial para implementar toda la lógica que permite transformar una fórmula booleana introducida por el usuario en su correspondiente forma 3FNC. Una parte central de esta lógica ha sido el desarrollo de un algoritmo capaz de generar y procesar mapas de Karnaugh, herramienta utilizada para simplificar expresiones booleanas y facilitar su conversión a la forma normal conjuntiva. Esta implementación, realizada completamente en JavaScript a través de la gestión de estructuras de datos con el objetivo de conseguir agrupaciones válidas dentro del mapa, constituye una de las piezas clave del funcionamiento interno de la aplicación.

2.3 HTML y CSS

HTML (HyperText Markup Language) es un lenguaje de marcado utilizado para crear documentos web, cuya función principal es describir la estructura del contenido mediante etiquetas que identifican elementos como encabezados, párrafos o listas. No se trata de un lenguaje de programación, sino de un sistema lógico para estructurar páginas. Existen varias versiones en uso, como HTML 4.01 y la más reciente HTML5, ambas con variantes más estrictas conocidas como XHTML. A menudo se complementa con tecnologías como CSS para el diseño visual y JavaScript para añadir funcionalidad dinámica, conformando así una base sólida para el desarrollo web moderno [8, p. 12].

CSS (Cascading Style Sheets) es un lenguaje de hojas de estilo utilizado para especificar la

presentación y el diseño de documentos escritos en lenguajes de marcado como HTML. Está diseñado para permitir la separación entre el contenido y su presentación visual, lo que facilita el control del diseño, colores, tipografías y disposición de los elementos. Además, CSS puede adaptarse a distintos contextos, como impresión o dispositivos móviles, y permite mantener una apariencia coherente en todo el sitio editando solo un archivo. Aunque HTML puede funcionar por sí solo, el uso de CSS es casi obligatorio para lograr un diseño profesional [8, p. 12].

HTML y CSS se han utilizado para definir y estructurar partes de la interfaz del usuario. HTML ha servido como base para organizar los distintos bloques de contenido, mientras que CSS ha dado estilo a cada uno de estos componentes, asegurando una presentación clara, coherente y agradable. La combinación de ambas tecnologías ha permitido construir una interfaz intuitiva y accesible para el usuario.

2.4 Docker

Docker es una plataforma de software de código abierto que permite crear, probar y desplegar aplicaciones usando contenedores: unidades ligeras, portables y aisladas que incluyen todo lo necesario para ejecutar una aplicación, como código, dependencias y configuración. A diferencia de las máquinas virtuales, los contenedores no requieren un sistema operativo completo por cada instancia, sino que comparten el núcleo del sistema anfitrión, lo que los hace más eficientes en consumo de recursos y más rápidos de iniciar. Gracias a esto, Docker permite crear entornos consistentes y reproducibles entre diferentes sistemas, resolviendo problemas como los conflictos de dependencias y facilitando el flujo de trabajo en desarrollo y producción [9].

En este proyecto, Docker se ha utilizado para encapsular la página web dentro de un contenedor, lo que permite ejecutarla de forma aislada del entorno local del desarrollador. Esto asegura que la aplicación funcione de manera idéntica en cualquier sistema donde se despliegue, ya sea en pruebas locales o en producción. Además, esta encapsulación facilita la escalabilidad del proyecto, ya que permite añadir nuevas funcionalidades o servicios en el futuro sin comprometer la estabilidad ni la estructura del sistema actual.

CAPÍTULO

3 Especificación y análisis

3.1 Introducción

El presente capítulo describe los requisitos para el desarrollo de una aplicación web interactiva cuyo propósito es facilitar la comprensión de las reducibilidades entre problemas NP-completos, un tema fundamental en teoría de la complejidad computacional. A través de esta herramienta, se busca que los usuarios, potencialmente estudiantes, puedan visualizar y experimentar de forma intuitiva cómo, a partir del problema SAT, es posible reducir polinómicamente a otros problemas NP-completos, fortaleciendo así su entendimiento conceptual y práctico. En este proyecto, se abarcan tres etapas clave:

1. Transformación de una fórmula booleana a su Forma Normal Conjuntiva de tres literales por cláusula (3FNC), utilizando mapas de Karnaugh.
2. Reducción del problema 3SAT a otro problema NP-completo, con representación visual a través de grafos o tablas.
3. Visualización de posibles soluciones al problema resultante, permitiendo al usuario seleccionar y explorar una solución específica.

El desarrollo cubrirá las reducciones desde 3SAT hacia cuatro problemas: CLIQUE, VERTEX-COVER, HAMPATH y SUBSET-SUM, acompañadas de explicaciones detalladas de cada paso de la transformación.

♦ 3.1.1. Alcance del producto

Esta aplicación está alineada con objetivos académicos y formativos, siendo particularmente útil en cursos de algoritmia, complejidad computacional o informática teórica. Asimismo, al

tratarse de un recurso interactivo, se espera que sea una herramienta de consulta tanto para estudiantes como para docentes o entusiastas de la computación teórica.

♦ 3.1.2. Valor del producto

La misión principal de esta página es la educativa, ya que convierte un contenido típicamente abstracto, complejo y estático en una experiencia visual y manipulable. Algunos de los beneficios que proporciona al usuario son:

- Explorar ejemplos reales de reducciones a problemas NP-completos.
- Comprender la lógica detrás de cada transformación de manera gráfica.
- Acceder a explicaciones didácticas que fortalecen la intuición y el conocimiento.

Al ofrecer un enfoque interactivo y visual, esta herramienta busca complementar los recursos pedagógicos existentes en el área, contribuyendo a mejorar la comprensión de las reducibilidades entre problemas NP-completos.

♦ 3.1.3. Público objetivo

El público objetivo está compuesto principalmente por:

- Estudiantes universitarios de carreras relacionadas con ciencias de la computación, ingeniería informática o matemáticas aplicadas.
- Docentes y educadores que buscan material didáctico interactivo para acompañar sus clases.
- Personas interesadas en la teoría de la computación que desean revisar o visualizar estas reducciones.

Este público espera una interfaz clara, visualmente atractiva y con un enfoque pedagógico que priorice la comprensión conceptual.

3.1.4. Uso previsto

Se espera que los usuarios interactúen con la aplicación de la siguiente forma:

1. Escoger un problema NP-completo destino (por ejemplo, CLIQUE).
2. Introducir una fórmula booleana para transformarla automáticamente a 3FNC.
3. Visualizar cómo se realiza la reducción desde 3SAT al problema elegido.
4. Escoger y examinar una o varias soluciones posibles representadas visualmente, comprendiendo cómo la solución al problema destino (por ejemplo, CLIQUE) refleja una solución del problema original (SAT).
5. Consultar explicaciones adicionales para profundizar en el entendimiento de cada transformación.

Algunos posibles escenarios de uso podrían ser los siguientes:

1. Estudiante que desea practicar reducciones entre problemas para un examen.
2. Docente que proyecta la aplicación en clase para explicar el proceso.
3. Usuario autónomo que explora diversas reducciones por curiosidad o repaso.

3.2 Requisitos

♦ 3.2.1. Requisitos funcionales

RF-1 Ingreso y procesamiento de fórmulas booleanas.

RF-1.1 Ingreso de fórmula booleana. El sistema deberá permitir al usuario ingresar una fórmula booleana válida mediante una interfaz de texto.

RF-1.2 Validación de sintaxis. Si la fórmula ingresada no es válida, el sistema deberá mostrar un mensaje de error.

RF-1.3 Conversión a 3FNC. Al ingresar una fórmula válida, el sistema deberá transformarla automáticamente a su forma normal conjuntiva con 3 literales por cláusula (3FNC).

RF-2 Reducción de 3SAT a otros problemas NP-completos.

RF-2.1 Generación visual del gadget. Una vez realizada esta conversión, el sistema deberá generar el gadget correspondiente a cada problema, basado en la reducción desde 3SAT definida teóricamente.

RF-2.2 Explicación del funcionamiento del gadget. Junto a la representación visual, el sistema deberá proporcionar una explicación detallada sobre cómo se construye el gadget a partir de la fórmula en 3FNC y por qué esta construcción es válida como reducción desde 3SAT al problema elegido.

RF-2.3 Manejo de casos especiales: tautologías y contradicciones. En caso de que la fórmula ingresada sea una tautología o una contradicción, el sistema deberá generar una explicación adicional que aclare cómo se reflejan estos casos en la estructura del gadget y sus diferencias con una fórmula satisfacible.

RF-3 Exploración y visualización de posibles asignaciones.

RF-3.1 Selección de una asignación booleana. El sistema permitirá al usuario explorar y seleccionar manualmente una asignación de valores de verdad para las variables de la fórmula en 3FNC.

RF-3.2 Generación del gadget según la asignación elegida. Una vez seleccionada una asignación, el sistema deberá generar el gadget correspondiente a la susodicha, resaltando visualmente los elementos relevantes que reflejan cómo la fórmula se satisface o no.

RF-3.3 Relación entre la fórmula 3FNC y el gadget generado. La aplicación proporcionará una explicación clara de la relación entre las asignaciones de la 3FNC y la estructura resultante del gadget, ayudando al usuario a entender cómo las asignaciones sobre las variables afectan al problema NP-completo representado.

♦ 3.2.2. Requisitos de la interfaz

RF-4. Navegación entre vistas principales. El sistema deberá permitir al usuario navegar de forma fluida entre las distintas vistas principales de la aplicación, incluyendo la página de inicio y las secciones correspondientes a cada uno de los problemas: CLIQUE, VERTEX-COVER, HAMPATH y SUBSET-SUM.

♦ 3.2.3. Requisitos no funcionales

RNF-1. Compatibilidad web. La aplicación deberá funcionar correctamente en navegadores modernos (Chrome, Firefox, Edge) sin necesidad de instalación adicional.

RNF-2. Optimización de la transformación a 3FNC mediante simplificación. La transformación de la fórmula booleana original a su forma normal conjuntiva de tres literales por cláusula (3FNC) deberá, en la medida de lo posible, generar una versión más reducida o simplificada que la fórmula original, entendiéndose por "más reducida" una fórmula con menor cantidad de cláusulas o con una estructura más simplificada que conserve la equivalencia lógica. Esta optimización se logrará mediante el uso de mapas de Karnaugh durante el proceso de simplificación a FNC.

RNF-3. Representación visual e interactiva del gadget. El gadget generado a partir de la fórmula en 3FNC deberá representarse de forma visual e interactiva, permitiendo al usuario explorar sus componentes. Cumplirá con las siguientes características:

- Los componentes del grafo podrán ser arrastrados libremente por el usuario para reorganizar su disposición.
- La interfaz deberá permitir acercar y alejar la vista del grafo.
- La interacción deberá mantenerse fluida y adaptable, incluso en casos con una cantidad considerable de nodos y aristas.

RNF-4. Interfaz visual para la selección de asignaciones. La selección de asignaciones de valores de verdad para las variables deberá realizarse de forma visual en la medida de lo posible, utilizando representaciones gráficas como mapas de Karnaugh o tablas interactivas que faciliten la comprensión.

- Cuando el número de variables lo permita, el sistema deberá ofrecer mapas visuales que permitan al usuario seleccionar asignaciones de forma intuitiva.
- En casos donde la cantidad de variables sea demasiado elevada para una visualización clara, la aplicación deberá adaptarse mostrando una interfaz alternativa, permitiendo al usuario seleccionar manualmente el valor de cada variable mediante controles adecuados (por ejemplo, listas desplegables o interruptores).

RNF-5. Tiempo de respuesta aceptable. Las transformaciones y visualizaciones no deberán tardar más de 2 segundos en condiciones normales (fórmulas con menos de 11 variables).

RNF-6. Interfaz intuitiva. La navegación deberá ser sencilla y clara para usuarios sin conocimientos técnicos avanzados.

RNF-7. Cambio de modo claro/oscuro. El sistema ofrecerá una opción visible para alternar entre el modo claro y el modo oscuro de la interfaz.

RNF-8. Persistencia de la preferencia de modo visual. El sistema conservará la preferencia de visualización del usuario (modo claro u oscuro) durante toda la sesión activa. Además, esta preferencia se almacenará de manera local en el navegador del usuario, de modo que se aplique automáticamente en futuras visitas sin necesidad de reconfiguración manual.

3.3 Casos de uso

Título	Ingreso y validación de fórmula booleana
Descripción	El sistema deberá permitir al usuario ingresar una fórmula booleana válida mediante una interfaz de texto
Requisitos	RF-1.1, RF-1.2
Pre-condición	
Post-condición	El usuario ha introducido una fórmula bien formada
Escenario principal	
1. El usuario introduce una fórmula booleana bien formada en el campo de entrada. 2. El sistema verifica que la fórmula es correcta. 3. El sistema muestra el resto del contenido relacionado con la fórmula introducida.	
Escenario alternativo	
Fórmula inválida por sintaxis incorrecta 1. El usuario introduce una fórmula con sintaxis incorrecta. 2. El sistema comprueba la fórmula y detecta un error de sintaxis. 3. El sistema muestra un mensaje de error indicando que la fórmula es inválida y no realiza el resto de las operaciones. 4. El sistema permite al usuario corregir la fórmula y repetir el proceso.	

Título	Conversión de fórmula booleana a 3FNC
Descripción	Al ingresar una fórmula válida, el sistema deberá transformarla automáticamente a su forma normal conjuntiva con 3 literales por cláusula (3FNC)
Requisitos	RF-1.3, RNF-2, RNF-5
Pre-condición	El usuario ha introducido una fórmula bien formada
Post-condición	
Escenario principal	
1. El sistema realiza operaciones para internalizar las negaciones en la fórmula. 2. El sistema genera una forma normal conjuntiva (FNC) equivalente, aplicando técnicas de simplificación lógica (por ejemplo, mediante mapas de Karnaugh). 3. A partir de la FNC, el sistema convierte cada cláusula a una con exactamente 3 literales (3FNC). Se usan variables auxiliares si procede.	
Escenario alternativo	

Título	Generación visual e interactiva del gadget
Descripción	El sistema deberá generar el gadget correspondiente a cada problema, basado en la reducción desde 3SAT definida teóricamente
Requisitos	RF-2.1
Pre-condición	El sistema ha obtenido la fórmula 3FNC equivalente
Post-condición	
Escenario principal	
1. El sistema detecta que una fórmula booleana ha sido convertida con éxito a 3FNC. 2. El sistema construye el gadget correspondiente basado en la teoría de reducción desde 3SAT. 3. El sistema activa elementos interactivos en la visualización (por ejemplo: nodos seleccionables, explicaciones extras, relaciones resaltables).	
Escenario alternativo	

Título	Explicación del funcionamiento del gadget
Descripción	El sistema deberá proporcionar una explicación detallada sobre cómo se construye el gadget a partir de la fórmula en 3FNC y por qué esta construcción es válida como reducción desde 3SAT al problema correspondiente
Requisitos	RF-2.2
Pre-condición	El sistema ha obtenido la fórmula 3FNC equivalente
Post-condición	
Escenario principal	
1. El sistema genera una explicación detallada de la creación del gadget. Cada explicación es distinta para cada problema.	
Escenario alternativo	

Título	Manejo de tautologías y contradicciones
Descripción	En caso de que la fórmula ingresada sea una tautología o una contradicción, el sistema deberá generar una explicación adicional que aclare cómo se reflejan estos casos en la estructura del gadget y sus diferencias con una fórmula satisfacible
Requisitos	RF-2.3
Pre-condición	La fórmula introducida es una tautología o contradicción
Post-condición	
Escenario principal	
1. El sistema determina que la fórmula es una tautología o contradicción. 2. El sistema genera un modelo de 3FNC en el que se mantienen todas las variables introducidas por el usuario originalmente. 3. El sistema añade explicaciones adicionales sobre las tautologías y contradicciones.	
Escenario alternativo	

Título	Selección de una asignación booleana
Descripción	El sistema permitirá al usuario explorar y seleccionar manualmente una asignación de valores de verdad para las variables de la fórmula en 3FNC
Requisitos	RF-3.1, RNF-4
Pre-condición	El sistema ya ha generado la fórmula en 3FNC y su correspondiente gadget visual
Post-condición	
Escenario principal	
1. El sistema presenta una tabla interactiva con todas las asignaciones posibles. 2. El usuario selecciona una de las asignaciones. 3. El sistema registra la asignación y la muestra por pantalla.	
Escenario alternativo	
Cantidad elevada de variables 1. El sistema detecta que hay demasiadas variables para una visualización clara de tablas interactivas. Por tanto, muestra una interfaz alternativa más sencilla y manual. 2. El usuario selecciona los valores de las variables una a una. 3. El sistema registra la asignación y la muestra por pantalla.	

Título	Generación del gadget según la asignación elegida
Descripción	El sistema deberá generar el gadget correspondiente a la asignación, resaltando visualmente los elementos relevantes que reflejan cómo la fórmula se satisface o no
Requisitos	RF-3.2
Pre-condición	El usuario ha elegido una asignación
Post-condición	
Escenario principal	
1. El sistema genera el gadget para la asignación seleccionada. Se resaltan los elementos importantes.	
Escenario alternativo	

Título	Explicación de la relación entre la fórmula 3FNC y el gadget generado
Descripción	La aplicación proporcionará una explicación clara de la relación entre las asignaciones y la estructura resultante del gadget, ayudando al usuario a entender cómo las asignaciones sobre las variables afectan al problema NP-completo representado
Requisitos	RF-3.3
Pre-condición	
Post-condición	
Escenario principal	
1. El sistema presenta una explicación sobre la relación entre la fórmula 3FNC y el gadget.	
Escenario alternativo	

CAPÍTULO

4 Panorama general: Presentación de la página

4.1 Inicio del proyecto

La página web creada se basa en el modelo Single Page Application (SPA), o aplicación de página única, un tipo de aplicación web o sitio web que ofrece una experiencia de usuario más fluida y dinámica al evitar la recarga completa de la página durante la navegación. En lugar de utilizar el método tradicional de cargar nuevas páginas desde el servidor en cada interacción, una SPA actualiza de forma dinámica el contenido visible mediante la reescritura de la página actual con datos obtenidos del servidor [6, p. 497].

Este enfoque tiene como principal objetivo mejorar la velocidad de respuesta y proporcionar una experiencia más similar a la de una aplicación nativa. En una SPA, todo el código necesario (incluyendo HTML, JavaScript y CSS) se carga en una única solicitud inicial, o bien se obtiene de manera dinámica conforme el usuario interactúa con la aplicación. Como resultado, se elimina la necesidad de realizar recargas completas de página, lo cual optimiza el rendimiento y la usabilidad.

Este proyecto se apoya en React, que es uno de los frameworks de Javascript que han adoptado los principios de las SPA. Como mencionamos en la descripción de React, esta se basa en componentes, lo cual facilita construir aplicaciones de página única, ya que actualiza dichos componentes de forma dinámica sin necesidad de recargar la página completa. Esto no solo se traduce en una optimización del rendimiento, sino también en una experiencia más ágil y fluida para el usuario.

4.2 Punto de entrada

Para la puesta en marcha del proyecto, se han creado un total de 5 vistas: la página principal y 4 vistas más que se corresponden con cada una de las reducciones de 3SAT a CLIQUE, VERTEX-COVER, HAMPATH y SUBSET-SUM. Adicionalmente, si se intenta acceder a una dirección que no está en el servidor, esta se derivará a una página con el error 404. Naturalmente, esta pieza de código que sirve de punto de entrada a nuestra aplicación está realizada con componentes.

```
<BrowserRouter>
    <Navigator theme={theme} setTheme={setTheme}
        language={language} setLanguage={setLanguage}/>
    <Routes>
        <Route path="/" element={<HomePage theme={theme} language={
            language}/>}/>
        <Route path="/machineClique"
            element={<MachineClique theme={theme}/>}/>
        <Route path="/machineVertex"
            element={<MachineVertexCover theme={theme}/>}/>
        <Route path="/machineHamPath"
            element={<MachineHamPath theme={theme}/>}/>
        <Route path="/machineSubSetSum"
            element={<MachineSubSetSum theme={theme}/>}/>
        {/* If none of the above -> error 404*/}
        <Route path="*" element={<Error404 theme={theme}/>}/>
    </Routes>
</BrowserRouter>
```

Código 4.1: Código del punto de entrada

- Se incluye `BrowserRouter`, en el cual se envuelven todas las partes que necesitan enrutamiento basado en el historial de navegación.
- `Navigator` es un componente personalizado que contiene la barra de navegación. Se analiza con más detalle en la siguiente sección.
- Dentro de los contenedores `Routes` se especifican las diferentes rutas y componentes

que deben renderizar cada una. Se corresponden con cada una de las vistas de la página web, y examinaremos como están estructuradas en la sección Vistas.

Por si al lector le surge la duda, la variable `theme` es un estado creado para permitir que la página web cambie de modo claro a oscuro y viceversa. Su introducción responde no solo a una intención estética, sino también a mejorar la comodidad visual de los usuarios que navegan en entornos con poca luz y a reducir el consumo de batería de las pantallas.

4.3 Navegador

Para poder viajar entre las distintas vistas, se ha realizado un navegador. En este se incluyen las principales páginas: Inicio, Clique, Vertex Cover, HamPath y Subset Sum. Igualmente, se añade un icono que simboliza el cambio del modo claro a oscuro y viceversa, además de la opción de cambiar los textos a inglés. No obstante, por el momento, solo se ha traducido la página de inicio. El resto de vistas están disponibles únicamente en español.

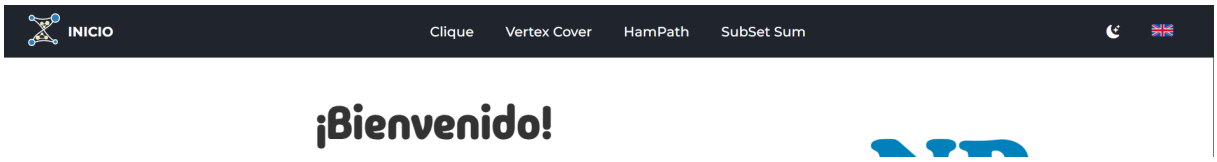


Imagen 4.1: Navegador en modo claro en español

Imagen 4.2: Navegador en modo claro en inglés

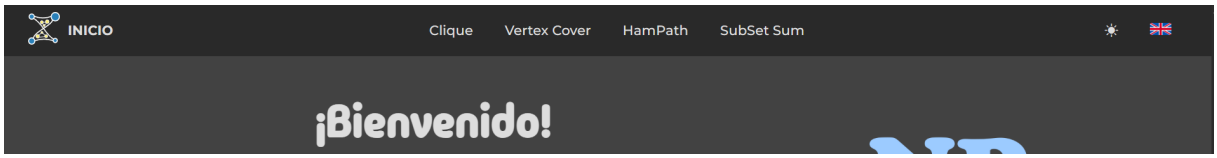


Imagen 4.3: Navegador en modo oscuro

No es necesario mencionar que este navegador cuenta con su propio CSS, haciendo que la interacción con el usuario se sienta más dinámica y reactiva ante las acciones.

4.4 Vistas

Antes de explicar el código que se ha desarrollado para poder realizar las reducciones, veamos una pincelada general de la estructura de la página, para comprender mejor cómo se organizan los elementos.

♦ 4.4.1. Página de inicio

Iniciamos el recorrido con la página de inicio. Esta vista es la primera que se muestra al entrar en el sitio web. En ella se incluye una explicación general sobre cómo se realiza la reducción. Se muestra un trozo de esta en la imagen 4.4.

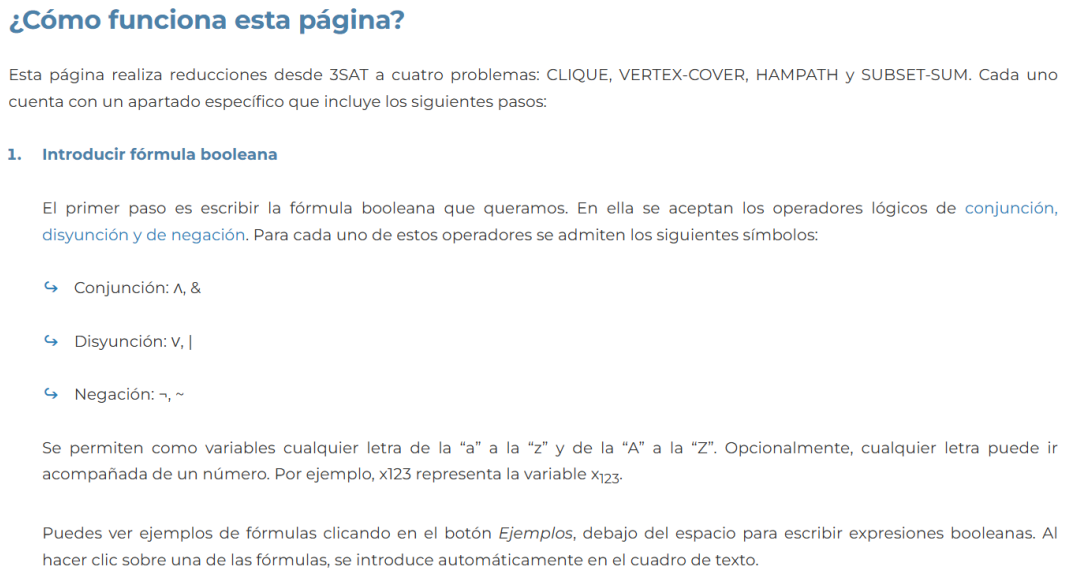


Imagen 4.4: Extracto de la explicación presentada en la página de inicio

Asimismo, como bienvenida al usuario, se ha incorporado un pequeño extra para hacer la experiencia más interesante. Es un sistema dinámico que va mostrando una serie de URLs a intervalos regulares, invitando al usuario a hacer clic en ellas para acceder a las páginas correspondientes.

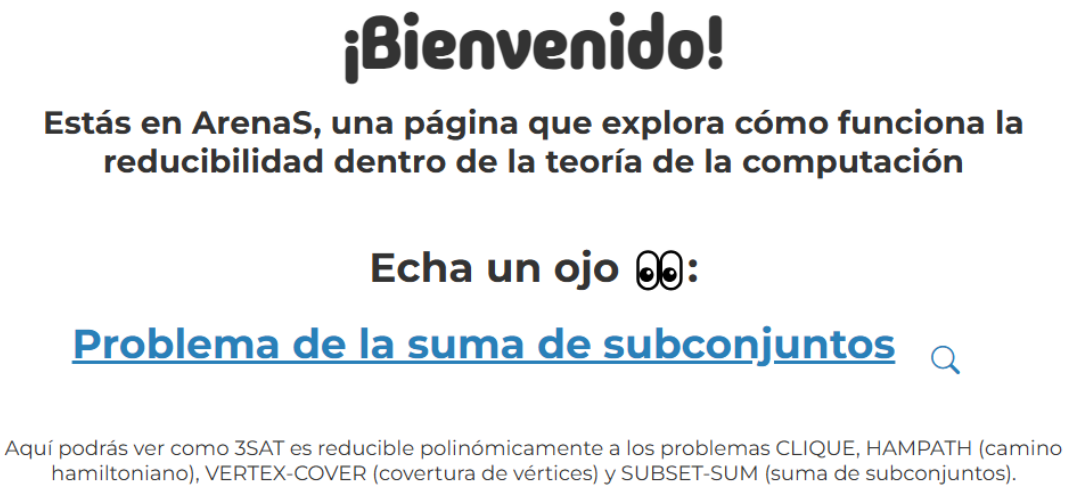


Imagen 4.5: Ejemplo de enlace que el usuario puede clicar

El código que las ejecuta es una función *javascript* que simula la escritura por teclado y que va añadiendo letra a letra el texto que se quiere mostrar. El desarrollo es relativamente sencillo, constando de 3 partes: primero, se simula la escritura del texto de manera progresiva; una vez que el texto se ha completado, este se convierte en un hipervínculo clicable que redirige a la página web correspondiente; finalmente, se simula el borrado del texto, eliminando el último carácter de la cadena en cada iteración.

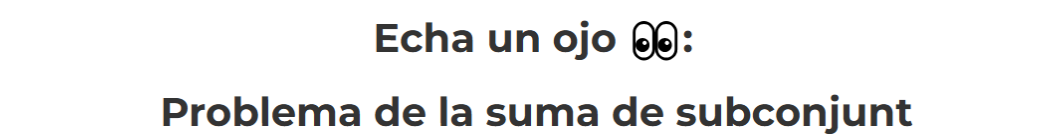


Imagen 4.6: Se realiza efecto de escritura y borrado. En estos casos, no se puede clicar sobre el texto

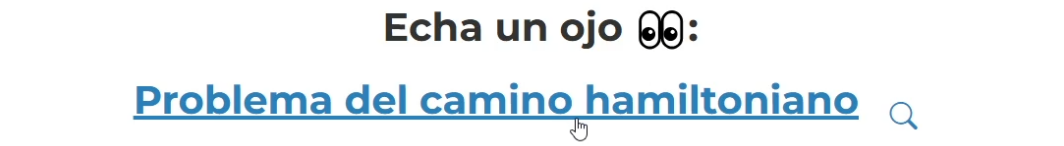


Imagen 4.7: Cuando el texto está al completo, se permite al usuario clicar sobre él. Al pulsar, se abre una nueva pestaña con información sobre el tema seleccionado

4.4.2. Vistas de las reducciones

Una vez terminada la explicación de la página inicial, sigamos con la parte más crucial de la página web: las vistas de las reducciones.

Estas vistas se asemejan unas a otras en estructura. Es por este motivo que las imágenes expuestas a continuación muestran la vista de CLIQUE únicamente, puesto que enseñar las demás sería redundante.

Las vistas de las reducciones siempre comienzan con una breve explicación del problema y por qué es considerado NP-completo. El objetivo de esta pequeña introducción es el de proporcionar contexto al usuario sobre el problema que se va a abordar. En algunos de ellos, se incluyen ejemplos para ilustrar qué casos pueden considerarse soluciones del problema y cuáles no.

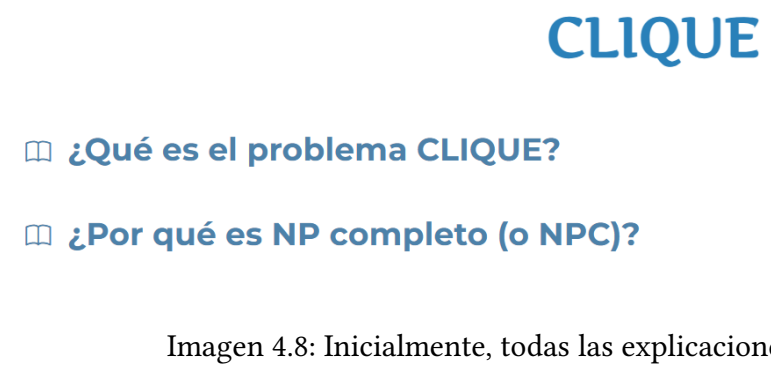


Imagen 4.8: Inicialmente, todas las explicaciones están plegadas

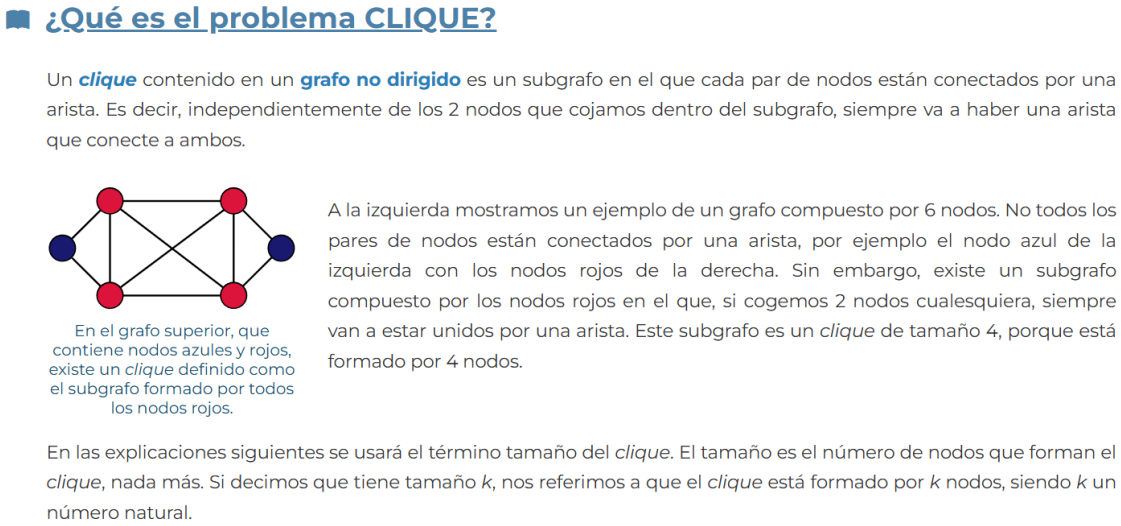


Imagen 4.9: Cuando clicamos sobre una de ellas, se despliega mostrando la explicación que contiene en su interior

Después de este breve preámbulo, se deja al usuario introducir cualquier fórmula booleana. Adicionalmente, si el usuario no desea pensar en una concreta, se ha creado un desplegable con algunos ejemplos que puede usar, clicando directamente en ellos. Todos estos se ven en la imagen 4.11.

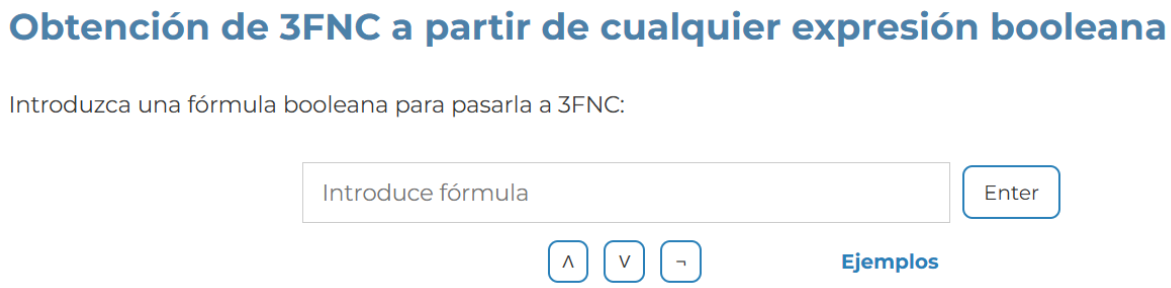
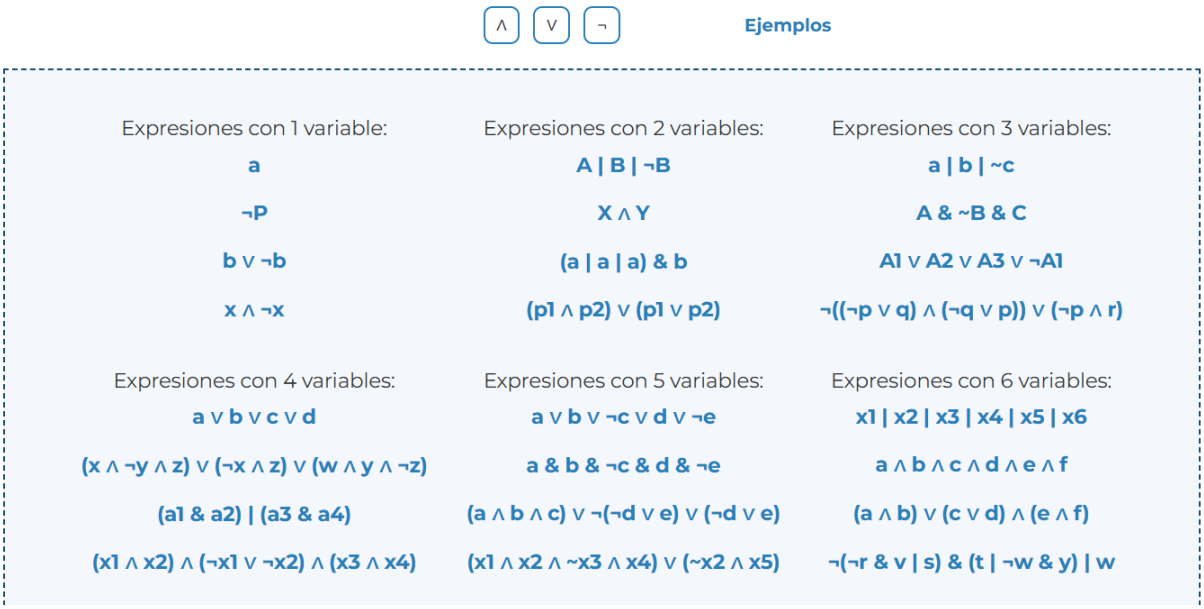


Imagen 4.10: Se puede introducir cualquier fórmula booleana que se desee



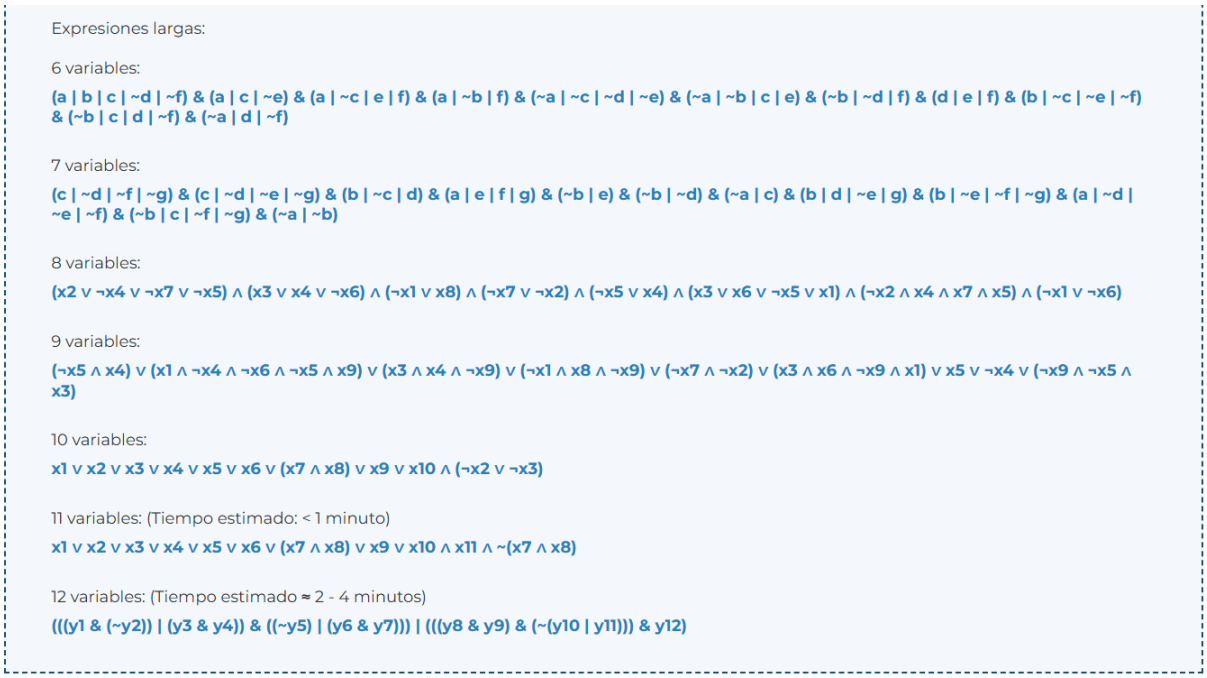
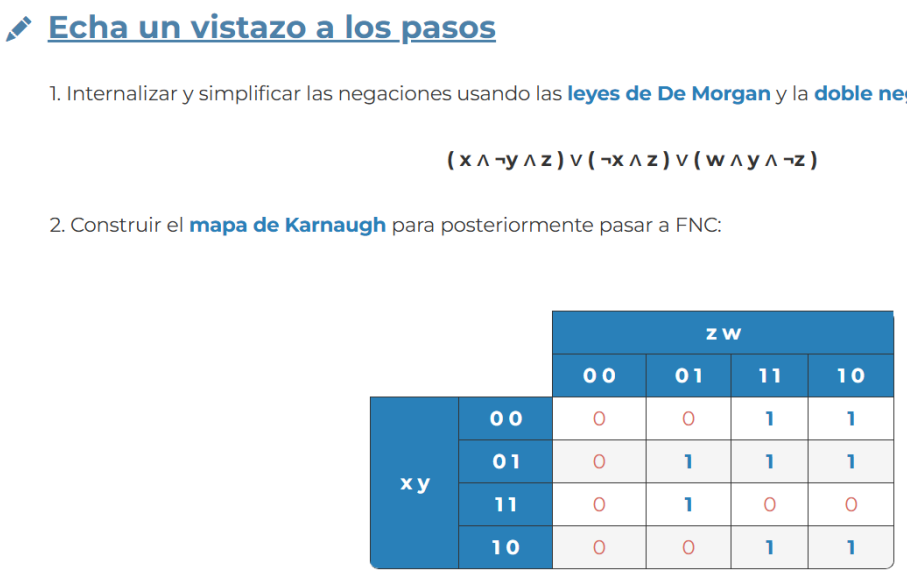


Imagen 4.11: Clicando sobre el botón *Ejemplos*, el usuario puede elegir entre todas estas fórmulas de ejemplo. Sus tamaños varían entre 1 y 12 variables

Una vez introducida, opcionalmente se pueden ver los pasos que se han realizado para transformar la fórmula booleana en 3FNC. En este proceso se muestran los mapas de Karnaugh resultantes y el posterior paso de forma normal conjuntiva a 3 variables por cláusula.



3. Obtener **FNC** a partir del mapa de Karnaugh **(Vídeo explicación en inglés)**:

xy \ zw	00	01	11	10
00	0	0	1	1
01	0	1	1	1
11	0	1	0	0
10	0	0	1	1

$$(y \vee z) \wedge (z \vee w) \wedge (\neg x \vee \neg y \vee \neg z)$$

4. Pasar a 3FNC **(Explicación en inglés)**:

$$(y \vee z \vee z) \wedge (z \vee w \vee w) \wedge (\neg x \vee \neg y \vee \neg z)$$

Imagen 4.12: Opcionalmente se pueden ver los pasos. Se ilustra un ejemplo con la fórmula $(x \wedge \neg y \wedge z) \vee (\neg x \wedge z) \vee (w \wedge y \wedge \neg z)$

En el caso de tratarse de una fórmula insatisfacible o tautología, se muestra otra explicación opcional en la que se detalla por qué estas fórmulas también tienen una representación gráfica.

La expresión da como resultado **FALSO**, es decir, es una fórmula **insatisfacible** o **contradicción**:

$$(x \vee x \vee x) \wedge (\neg x \vee \neg x \vee \neg x)$$

Echa un vistazo a los pasos

El resultado es una contradicción. ¡No lo entiendo!

¡Oh, no! Tu expresión da como resultado Falso para cualquier asignación que reciban las variables. ¿Qué hacemos ahora? ¿Se puede hacer la reducción de 3SAT a CLIQUE?

La respuesta es **SÍ**. Cada **Forma Normal Conjuntiva (FNC)** de 3 literales por cláusula puede ser:

⇝ **Insatisfacible o contradicción**: la fórmula es falsa en todas las asignaciones.

⇝ **Válida o tautología**: la fórmula es verdadera en todas las asignaciones.

⇝ **Satisfacible**: la fórmula es verdadera para algunas asignaciones de valores lógicos. Cuando la FNC es satisfacible, pertenece al conjunto a 3SAT.

Imagen 4.13: Si la fórmula es insatisfacible o tautología, es posible desplegar una explicación opcional

Seguidamente, se muestra el *gadget* y una descripción de cómo se realiza. Para los problemas cuyo *gadget* es un grafo, se ha usado la herramienta D3.js, la cual será expuesta con mayor detalle en un capítulo posterior. En el caso de las fórmulas insatisfacibles o tautologías, este grafo viene acompañado de una explicación en la que se recalca lo expuesto en la imagen 4.13. Por añadidura, en el caso de las fórmulas insatisfacibles, acompañando a la explicación hay un botón que, al pulsarlo, enseña las aristas que faltan para que fuera satisfacible, o lo que es lo mismo, para que existiera un *clique* de tamaño igual al número de cláusulas *k*.

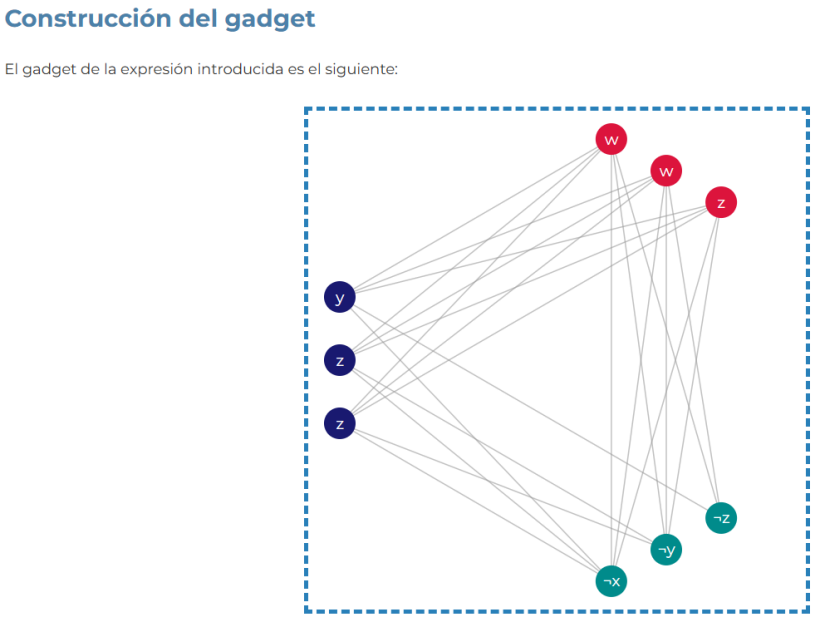


Imagen 4.14: Se crea el *gadget* de la fórmula introducida

Construcción del gadget

Cuando una fórmula es insatisfacible, el modelo $(x_1 \vee x_2 \vee \ldots \vee x_n) \wedge \neg x_1 \wedge \neg x_2 \wedge \ldots \wedge \neg x_n$, presenta las **fórmulas atómicas** y sus **negaciones** en cláusulas distintas. Todos los literales positivos están agrupados en una o varias cláusulas. Las negaciones de cada variable ocupan una cláusula completa.

En el grafo, esto se representa con tripletes que contienen solo los literales positivos de las variables y otros tripletes que contienen únicamente las negaciones de una misma variable. A consecuencia de esto, siempre habrá como mínimo un triplete que no tendrá nodos seleccionados. Es decir, no existirán *cliques* del mismo tamaño que el número de cláusulas.

Cliquea en la palanca para ver cuáles son las aristas que faltan. Al clicarla, verás que hay aristas de color negro discontinuas. Estas representan los enlaces que faltan para que el grafo pudiera formar un ***clique* de tamaño igual al número de cláusulas**.

El gadget de la expresión introducida es el siguiente:

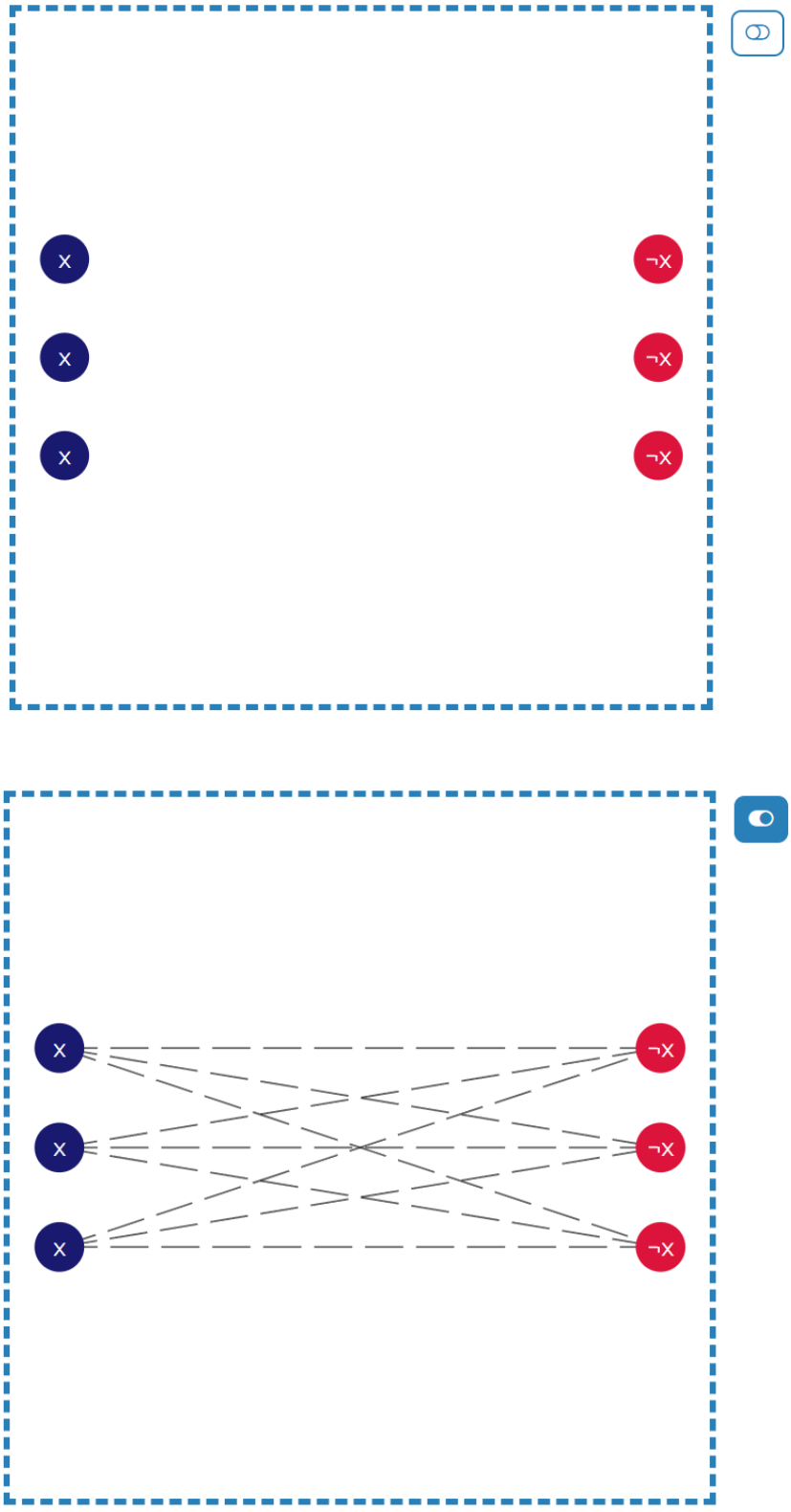
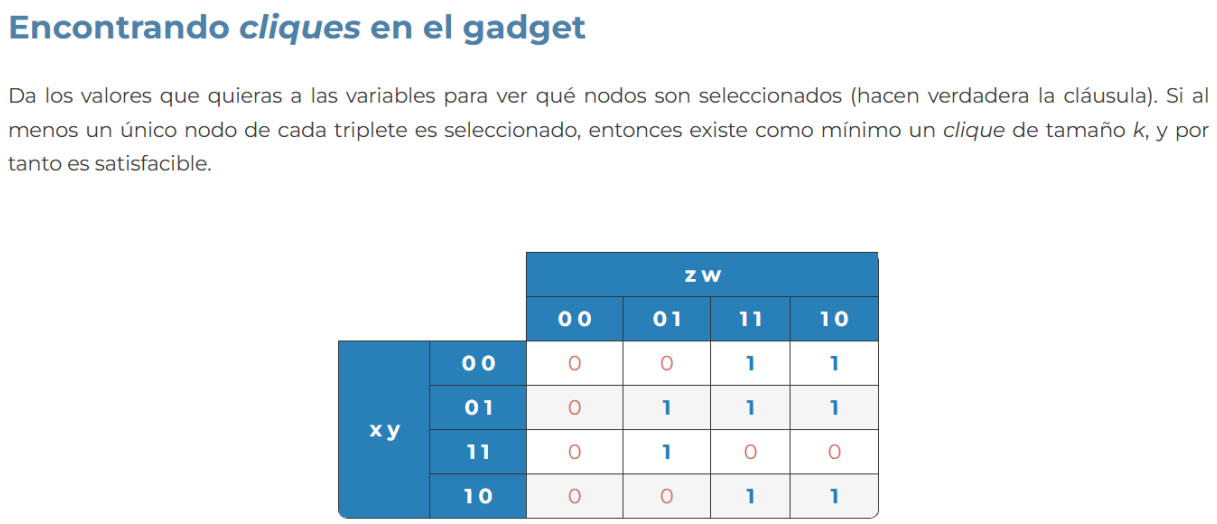


Imagen 4.15: Para las fórmulas insatisfacibles, se da una pequeña explicación complementaria del grafo y se muestran las aristas que debería tener para que fuera satisfacible

Por último, se muestra la opción de asignar valores a las variables para así ver cómo se comporta el *gadget* para esa asignación específica. Para dar los valores, solo tenemos que escoger

una casilla del mapa. En los casos donde la fórmula 3FNC resultante es muy larga (más de 4 tablas), se da la opción al usuario de asignar uno por uno los valores, ya que realizar la búsqueda en más de cuatro tablas puede convertirse en una tarea ardua.

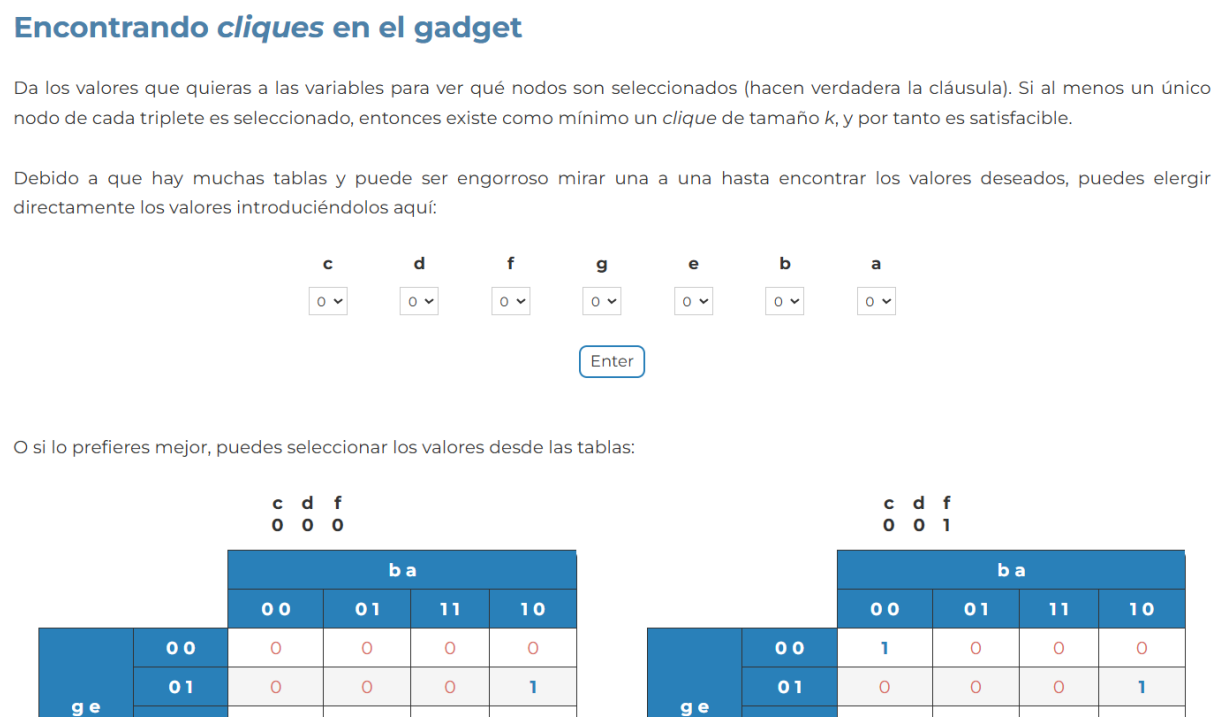


Encontrando *cliques* en el gadget

Da los valores que quieras a las variables para ver qué nodos son seleccionados (hacen verdadera la cláusula). Si al menos un único nodo de cada triplete es seleccionado, entonces existe como mínimo un *clique* de tamaño *k*, y por tanto es satisfacible.

		zw			
		00	01	11	10
xy	00	0	0	1	1
	01	0	1	1	1
	11	0	1	0	0
	10	0	0	1	1

Imagen 4.16: Asignación de valores mediante tablas



Encontrando *cliques* en el gadget

Da los valores que quieras a las variables para ver qué nodos son seleccionados (hacen verdadera la cláusula). Si al menos un único nodo de cada triplete es seleccionado, entonces existe como mínimo un *clique* de tamaño *k*, y por tanto es satisfacible.

Debido a que hay muchas tablas y puede ser engorroso mirar una a una hasta encontrar los valores deseados, puedes elergir directamente los valores introduciéndolos aquí:

c d f g e b a

0 0 0 0 0 0 0

Enter

O si lo prefieres mejor, puedes seleccionar los valores desde las tablas:

c d f
0 0 0

		ba			
		00	01	11	10
ge	00	0	0	0	0
	01	0	0	0	1

c d f
0 0 1

		ba			
		00	01	11	10
ge	00	1	0	0	0
	01	0	0	0	1

Imagen 4.17: Si hay más de 4 tablas, se deja a elección del usuario asignar valores mediante tablas o listas desplegables

Esta asignación viene acompañada de su correspondiente grafo o tabla, según el problema en que nos encontremos. En la imagen 4.18, los nodos coloreados son aquellos que pertenecen a algún *clique*. En cambio, se colorean de gris aquellos que no pertenecen a ninguno.

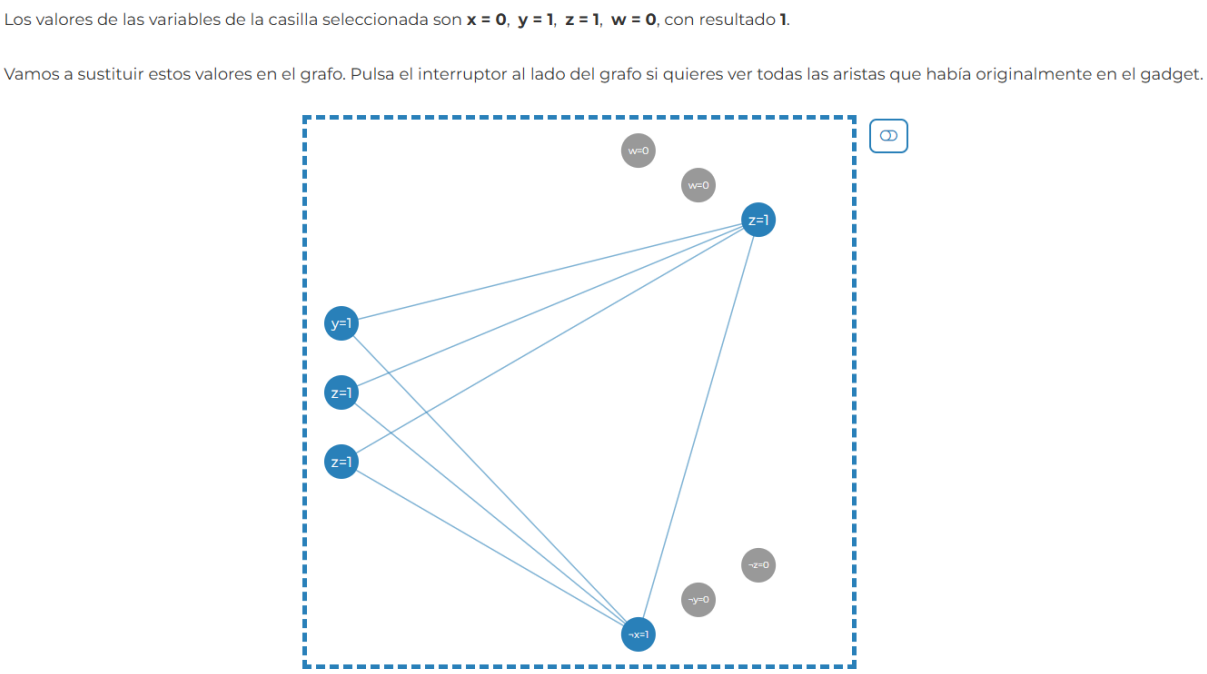


Imagen 4.18: Se muestran todos los *cliques* posibles

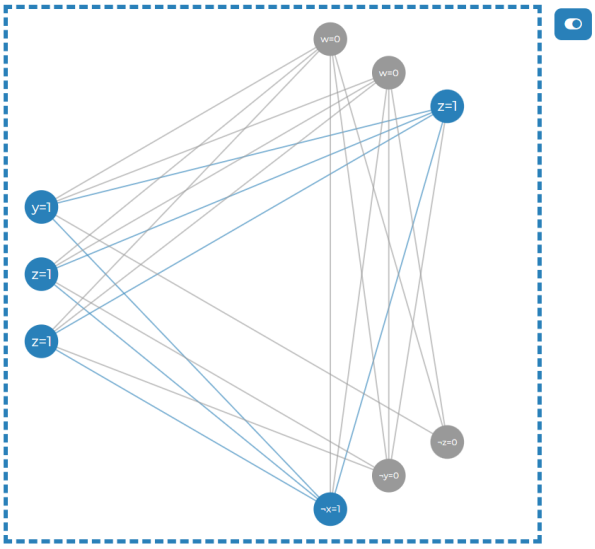


Imagen 4.19: Al pulsar la palanca, se siguen mostrando todos los *cliques* posibles. La diferencia es que ahora también se pueden ver las aristas de los nodos que no pertenecen a un *clique* de tamaño k

En el problema del CLIQUE, se añade además la posibilidad de escoger un solo *clique* de entre todos los disponibles.

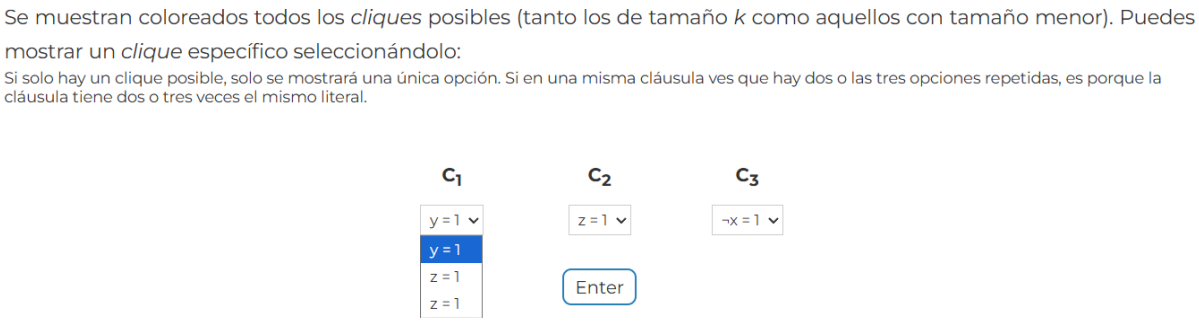


Imagen 4.20: Se puede elegir entre los tres nodos de cada cláusula

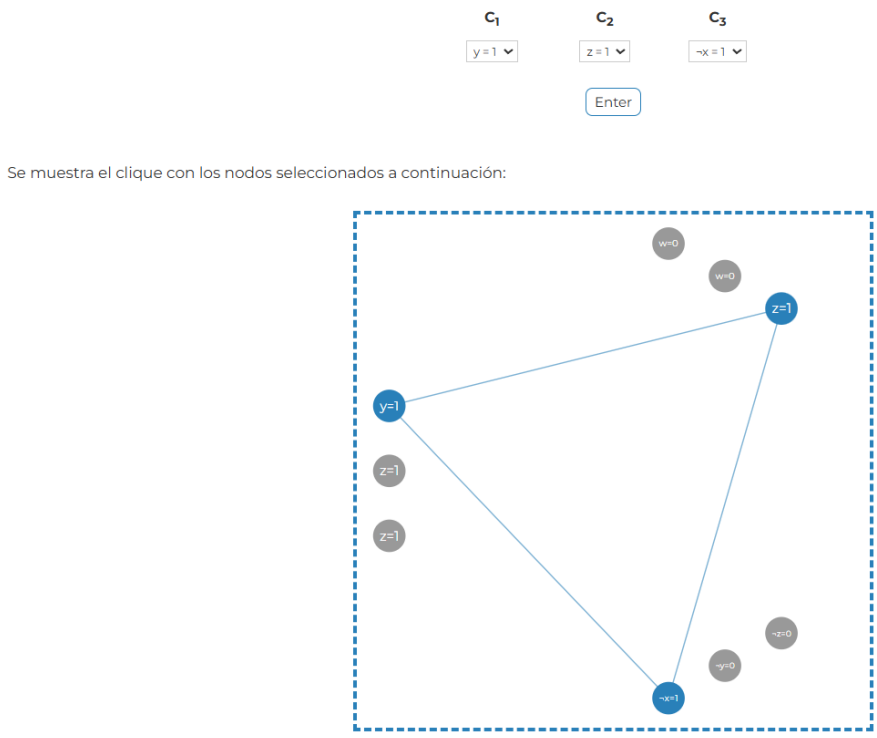


Imagen 4.21: Se muestra un *clique* específico para los valores seleccionados

CAPÍTULO

5 Transformación de fórmula booleana a 3FNC

El primer paso consiste en que el usuario introduzca una fórmula booleana cualquiera, mediante un campo de entrada (input). Posteriormente, esta fórmula es transformada a 3FNC en 4 pasos: simplificación de la expresión, construcción del mapa de Karnaugh, obtención de FNC a partir del mapa de Karnaugh y conversión de FNC a 3FNC. Desarrollamos cada uno de los pasos a continuación.

5.1 Simplificar la fórmula internalizando las negaciones

Antes de simplificar la fórmula, primeramente debemos comprobar que sea una fórmula bien formada (*fbf*). Para ello, se ha realizado una gramática de tipo 2 (libre de contexto). Es la siguiente, usando la notación de Backus-Naur (BNF):

$$
\begin{aligned}
\langle \text{exp} \rangle &::= \langle \text{or}_{\text{exp}} \rangle \mid \langle \text{and}_{\text{exp}} \rangle \mid \langle \text{par}_{\text{exp}} \rangle \mid \langle \text{not}_{\text{exp}} \rangle \mid \langle \text{var} \rangle \\
\langle \text{or}_{\text{exp}} \rangle &::= \langle \text{exp} \rangle \vee \langle \text{exp} \rangle \\
\langle \text{and}_{\text{exp}} \rangle &::= \langle \text{exp} \rangle \wedge \langle \text{exp} \rangle \\
\langle \text{par}_{\text{exp}} \rangle &::= (\langle \text{exp} \rangle) \\
\langle \text{not}_{\text{exp}} \rangle &::= \neg \langle \text{exp} \rangle
\end{aligned}
$$

Donde $\langle var \rangle$ es cualquier variable que cumpla la siguiente expresión regular:

$$[A - Za - z]\backslash d^*$$

Es decir, una letra mayúscula o minúscula, que puede estar acompañada de cualquier número. Nótese que el número que acompaña a la variable es una mera etiqueta, por lo que se aceptan valores que comiencen por 0, como por ejemplo x_{000} o a_{03}.

A partir de esta gramática, se ha creado un analizador sintáctico que verifica que la fórmula introducida solo contenga caracteres válidos y que sea una fórmula bien formada. Esto último se consigue si la fórmula introducida se ajusta a la gramática. Este analizador, implementado en la función `parser`, construye un árbol de análisis descendente.


```
AST: {
  "type": "or",
  "left": {
    "type": "var",
    "name": "a"
  },
  "right": {
    "type": "and",
    "left": {
       "type": "var",
       "name": "b"
    },
    "right": {
      "type": "not",
      "expr": {
        "type": "var",
        "name": "c"
      }
    }
  }
}
```

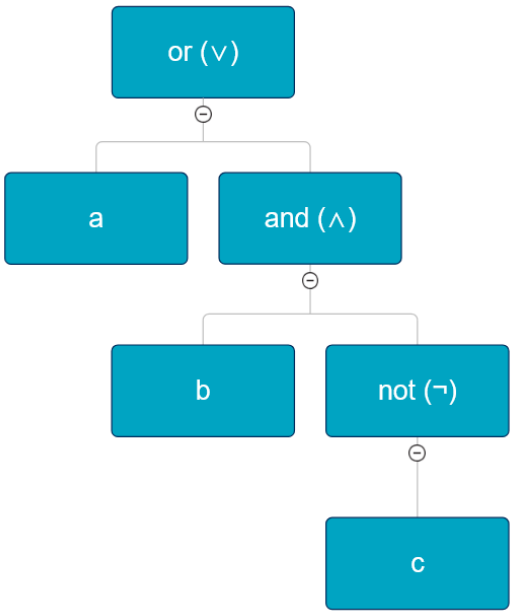



Imagen 5.1: A la izquierda, se muestra el árbol sintáctico para la expresión $a \vee b \wedge \neg c$. A la derecha, se incluye un dibujo de este mismo árbol para mayor claridad

Debido a que el usuario puede introducir cualquier fórmula, es posible que esta contenga ambigüedades. Un ejemplo es la expresión $a \wedge b \vee c \wedge d$. Una misma asignación puede tener distintos resultados según el orden de precedencia escogido. Para la asignación $a = 0, b = 1, c = 1$ y $d = 1$ obtenemos diferentes resultados:

$$\text{Ejemplo 1: } (a \wedge b) \vee (c \wedge d) = (0 \wedge 1) \vee (1 \wedge 1) = 0 \vee 1 = 1$$

$$\text{Ejemplo 2: } a \wedge (b \vee c) \wedge d = 0 \wedge (1 \vee 1) \wedge 1 = 0 \wedge 1 \wedge 1 = 0$$

En el ejemplo 1, hemos elegido que el operador *and* tenga mayor precedencia que el operador *or*. El resultado final es verdadero. Sin embargo, si elegimos que el operador *or* tenga mayor precedencia, entonces estamos en el ejemplo 2 y el resultado es falso. Para evitar estas ambigüedades, se ha establecido un orden de precedencia de operadores, de mayor a menor: paréntesis, operador *not*, operador *and* y operador *or*. Por tanto, según este criterio elegido,

si el usuario introdujera la fórmula $a \wedge b \vee c \wedge d$, nuestro analizador realizaría el árbol del ejemplo 1.

Una vez verificada que la fórmula es una *fbf*, se procede a su simplificación mediante la internalización de las negaciones. Esto implica que, en el árbol sintáctico, las negaciones se "bajan", convirtiéndose en nodos padre de las hojas (es decir, padres de las variables).

```
AST: {
  "type": "not",
  "expr": {
    "type": "or",
       "left": {
         "type": "var",
         "name": "a"
       },
       "right": {
         "type": "and",
           "left": {
             "type": "var",
             "name": "b"
           },
           "right": {
             "type": "not",
               "expr": {
                 "type": "var",
                 "name": "c"
               }
           }
        }
    }
}
```

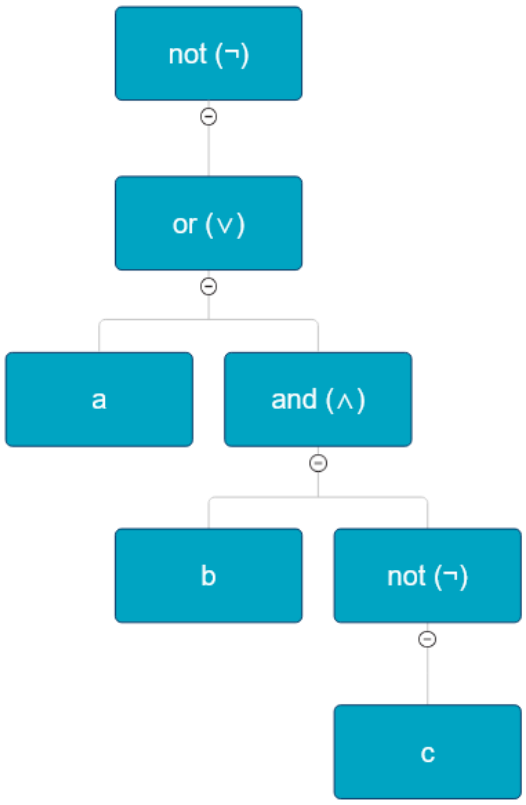


Imagen 5.2: Árbol sintáctico creado para la fórmula $\neg(a \vee b \wedge \neg c)$. Se aprecia que esta fórmula es ambigua. Según la precedencia escogida, esta expresión es analizada como $\neg(a \vee (b \wedge \neg c))$, como se puede ver en la imagen derecha

Si durante el procedimiento de "bajar" las negaciones nos encontramos con un operador *and* u *or*, este pasa a ser su opuesto por las leyes de De Morgan. Es decir, si hallamos un operador *or*, este pasa a ser *and* y viceversa. Este proceso no termina hasta que la negación que se encontraba en una posición "alta", llega a ser padre de una de las hojas, es decir, padre de una variable. Cabe destacar que si esa variable ya está negada, se debe aplicar la ley de la doble negación. A continuación se muestran las reglas mencionadas:

Leyes de De Morgan:

$$\neg(A \vee B) = \neg A \wedge \neg B$$

$$\neg(A \wedge B) = \neg A \vee \neg B$$

Ley de la doble negación:

$$\neg\neg A = A$$

```
PushNeg: {
  "type": "and",
  "left": {
    "type": "not",
    "expr": {
      "type": "var",
      "name": "a"
    }
  },
  "right": {
    "type": "or",
    "left": {
      "type": "not",
      "expr": {
        "type": "var",
        "name": "b"
      }
    },
    "right": {
      "type": "var",
      "name": "c"
    }
  }
}
```

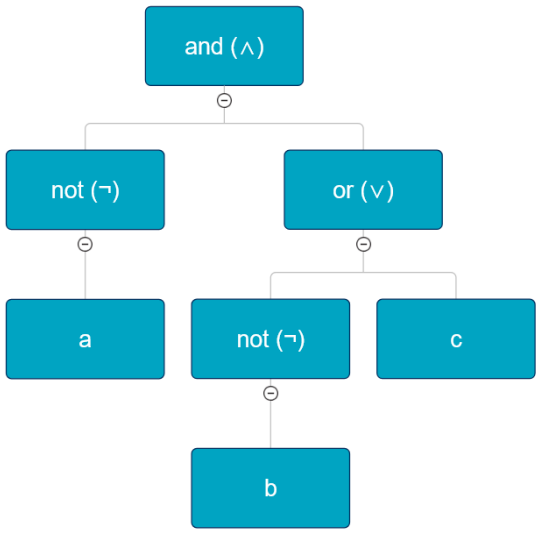


Imagen 5.3: La función `pushNegations` realiza operaciones en el árbol para así internalizar las negaciones. Si comparamos con la imagen 5.2, vemos que lo único que se ha realizado es llevar la negación que era raíz del árbol hacia abajo, convirtiéndola en padre de cada una de las variables

Adicionalmente, el árbol de sintaxis abstracta (AST) generado debe transformarse en una cadena de caracteres. Esta tarea la realiza la función recursiva `printExpr`, que a partir del árbol contenido en el JSON, genera la fórmula en forma de texto. Este resultado será analizado en el siguiente paso, para la realización de los mapas de Karnaugh.

5.2 Construir el mapa de Karnaugh

En el paso anterior, obtuvimos una fórmula con las negaciones ya internalizadas y la convertimos en una cadena de texto utilizando la función `printExpr`. El siguiente paso es la

construcción de los mapas de Karnaugh. Para ello, se debe evaluar esta fórmula con cada una de las posibles asignaciones.

En un mapa de Karnaugh, cada casilla representa el resultado de evaluar la expresión lógica con una asignación de valores concreta, determinados por su fila y columna correspondientes. Por ejemplo, en la imagen 5.4, la casilla en la fila 3 y columna 2 se corresponde con la asignación $x = 1, y = 1$ (fila 3) y $z = 0, w = 1$ (columna 2). Si evaluamos la expresión con esta asignación concreta, el resultado es 1, que es el valor que simboliza esta casilla.

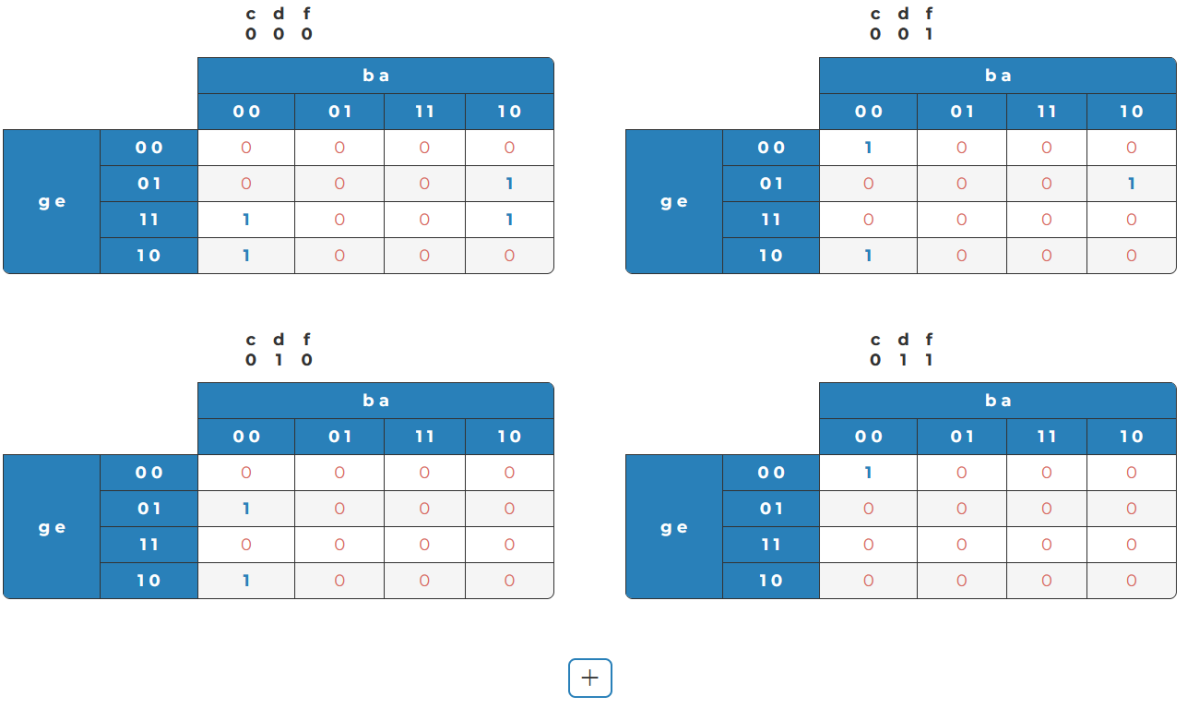
		zw			
		00	01	11	10
xy	00	0	0	1	1
	01	0	1	1	1
	11	0	1	0	0
	10	0	0	1	1

Imagen 5.4: Dibujo de mapa de Karnaugh para la fórmula lógica $(x \wedge \neg y \wedge z) \vee (\neg x \wedge z) \vee (w \wedge y \wedge \neg z)$

Se consiguen los valores de verdad para cada una de las casillas, es decir, para cada una de las asignaciones. Los resultados de cada uno de los mapas se guardan en una matriz. Si hay más de 1 mapa (más de 4 variables), cada uno está asociado a su propia matriz.

Para dibujar el mapa de Karnaugh con los valores de verdad conseguidos, se ha creado un componente de React llamado `PaintTruthTables`. Este devuelve una tabla que simula el mapa, cuyas celdas son los valores para cada una de las asignaciones. Se muestra un ejemplo en la anterior imagen 5.4.

Cuando se introducen más de 6 variables, se crean más de cuatro tablas, lo que dificulta la visualización en la página. Por este motivo, se ha decidido ocultar el resto. Si el usuario desea ver más, se ofrece un botón que carga bloques de cuatro tablas por vez.



c d f = 0 0 0

ge \ ba	00	01	11	10
00	0	0	0	0
01	0	0	0	1
11	1	0	0	1
10	1	0	0	0

c d f = 0 0 1

ge \ ba	00	01	11	10
00	1	0	0	0
01	0	0	0	1
11	0	0	0	0
10	1	0	0	0

c d f = 0 1 0

ge \ ba	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	0	0	0	0
10	1	0	0	0

c d f = 0 1 1

ge \ ba	00	01	11	10
00	1	0	0	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

+

Imagen 5.5: Mapas de Karnaugh para una expresión de 7 variables. Si el usuario desea ver las otras cuatro tablas restantes, solo debe clicar sobre el botón +

Nótese que cuando los mapas tienen más de 4 variables, se deben indicar el resto de ellas encima del mapa. Esto es así porque cada mapa puede contener en su interior hasta 4. Si hubieran más, se indican en la parte superior junto con sus respectivas asignaciones. Esto se consigue a través de la adición del elemento `<caption>`. En la imagen anterior 5.5 se ve claramente cómo se dibuja.

5.3 Obtener FNC a partir del mapa de Karnaugh (hasta 4 variables)

El procedimiento a seguir consta de 3 pasos: primeramente se buscan los maxtérminos (descritos a continuación); seguidamente, se escogen las cajas óptimas; por último, se consigue la Forma Normal Conjuntiva (FNC).

♦ 5.3.1. Búsqueda de los maxtérminos

Los maxtérminos son expresiones lógicas de n variables, en las que cada variable aparece exactamente una vez. En ellas se emplea únicamente el operador complemento (*not*) y la disyunción lógica (*or*). Cada uno de estos términos da como resultado falso solo para una asignación específica: aquella que hace que todos los literales sean falsos, lo que provoca que la disyunción también sea falsa. Por este motivo, los maxtérminos se usan para representar los casos en los que la función booleana da un resultado falso. Por ejemplo, si la función se hace 0 en la asignación $a = 1$, $b = 0$ y $c = 1$, entonces el maxtérmino asociado es $(\neg a \vee b \vee \neg c)$ y únicamente dará como resultado 0 cuando se le asignen esos valores [10, pp. 9–10].

Una vez aclarado qué son los maxtérminos, procedemos a explicar cómo llevar a cabo su búsqueda. Vamos recorriendo la tabla de verdad hasta que nos encontramos un valor 0. Para obtener los máxtérminos, debemos agrupar los ceros en cajas cuyo tamaño sea potencia de 2. Nos interesan las cajas más grandes, pues esto hará que la fórmula reduzca su número de cláusulas. De esta tarea se encarga la función `getBoxWithAdjacentZeros`.

La lógica que se sigue para encontrar las agrupaciones es ir viendo si hay ceros adyacentes. Empezamos analizando si se puede expandir hacia la derecha. Dentro de este condicional, se examina el contenido de la celda adyacente a la derecha. Si es un 0, seguimos ejecutando el bucle hasta asegurarnos de que hay una cantidad de ceros igual a una potencia de 2. Si es así, expandimos el ancho de la caja al doble, multiplicándolo por 2 en cada iteración, ya que se trata de potencias de 2. Si encontramos un 1, `foundMatch` es falso, ya que no hemos podido encontrar una caja más grande. Por tanto, salimos del bucle.

```
1    export function getBoxWithAdjacentZeros(map, i, j, iLength, jLength) {
2        let foundMatch = true;
3        let corner = null;
4
5        // RIGHT SIDE: Expand horizontally to the right
6        for (let l = j + 1; jLength < map[0].length && l < j + 2*jLength &&
             foundMatch; l++) {
7        // If we encounter a cell that is 1 -> stop expanding
8            if (map[i][mod(l, map[0].length)] > 0) {
```

```
                foundMatch = false;

            // If there are 2^n zeros on the right -> expand
            } else if (l === j + 2*jLength - 1) {
                jLength *= 2;
            }
        }

        ...
    }
```

Código 5.1: Se deben encontrar agrupaciones de ceros cuyo tamaño sea una potencia de 2

Este mismo desarrollo se sigue con cada uno de los lados restantes: izquierda, abajo y arriba. En cada una de estas direcciones buscamos agrupaciones. Si las encontramos, nos aseguramos de que la longitud total de la fila o columna sea potencia de 2, consiguiendo así agrupaciones con 2^n elementos, siendo n un número natural.

Adicionalmente, se marcan las casillas que han sido seleccionadas, restando 1 a su valor actual. Esto permite llevar un control de cuántas veces una misma casilla ha sido utilizada en distintas agrupaciones y así poder realizar el paso de optimización más adelante.

La función termina devolviendo un array con la posición de la caja y su tamaño: `[i, j, iLength, jLength]`. Por eficiencia, se ha optado por devolver una lista de 4 números, en vez de una matriz completa con todas las casillas seleccionadas. Los datos que identifican a las cajas son la posición `(i, j)` de la esquina superior izquierda, y las dimensiones `iLength` y `jLength`, alto y ancho, respectivamente. Almacenar otros datos sería redundante e inútil, ya que solo nos importa cómo es la caja. Su contenido ya lo sabemos (todas las casillas son 0).

♦ 5.3.2. Escoger las agrupaciones óptimas

Puede darse el caso en el que hayamos escogido cajas que no hagan falta, ya que las casillas que componen una determinada agrupación pueden haber sido seleccionadas en otras cajas. Para ello, comprobamos una a una las cajas para ver si son óptimas.

```
// 2. See if all boxes are optimal -> if there is a redundant box,
    remove it
for (let box of listMaxterms) {
    if (isOptimalBox(mat, box))
        optimal.push(box);
}
```

Código 5.2: Iteramos sobre las agrupaciones para verificar si son óptimas

Una agrupación es óptima si alguna de las casillas que comprende tiene el valor -1, es decir, solo ha sido seleccionada por esta caja. En el caso de que todos sus valores sean menores que -1, lo que indica que todas sus casillas están ya incluidas en otras cajas, entonces no es óptima y se descarta.

Para descartar una caja, debemos sumar 1 a todas las casillas que la componen. Esto se hace para señalar que una de las cajas ha sido eliminada, permitiendo que otras agrupaciones potencialmente óptimas puedan seleccionarse.

♦ 5.3.3. Obtención de la FNC

Para obtener la Forma Normal Conjuntiva, debemos transformar cada agrupación en un maxtérmino. La función `getCNFFromBox` nos ayuda en este propósito.

```
// 3. From optimal maxterms -> get CNF
for (let [key, value] of mapBoxes) {
    for (let box of value) {
        cnf = cnf.concat(getCNFFromBox(box, variables, key));
        cnf.push("∧")
    }
    cnf.pop(); // Removes last ∧
}
```

Código 5.3: `getCNFFromBox` consigue los maxtérminos, que posteriormente se concatenan con el operador ∧

El paso de agrupación a maxtérmino es sencillo: debemos fijarnos en aquellas variables que no cambian su valor. En el siguiente ejemplo 5.6, la caja azul comprende dos casillas en la fila 3, en las columnas 3 y 4. Considerando ambas casillas, la variable w toma los dos valores posibles (0 y 1), mientras que x, y y z solo toman el valor 1. Para conseguir la FNC, consideramos únicamente las variables que toman un solo valor, en este caso, x, y y z. Para que el resultado del maxtérmino sea 0, debemos negar la variable si su asignación es 1, y mantenerla positiva si es 0. Siguiendo este ejemplo, tomamos sus complementarios: $\neg x$, $\neg y$ y $\neg z$. Como es un maxtérmino, el operador que usamos para unir estas variables es la disyunción. Nos queda:

$$(\neg x \vee \neg y \vee \neg z)$$

Tiene sentido que, cuando la asignación es 1, debamos coger la negación de la variable, ya que si sustituimos en el maxtérmino, el resultado es 0, lo cual coincide con los valores de verdad que toman las casillas.

$$(\neg x \vee \neg y \vee \neg z) = 0 \vee 0 \vee 0 = 0$$

De igual forma, si sus asignaciones fueran 0, nos interesa tener los literales positivos, ya que esto hará que el resultado también sea 0:

$$(x \vee y \vee z) = 0 \vee 0 \vee 0 = 0$$

		zw			
		00	01	11	10
xy	00	0	0	1	1
	01	0	1	1	1
	11	0	1	0	0
	10	0	0	1	1

Imagen 5.6: Ejemplo de mapa de Karnaugh. Notar que cuando dos agrupaciones se solapan, los colores de las casillas involucradas se combinan, simulando la superposición

Implantamos esta misma idea en el código. Recorremos la agrupación y almacenamos las asignaciones que abarca, tanto en las filas como en las columnas. Marcamos las variables que

cambian usando la función `markVariablesThatChanges`.

```
// Save value assigments of first cell in the box
valuesRow = decimalToGrayCode(iStart, varsInRow);
if (variables.length > 1)
    valuesCol = decimalToGrayCode(jStart, varsInCol);
listVars = valuesRow.concat(valuesCol);

for (let i = iStart; i < iStart + iLength; i++) {
    for (let j = jStart; j < jStart + jLength; j++) {
        valuesRow = decimalToGrayCode(mod(i, rows), varsInRow);
        if (variables.length > 1)
            valuesCol = decimalToGrayCode(mod(j, cols), varsInCol);
        // Result of each variable will be 0 if it changes and 1 if it
           does not.
        listVars = markVariablesThatChanges(listVars, valuesRow.concat(
           valuesCol));
    }
}
```

Código 5.4: Almacenamos las asignaciones de las variables en una casilla concreta con `valuesRow` y `valuesCol`. Posteriormente, su concatenación se pasa como argumento de `markVariablesThatChanges`

Se ha visto cómo es el procedimiento cuando se trabaja con 4 variables o menos. En estos casos, la minimización es relativamente sencilla y los mapas de Karnaugh nos permiten encontrar una FNC más simple rápidamente. Sin embargo, al superar las 4, se necesitan comparar las tablas para encontrar las mayores agrupaciones posibles.

5.4 Obtener FNC a partir del mapa de Karnaugh (más de 4 variables)

En caso de tener 5 o más variables, seguimos pasos similares a los antes mencionados. La diferencia ahora es que se deben hacer mayores agrupaciones, uniendo las cajas de cada una de las tablas. Para realizar esta tarea, se han seguido los siguientes pasos:

1. Registrar la posición de los ceros compartidos entre las distintas tablas.
2. Realizar cajas con aquellos ceros que no son compartidos y, por tanto, la única agrupación posible es dentro de esa misma tabla
3. Generar todas las cajas posibles por los ceros compartidos (agrupaciones entre las tablas)
4. Obtener todas las cajas posibles dentro de cada tabla individual
5. Ordenar las cajas obtenidas según su tamaño
6. Seleccionar las cajas usando un algoritmo voraz

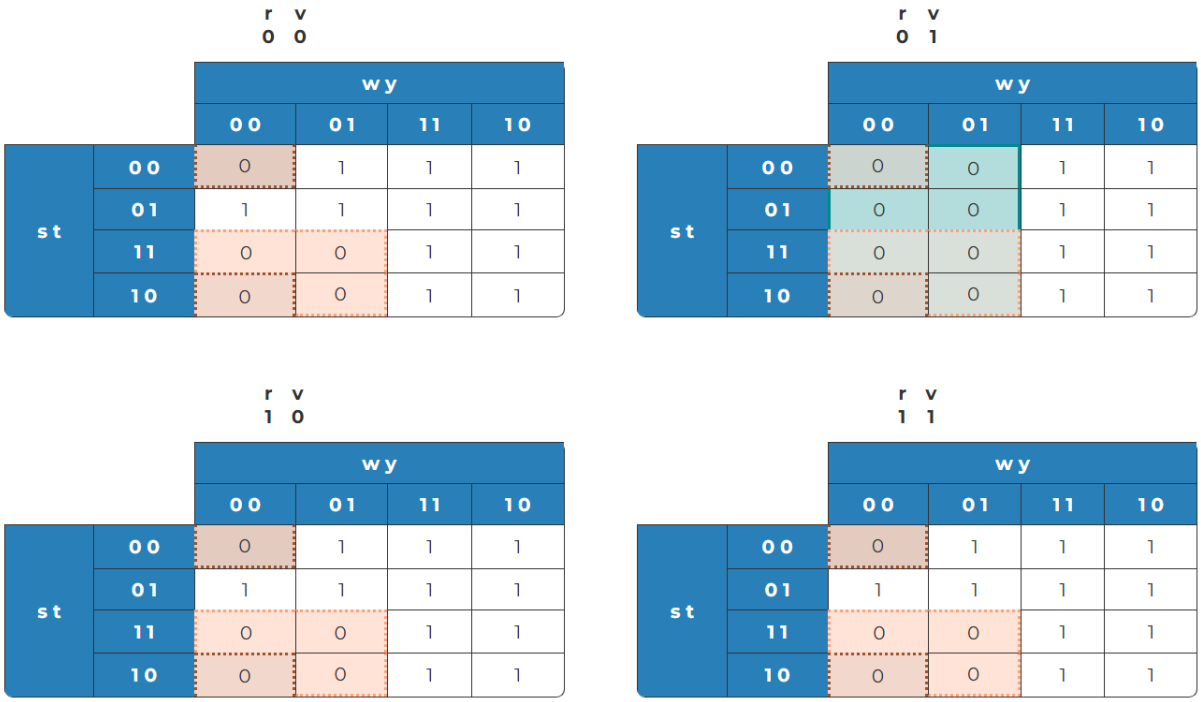


Imagen 5.7: Mapa de Karnaugh para la expresión de 6 variables $\neg(\neg r \wedge v \vee s) \wedge (t \vee \neg w \wedge y) \vee w$. Las cajas que pertenecen a varios mapas se delinean con un patrón de puntos, mientras que las que solo pertenecen a ese mapa, se dibujan con una línea continua. Por ejemplo, la caja marrón es compartida por las 4 tablas, porque está presente en todas ellas y además estas tablas son adyacentes entre sí

♦ 5.4.1. Registrar ceros compartidos entre tablas adyacentes

En primer lugar, solo se comparan las tablas que son adyacentes, es decir, aquellas cuyas asignaciones de las variables externas solo difieren en un valor. Este proceso es similar al que se

emplea para identificar qué casillas son adyacentes dentro de una tabla. En este último caso, las casillas son adyacentes si sus asignaciones difieren en un único bit.

Para hallar qué ceros se comparten entre las tablas, vamos comparando cada una de las casillas de una tabla con la de la otra. Si encontramos un cero compartido, se guarda como una entrada en el mapa, siendo la clave una cadena compuesta por las dos tablas.

Asimismo, no solo deben compararse tablas simples, sino también la combinación de tablas adyacentes entre sí. Para ello, se identifican los ceros compartidos por dichas combinaciones. Al igual que antes, si se encuentra un cero compartido, su posición se almacena en el mapa.

Al final, se devuelve un mapa con las posiciones de los ceros compartidos como valores y las combinaciones de tablas a las que pertenecen como clave.

5.4.2. Agrupaciones con ceros no compartidos

Las casillas que no se comparten entre tablas solo pueden formar parte de agrupaciones dentro de su propia tabla. Por este motivo, se les da prioridad, y no se calculan cajas adicionales para aquellas casillas que ya están incluidas en alguna de ellas.

Las agrupaciones se realizan con la misma función que se mostró anteriormente, `getBoxWithAdjacentZeros`, en la subsección 5.3.1.

5.4.3. Obtener agrupaciones entre tablas para los ceros compartidos

Se escogen agrupaciones cuyo tamaño sea potencia de 2. Al igual que antes, se llama a la función `getBoxWithAdjacentZeros`, y se verifica que las cajas son óptimas, eliminando aquellas que son redundantes.

5.4.4. Obtener agrupaciones en las tablas simples

Este proceso es muy parecido al desarrollado en la subsección 5.3.1, por lo que no es necesario explicarlo de nuevo.

5.4.5. Ordenar las cajas obtenidas según su tamaño

Se ordenan las agrupaciones de mayor a menor tamaño y se introducen en un mapa. Las claves son el tamaño de la caja y los valores son listas que contienen la posición de las cajas y las tablas donde se encuentran. Este es un paso previo al desarrollo del algoritmo voraz de selección de las agrupaciones.

5.4.6. Seleccionar las agrupaciones usando un algoritmo voraz

La función `selectGreedyBoxes` se basa en escoger la mejor opción en cada iteración, con el objetivo de aproximarse a una solución óptima. Este enfoque se corresponde con el principio de los algoritmos voraces, por lo que esta función pertenece a este tipo de algoritmos.

Se comienza introduciendo las agrupaciones que contienen ceros no compartidos, ya que solo pueden formar parte de una única caja. A continuación, se itera sobre el resto de cajas siguiendo el orden previamente establecido: desde las de mayor tamaño hasta las de menor.

Seguidamente, se seleccionan las cajas que contienen el mayor número de casillas no escogidas por otras agrupaciones. Dicho de otra forma, se calcula la diferencia entre el tamaño total de la caja y la cantidad de ceros que ya están presentes en otras cajas. Este criterio se aplica porque las cajas que contienen ceros no agrupados en otras deben ser necesariamente seleccionadas, al ser las únicas que los abarcan.

Se itera desde el número máximo de ceros posible hasta 0. El número máximo de ceros es la agrupación que abarca todas las casillas de todas las tablas. Si una caja es redundante, es decir, que todas sus casillas están cubiertas por otras cajas, se descarta. En caso de empate entre agrupaciones con el mismo número de ceros exclusivos de una caja, se da preferencia a

aquellas que resultan de la combinación de varias tablas.

```
function selectGreedyBoxes(mapFirst, mapOrd) {
    ...
    // Insert boxes that belong to only one table (=mapSeparated)
    for (let [table, boxes] of mapFirst) {
        // Only continues if boxes.length > 0
        for (let box of boxes) {
            if (map.has(table))
                map.get(table).push(box);
            else
                map.set(table, [box]);
            aux = JSON.parse(JSON.stringify(box)).concat(table);
            // Save occupied cells inside map
            saveOccupiedCells(occupiedCells, aux);
        }
    }
    // Get max size that appears in map
    mapOrd.forEach((v, k, m) => { if (k > max) max = k; } );

    for (let free0 = max; free0 > 0; free0--) {
        for (let [size, boxes] of mapOrd) {
            // Size of box needs to be greater than the number of free
               zeros wanted -> e.g.: If we want a box with 7 free zeros, we
                need a 8-box. A 4-box is impossible to contain 7 free0.
            if (size >= free0) {
                for (let i = 0; i < boxes.length; i++) {
                    boxFree0 = size - numberOfNonFreeZeros(boxes[i],
                       occupiedCells);
                    // If the box contains the same number of free 0s as
                       the wanted number -> choose this box
                    if (boxFree0 === free0) {
                        if (map.has(boxes[i][4])) {
                            map.get(boxes[i][4]).push(boxes[i].slice(0,
                               boxes[i].length-1));
                        // If combination of tables has not been inserted
                           before
                        } else {
                            map.set(boxes[i][4], [boxes[i].slice(0, boxes[i
```

```
                                ].length-1)]);
                        }
                        // Save occupied cells inside map
                        saveOccupiedCells(occupiedCells, boxes[i]);
                        // Remove added box
                        boxes.splice(i, 1);
                        i--;
                    // Remove box that does not include free 0s
                    } else if (boxFree0 === 0) {
                        boxes.splice(i, 1);
                        i--;
                    }
                }
            }
        }
    ...
}
```

Código 5.5: Algoritmo voraz para la selección de cajas. La variable free0 indica la cantidad de ceros que son exclusivos de esa determinada caja

5.5 Conversión de FNC a 3FNC

El último paso de este proceso completo consiste en transformar la Forma Normal Conjuntiva (FNC) a su equivalente 3FNC (Forma Normal Conjuntiva con cláusulas de tres literales). Para realizarlo, necesitamos modificar cada maxtérmino para que contenga exactamente 3 literales.

Si un maxtérmino tiene solo uno o dos literales, el procedimiento es extremadamente sencillo, ya que solo hay que repetir uno de los literales hasta alcanzar los tres. Por ejemplo, el maxtérmino (a) se convierte en $(a \vee a \vee a)$ y $(a \vee \neg b)$ en $(a \vee \neg b \vee \neg b)$. En caso de que el maxtérmino ya contenga 3 literales en su interior, no es necesario modificarlo.

Cuando el maxtérmino tiene 4 o más literales, es un poco más complejo. Si encontramos un maxtérmino así, es necesario dividirlo en cláusulas de tres literales encadenadas mediante

nuevas variables auxiliares. El esquema de transformación es el siguiente:

$$(l_1 \vee l_2 \vee \mu_1), (l_3 \vee \neg\mu_1 \vee \mu_2), (l_4 \vee \neg\mu_2 \vee \mu_3), \quad \ldots \quad , (l_{k-2} \vee \neg\mu_{k-4} \vee \mu_{k-3}), (l_{k-1} \vee l_k \vee \neg\mu_{k-3})$$

donde k es el número total de variables, l_i con $1 \leq i \leq k$ cada una de las variables y μ_j con $1 \leq j \leq k-3$ las variables auxiliares.

Este esquema divide los maxtérminos y los convierte en un conjunto equivalente de cláusulas de tres literales [11].

CAPÍTULO

6 Reducción de 3SAT a problemas NP-completos

En este capítulo se describe cómo se realiza la reducción de 3SAT a los problemas CLIQUE, VERTEX-COVER, HAMPATH y SUBSET-SUM. En los tres primeros problemas, la fórmula 3FNC se transforma en un grafo, mientras que para el problema de SUBSET-SUM, el resultado es un conjunto de números, cada uno expresado cifra a cifra en una tabla, de los cuales se selecciona un subconjunto. Para la visualización de los grafos, se ha usado la librería D3.js, la cual permite representar los grafos de forma dinámica. Gracias a ella podemos mover los nodos, resaltar los elementos cuando son seleccionados y aumentar o disminuir el tamaño de la visualización, entre otras funcionalidades.

6.1 Reducción de 3SAT a CLIQUE

El problema 3SAT puede reducirse en tiempo polinómico al problema CLIQUE, lo que implica que cualquier instancia de una fórmula booleana en forma 3FNC (es decir, con cláusulas de exactamente tres literales) puede transformarse en un grafo tal que encontrar un *clique* de un tamaño determinado en dicho grafo es equivalente a encontrar una asignación que satisface la fórmula original. La idea central para construir el *gadget* consiste en crear un grafo no dirigido cuyas estructuras imiten el comportamiento lógico de las cláusulas y literales presentes en la fórmula booleana. La transformación se diseña de modo que cada posible asignación satisfactoria se corresponda con un subconjunto de vértices que forman un *clique* de tamaño k, donde k es el número de cláusulas en la fórmula [1, pp. 302–303].

Para lograr esto, se organiza el grafo resultante en k grupos de tres nodos cada uno, llamados *tripletes*, en los que cada nodo representa un literal de una cláusula. Así, cada cláusula está asociada a un grupo de tres nodos. Se conectan con aristas todos los pares de nodos, con excepción de aquellos que pertenecen al mismo triplete (es decir, a la misma cláusula) o aquellos cuyos literales sean mutuamente opuestos (como x y $\neg x$). De este modo, cualquier selección de un

nodo por cláusula que no sea opuesto y que esté completamente conectado con los otros nodos seleccionados (es decir, forme un *clique* de tamaño k) equivale a una asignación booleana que satisface un literal en cada cláusula, lo que garantiza que la fórmula completa se satisface. Esta construcción establece así la equivalencia entre la satisfacibilidad de la fórmula y la existencia de un *clique* en el grafo [1, pp. 302–303].

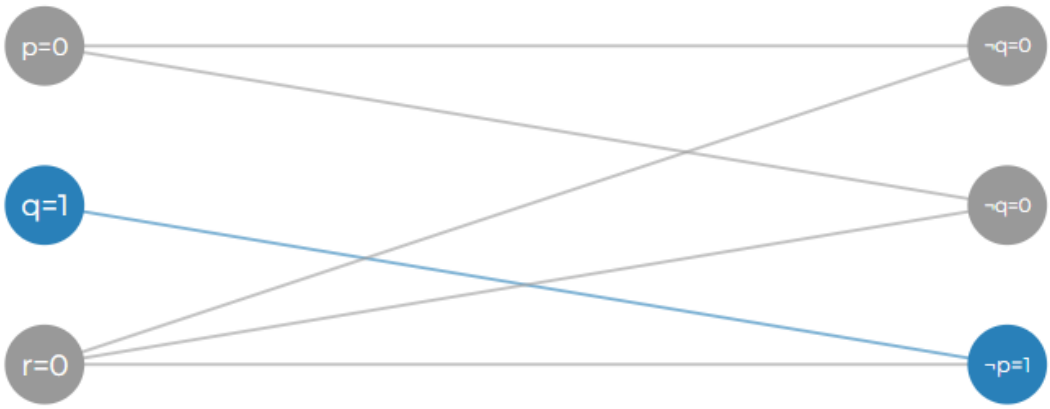


Imagen 6.1: Un posible *k-clique*, con $k = 2$ en este ejemplo, es el que contiene los vértices q y $\neg p$ y equivale a la asignación $p = 0, q = 1$ y $r = 0$, que hace satisfacible a la fórmula $(p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg q)$

Para dibujar los nodos del grafo, es necesario calcular los ángulos en los que se colocará cada triplete. Esto se logra dividiendo los 360 grados de la circunferencia entre el número de cláusulas (equivalente al número de tripletes). En cada uno de estos ángulos se posicionan tres nodos. Estos vértices se guardan en una lista llamada `nodes`.

```
function createListNodes (litClaus) {
    ...
    while (angle < 360 && i < numberClause) {
        // Middle literal
        // Calculate distance in horizontal axis ((halfSide-sizeNode*2)
            -> so as not to leave view)
        xMiddle = halfSide - (halfSide-SIZE_NODE*2) * sinDegrees(90-
            angle);
        // Calculate distance in vertical axis
        yMiddle = halfSide - (halfSide-SIZE_NODE*2) * sinDegrees(angle)
            ;
```

```
9             nodes.push({id: (i*3)+1, literal: litClaus[i][1], x: xMiddle, y
                  : yMiddle, color: COLORS[color]});
10            // Side left literal
11            xSide = xMiddle + DISTANCE_BETWEEN_LITERALS * sinDegrees(angle)
                  ;
12            ySide = yMiddle - DISTANCE_BETWEEN_LITERALS * sinDegrees(90-
                  angle);
13            nodes.push({id: (i*3), literal: litClaus[i][0], x: xSide, y:
                  ySide, color: COLORS[color]});
14            // Side right literal
15            xSide = xMiddle - DISTANCE_BETWEEN_LITERALS * sinDegrees(angle)
                  ;
16            ySide = yMiddle + DISTANCE_BETWEEN_LITERALS * sinDegrees(90-
                  angle);
17            nodes.push({id: (i*3)+2, literal: litClaus[i][2], x: xSide, y:
                  ySide, color: COLORS[color]});
18
19            angle += addAngle;
20            color = (color+1)%COLORS.length;
21            i++;
22        }
23        return nodes;
24    }
```

Código 6.1: Para crear los nodos, se divide la circunferencia en tantos ángulos como cláusulas tenga la fórmula 3FNC

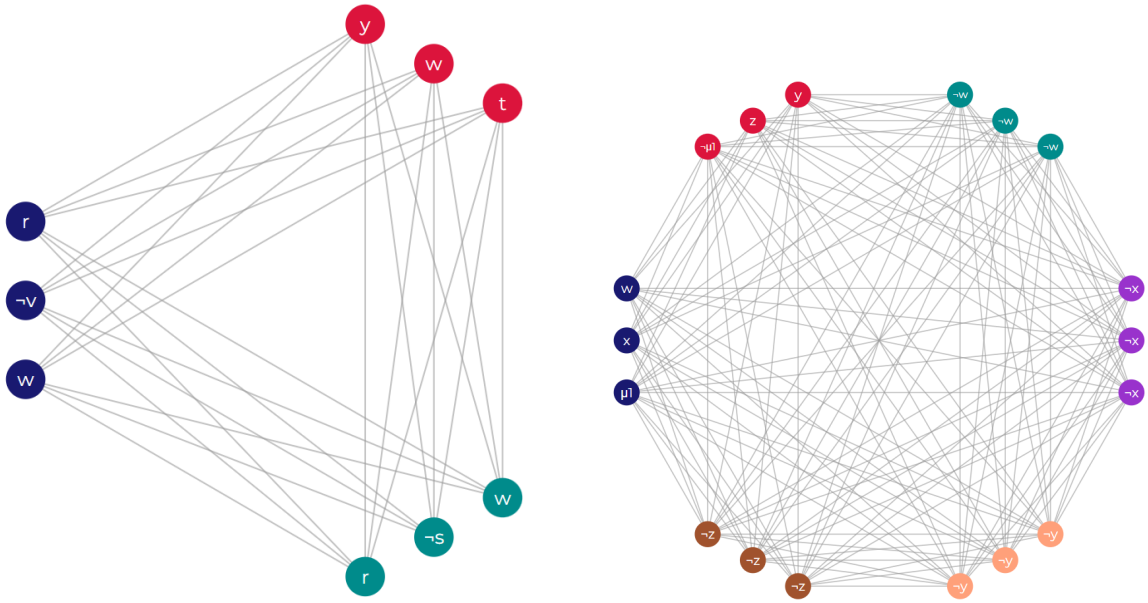


Imagen 6.2: Cuanto mayor número de cláusulas, menor tendrán que ser los ángulos de separación

En la creación de las aristas, se conectan únicamente aquellos nodos que no pertenecen al mismo triplete y que no son literales opuestos. Igualmente, se crean aristas imaginarias para las fórmulas insatisfacibles, para así mostrar qué conexiones faltan para que haya al menos un *clique*.

En la implementación, se recorren los nodos de cada triplete y se verifica si cumplen las condiciones para ser conectados mediante una arista. Si es así, dicha conexión se añade a la lista `links` que posteriormente se utiliza para construir el grafo.

```
function createLinks (litClaus) {
    ...
    for (let iSrc = 0; iSrc < litClaus.length-1; iSrc++) { // Last
       clause is not necessary -> there are no more connections
        for (let jSrc = 0; jSrc < litClaus[iSrc].length; jSrc++) { //
           Go over 3 literals of this clause

            // Pick source node (=literal)
            source = litClaus[iSrc][jSrc];
            posSrc = (iSrc * 3) + jSrc;

            // Create links for each clause (from left to right)
```

```
                for (let iTrgt = iSrc + 1; iTrgt < litClaus.length; iTrgt
                    ++) { // Starts from next clause
                    for (let jTrgt = 0; jTrgt < litClaus[iTrgt].length;
                        jTrgt++) {
                        target = litClaus[iTrgt][jTrgt];
                        posTrg = (iTrgt*3 + jTrgt);
                        // If the source and target literals satisfy the
                            condition of being connected, create a link
                        link = createLinksIfNecessary(source, posSrc,
                            target, posTrg);
                        if (link !== undefined)
                            links.push(link);

                        // ¬A <----> A -> Creates imaginary line for
                            explication of False expression (contradiction)
                        else if (isFalseExp) {
                            link = {source: posSrc, target: posTrg};
                            linksFalse.push(link);
                        }
                    }
                }
            }
        }
        return [links, linksFalse];
    }
```

Código 6.2: Se crea una arista si los nodos no pertenecen al mismo triplete y no son opuestos. Asimismo, las aristas imaginarias se guardan en la lista `linksFalse`

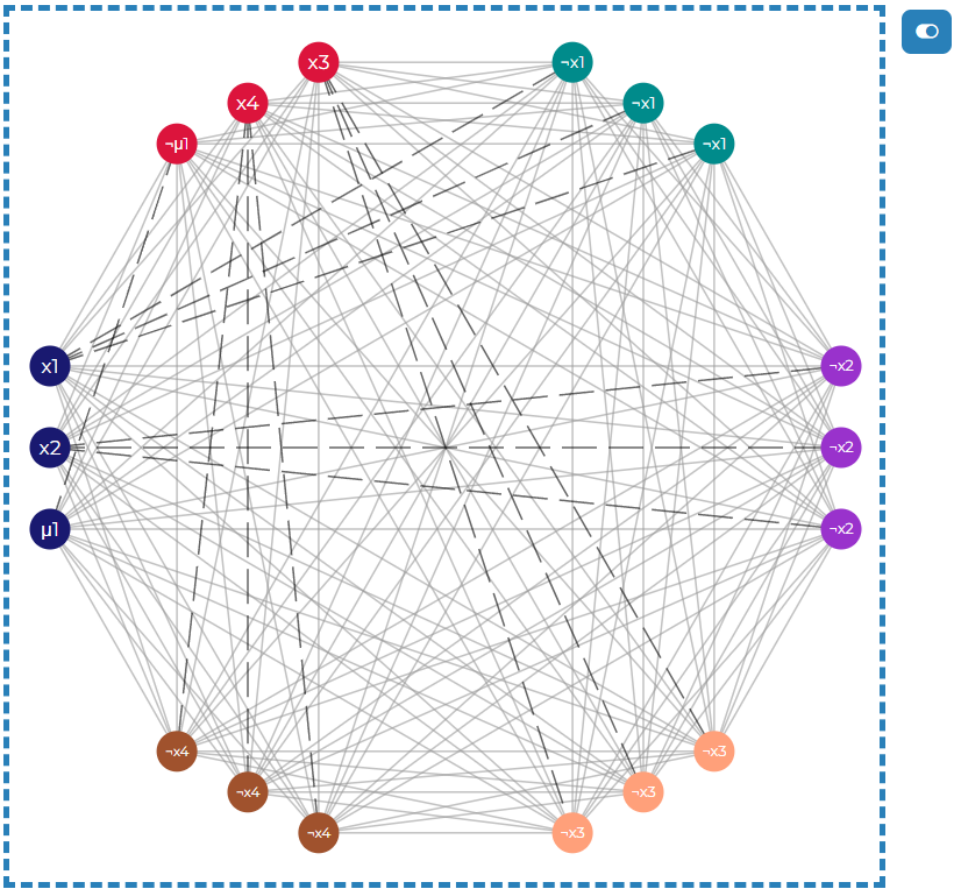


Imagen 6.3: Las aristas imaginarias se señalan con líneas discontinuas

Una vez obtenidos los nodos y aristas, se construye el grafo utilizando la librería D3.js. Este paso es común a todos los grafos, ya que comparten la estructura básica de aristas y nodos, siendo la creación de los elementos SVG que los representan muy parecida. Por este motivo, describiremos esta parte únicamente en este apartado, evitando repeticiones innecesarias para el resto de los problemas.

Se utilizan tres listas principales: `nodes`, que contiene los nodos del grafo; `links`, con las aristas reales del *gadget*; y `linksFalse`, que representa las aristas imaginarias. En el código, cada arista se dibuja a partir de las coordenadas (x, y) de los nodos que conecta.

```
1    linkGroup.selectAll('line')
2        .data(links)
3        .join('line')
4        .attr('stroke-width', 1.5)
5        .attr('x1', d => nodes.find( ({id:num}) => num === d.source).x)
6        .attr('y1', d => nodes.find( ({id:num}) => num === d.source).y)
```

```
        .attr('x2', d => nodes.find( ({id:num}) => num === d.target).x)
        .attr('y2', d => nodes.find( ({id:num}) => num === d.target).y)
        .attr('class', d => `link-${d.source} link-${d.target}`);

    linkFalseGroup.selectAll('line')
        .data(linksFalse)
        .join('line')
        .attr('stroke-width', 1.5)
        .attr('x1', d => nodes.find( ({id:num}) => num === d.source).x)
        .attr('y1', d => nodes.find( ({id:num}) => num === d.source).y)
        .attr('x2', d => nodes.find( ({id:num}) => num === d.target).x)
        .attr('y2', d => nodes.find( ({id:num}) => num === d.target).y)
        .attr('class', d => `linkF-${d.source} linkF-${d.target}`);

    nodeGroup.selectAll('circle')
        .data(nodes)
        .join('circle')
        .attr('cx', d => d.x)
        .attr('cy', d => d.y)
        .attr('r', SIZE_NODE)
        .attr('fill', d => d.color)
        .call(drag);
```

Código 6.3: Se crean nodos, aristas del *gadget* y aristas imaginarias si preocede

Asimismo, cuando se arrastra un nodo, su posición y la de todas las aristas conectadas a él deben actualizarse dinámicamente. Durante la interacción, se modifica temporalmente el color de las aristas del nodo seleccionado como retroalimentación a la acción del usuario. Una vez finalizado el arrastre, estos elementos recuperan su color original.

```
    let drag = d3.drag()
    .on('start', (event, d) => {
        // Change color of links on drag start
        d3.selectAll(`.link-${d.id}`).attr('stroke', d.color);
        d3.selectAll(`.linkF-${d.id}`).attr('stroke', d.color);
    })
    .on('drag', (event, d) => {
        d.x = event.x;
```

```
        d.y = event.y;
    })
    .on('end', (event, d) => {
        // Reset color of links on drag end
        d3.selectAll(`.link-${d.id}`).attr('stroke', '#999');
        d3.selectAll(`.linkF-${d.id}`).attr('stroke', '#000');
    });
```

Código 6.4: Cuando se arrastra un nodo, debe cambiar la posición de este y de las aristas. Adicionalmente, también cambian de color las aristas

6.2 Reducción de 3SAT a VERTEX-COVER

El propósito fundamental de esta reducción es demostrar que cualquier problema de NP puede reducirse en tiempo polinómico a VERTEX-COVER. Para ello, se construye una reducción desde el problema 3SAT. Esta reducción transforma una fórmula booleana en Forma Normal Conjuntiva de tres literales por cláusula (3FNC) en un grafo G y un número k, de tal manera que la fórmula es satisfacible si y solo si el grafo tiene un *vertex cover* de tamaño k [1, pp. 312–313].

La clave de la construcción está en diseñar dos tipos de *gadgets*: uno para representar las variables y otro para las cláusulas. Cada *gadget* de las variables consiste en dos nodos conectados entre sí, etiquetados con un literal y su negación. Uno de estos dos nodos debe formar parte del *vertex cover*, y la elección de cuál representa una asignación de verdad (*TRUE* o *FALSE*) para la variable. Por su parte, cada *gadget* de cláusula contiene tres nodos, uno por literal, que se conectan entre sí y también con los nodos correspondientes de los *gadgets* de las variables. Esta construcción obliga a que, para cada cláusula, al menos dos de los tres nodos deben incluirse en el *vertex cover*. Se realiza seleccionando el literal que satisface la cláusula (sin incluir su nodo en el *vertex cover*) y se añaden al *vertex cover* los otros dos nodos del *gadget* de la cláusula. De este modo, se simula que la cláusula queda satisfecha si al menos uno de sus literales es verdadero [1, pp. 312–313].

Finalmente, se define el valor de k como $k = m + 2l$, donde m es el número de variables y l el número de cláusulas. Este valor asegura que la única forma de tener un *vertex cover* de tamaño

exacto k es mediante una asignación satisfacible, ya que de esta forma se elige un nodo por cada *gadget* de las variables (en total m nodos) y dos nodos por cada cláusula (lo que suma $2l$ nodos). De este modo, se cubren todas las aristas del grafo utilizando exactamente $k = m + 2l$ nodos [1, pp. 312–313].

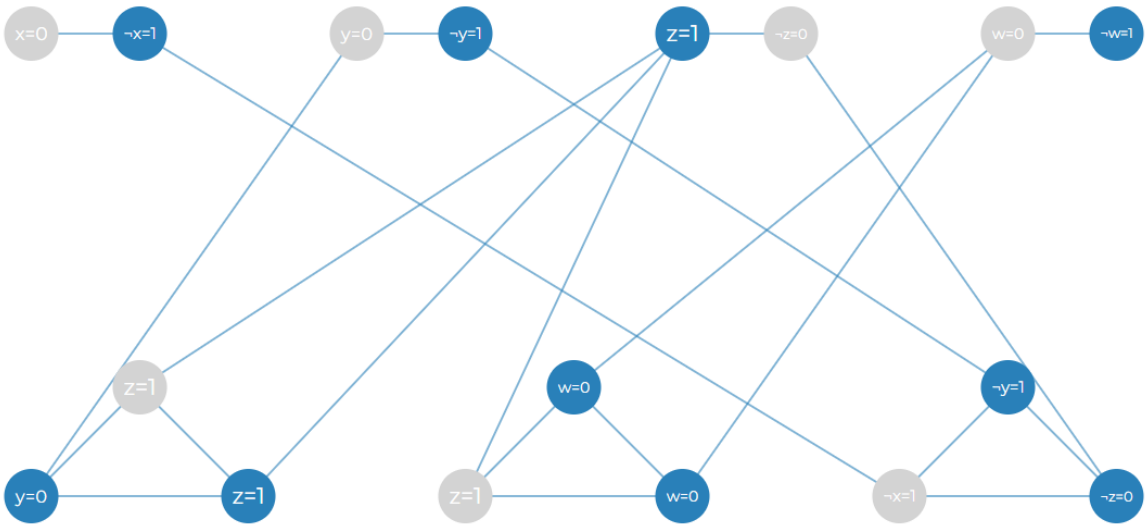


Imagen 6.4: Un posible *k-vertex cover*, con $k = 10$ en este ejemplo, es el que contiene los vértices azules. Equivale a la asignación $x = 0, y = 0, z = 1$ y $w = 0$ que hace satisfacible a la fórmula $(y \vee z \vee z) \wedge (z \vee w \vee w) \wedge (\neg x \vee \neg y \vee \neg z)$

Comenzamos realizando el *gadget* de las variables. Este paso es fácil, pues únicamente se deben representar dos nodos por variable: uno para el literal positivo y otro para el negativo.

```
1   // Paint 2 nodes per variable: x and ¬x
2   for (let v of vars) {
3       // Positive variable
4       nodes.push({id: id, literal: v, x: pos, y:
            DISTANCE_BETWEEN_LITERALS, color: COLORS[color]});
5       // Negative variable
6       pos += DISTANCE_BETWEEN_LITERALS;
7       nodes.push({id: id+1, literal: "¬" + v, x: pos, y:
            DISTANCE_BETWEEN_LITERALS, color: COLORS[color]});
8       // Add extra distance between different variables
9       pos += 2*DISTANCE_BETWEEN_LITERALS;
10      id += 2;
11      color = (color+1)%COLORS.length;
12  }
```

Código 6.5: Por cada variable se crean 2 nodos, uno al lado del otro

Seguidamente, se crean los nodos correspondientes al *gadget* de las cláusulas. Este caso también es relativamente sencillo, ya que solo hay que dibujar tres nodos (uno por literal) en cada cláusula.

```
    // Paint 2 nodes per variable: x and ¬x
    for (let c of litClaus) {
        // Paint clauses
        nodes.push({id: id, literal: c[0], x: pos, y: HEIGHT -
            DISTANCE_BETWEEN_LITERALS, color: COLORS[color]});

        // Second variable of the clause
        pos += DISTANCE_BETWEEN_LITERALS;
        nodes.push({id: id+1, literal: c[1], x: pos, y: HEIGHT -
            DISTANCE_BETWEEN_LITERALS*2, color: COLORS[color]});

        // Third variable of the clause
        pos += DISTANCE_BETWEEN_LITERALS;
        nodes.push({id: id+2, literal: c[2], x: pos, y: HEIGHT -
            DISTANCE_BETWEEN_LITERALS, color: COLORS[color]});

        pos += 2*DISTANCE_BETWEEN_LITERALS; // Add extra distance between
            different variables
        id += 3;
        color = (color+1)%COLORS.length;
    }
```

Código 6.6: Se crean 3 nodos por cláusula, que representan cada uno de los literales en su interior

Se procede ahora con la creación de las aristas. Primeramente, se conectan los nodos que representan los literales opuestos de cada variable en su *gadget*. A continuación, unimos entre sí los nodos que forman cada una de las cláusulas. Por último, para conectar el *gadget* de las variables y el de las cláusulas, debemos enlazar los literales análogos. Esto quiere decir que, si A es una variable cualquiera, debemos unir el nodo A del *gadget* de la variable con todos los literales A que aparecen en las cláusulas. Lo mismo con $\neg A$, tenemos que conectarlo con todos los nodos $\neg A$ del *gadget* de las cláusulas.

6.3 Reducción de 3SAT a HAMPATH

En esta ocasión, se quiere transformar una fórmula 3FNC en un grafo dirigido, de manera que exista un camino hamiltoniano de un nodo s a un nodo t si y solo si la fórmula es satisfacible.

Para construir este grafo G, se emplean dos tipos de *gadgets*, los cuales simulan el comportamiento de las variables y cláusulas. Cada variable se representa mediante una estructura en forma de diamante, la cual puede recorrerse de dos formas distintas, de izquierda a derecha o de derecha a izquierda, reflejando así las posibles asignaciones de verdadero o falso, respectivamente. Las cláusulas, por su parte, se representan como nodos individuales. Para que el camino hamiltoniano pase por todos los nodos del grafo (incluidos los de las cláusulas), debe utilizar los caminos correctos en los diamantes de las variables, lo que implica elegir una combinación de valores de verdad que satisfaga la fórmula [1, pp. 314–316].

El grafo se construye conectando los nodos de cada gadget de variable con los nodos de cláusula en función de la aparición de los literales. Si un literal x_i (o su negación) está presente en una cláusula c_j, se crean aristas desde un par específico de nodos en el diamante de x_i hacia el nodo que representa la cláusula c_j. Al finalizar este proceso, el grafo G está completamente definido [1, pp. 316–317].

Este diseño garantiza que un camino hamiltoniano de s a t existe si y solo si la fórmula es satisfacible. Con una asignación de verdad que haga verdadera la fórmula, se podrá visitar todos los nodos exactamente una vez, incluyendo los de las cláusulas. Esta correspondencia establece que resolver HAMPATH es al menos tan difícil como 3SAT, completando así la demostración de NP-completitud [1, pp. 317–318].

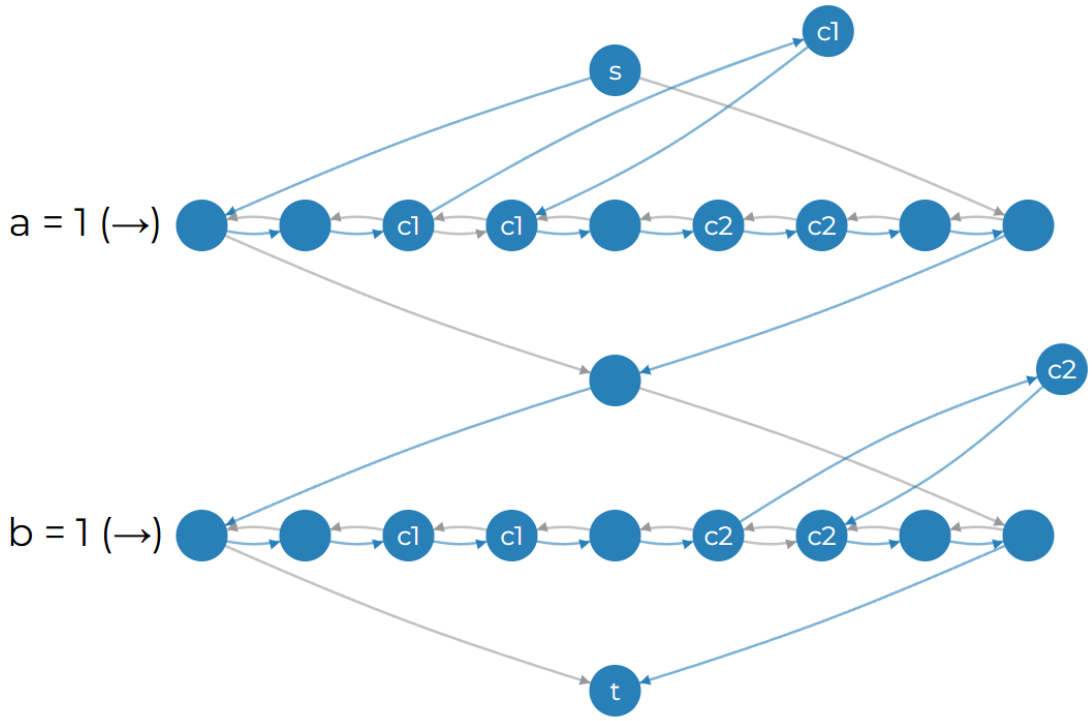


Imagen 6.5: La asignación $a = 1$ y $b = 1$ hace satisfacible a la fórmula $(a \vee a \vee a) \wedge (b \vee b \vee b)$, por lo que hay un camino hamiltoniano que empieza en s y termina en t

Primeramente, se crea el nodo de partida s. A continuación, se construyen los diamantes correspondientes a cada variable de la fórmula. Cada uno de estos diamantes incluye una fila central compuesta por pares de nodos, donde cada par representa una de las cláusulas. Cuando alcanzamos el último diamante, se crea el nodo de destino t, que marca el final del camino hamiltoniano.

```
...
// START NODE 's'
nodes.push({id: id, literal: "s", x: WIDTH_VIEW/2, y: posY, color:
    COLOR});
id++;

// PAINT DIAMONDS -> Paint 1 diamond for each variable
for (let [k, v] of mapValsVars) {

    i++;
    posY += 1.5 * DISTANCE_BETWEEN_LITERALS;
    posX = calculateDistanceCenter((3*litClaus.length + 3)/2);
    text = v === 1 ? "(→)" : "(←)";
```

```
13          texts.push({literal: k + " = " + v + " " + text, x: EXTRA_PADDING,
                y: posY});
14
15          // MIDDLE NODES
16          for (let j = 0; j < 3*litClaus.length + 3; j++) {
17              // Clause nodes
18              if (j !== 0 && j !== (3*litClaus.length + 2) && j % 3 !== 1) {
19                  nodes.push({id: id, literal: "c" + clause, x: posX, y: posY
                        , color: COLOR});
20
21                  // If we have painted 2 nodes -> get next number clause
22                  if (j % 3 === 0) {
23                      clause++;
24                      color = (color+1)%COLORS.length;
25                  }
26
27              // Intermediate nodes
28              } else  {
29                  nodes.push({id: id, literal: "", x: posX, y: posY, color:
                        COLOR});
30              }
31
32              posX += DISTANCE_BETWEEN_LITERALS;
33              id++;
34          }
35
36          // Paint BOTTOM VERTEX of diamond -> if it is ending variable then
                we need to paint ending node 't'
37          posY += 1.5 * DISTANCE_BETWEEN_LITERALS;
38          // All variables have been painted -> finish graph with t node
39          if (i === mapValsVars.size) {
40              nodes.push({id: id, literal: "t", x: WIDTH_VIEW/2, y: posY,
                    color: COLOR});
41
42          // Not last variable -> paint vertex diamond
43          } else {
44              nodes.push({id: id, literal: "", x: WIDTH_VIEW/2, y: posY,
                    color: COLOR});
45          }
```

```
46          id++;
47
48          ...
```

Código 6.7: Se crea un diamante por variable, además del nodo de partida s y de destino t

Por último, una vez generados todos los diamantes, se añaden los nodos que representan las cláusulas.

```
1   // PAINT NODE CLAUSES
2   posY = mapValsVars.size === litClaus.length ?
3               HEIGHT_VIEW/(litClaus.length+1) - DISTANCE_BETWEEN_LITERALS
4           :
5               HEIGHT_VIEW/(litClaus.length+1);
6   for (let i = 0; i < litClaus.length; i++) {
7       nodes.push({id: id, literal: "c" + clause, x: WIDTH_VIEW -
            EXTRA_PADDING, y: posY, color: "#999"});
8       id++;
9       posY += HEIGHT_VIEW/(litClaus.length+1);
10      color = (color+1)%COLORS.length;
11      clause++;
12  }
```

Código 6.8: Los nodos de las cláusulas se crean al final

Para crear las aristas, comenzamos realizando las uniones en cada uno de los diamantes. Las conexiones que unen los vértices exteriores del diamante son fáciles, ya que solo son cuatro y sus direcciones van del nodo superior al inferior.

Asimismo, las conexiones de los nodos interiores se deben hacer en ambas direcciones, tanto de izquierda a derecha como de derecha a izquierda, para reflejar cuando la variable toma el valor verdadero o falso, respectivamente. Se pueden ver estas aristas representadas en el interior de cada diamante en la imagen 6.5.

Para conectar los diamantes con los nodos cláusula, primero se tiene que comprobar si dentro de la cláusula está contenido el literal positivo o negativo de una variable. Si aparece el literal positivo, el recorrido va de izquierda a derecha, y por tanto la conexión hacia los nodos cláu-

sulas va desde el nodo izquierdo, hasta el nodo cláusula y vuelta al nodo derecho del centro del diamante. Si es negativo, el recorrido es al contrario, es decir, de derecha a izquierda. Por tanto, se recorre primero el nodo intermedio derecho, después el nodo cláusula y por último el izquierdo.

```
1    // Connection between diamonds clauses and external clauses
2    // Left node
3    if (j !== 0 && j !== (3*litClaus.length + 2) && j % 3 === 2) {
4        // If clause contains negation of variable -> From clause to
             diamond
5        if (litClaus[clause].includes("¬" + k)) {
6            links.push({source: idClauses + clause, target: src, curve:
                 CURVE_HEIGHT_CLAUSES, color: (v === 0 && !clausesSelected.
                 includes(idClauses + clause))? COLOR : "#999"});
7            ...
8        }
9        // If clause contains positive literal of variable -> From diamond
             to clause
10       if (litClaus[clause].includes(k)) {
11           links.push({source: src, target: idClauses + clause, curve: -
                 CURVE_HEIGHT_CLAUSES, color: (v === 1 && !clausesSelected.
                 includes(idClauses + clause)) ? COLOR : "#999"});
12           ...
13       }
14   // Right node
15   } else if (j !== 0 && j !== (3*litClaus.length + 2) && j % 3 === 0) {
16       // If clause contains negation of variable -> From diamond to
             clause
17       if (litClaus[clause].includes("¬" + k)) {
18           links.push({source: src, target: idClauses + clause, curve: -
                 CURVE_HEIGHT_CLAUSES, color: (v === 0 && !clausesSelected.
                 includes(idClauses + clause)) ? COLOR : "#999"});
19           ...
20       }
21       // If clause contains positive literal of variable -> From clause
             to diamond
22       if (litClaus[clause].includes(k)) {
23           links.push({source: idClauses + clause, target: src, curve:
```

```
                CURVE_HEIGHT_CLAUSES, color: (v === 1 && !clausesSelected.
                includes(idClauses + clause)) ? COLOR : "#999"});
            ...
        }
        ...
    }
```

Código 6.9: La dirección de las aristas que conectan los diamantes con los nodos cláusulas varía según si la variable representada aparece en la cláusula como literal positivo o negativo

6.4 Reducción de 3SAT a SUBSET-SUM

Dada una fórmula booleana en Forma Normal Conjuntiva con tres literales por cláusula (3FNC), se construye una instancia del problema SUBSET-SUM tal que existe un subconjunto que suma exactamente el valor objetivo t si y solo si la fórmula es satisfacible [1, pp. 320].

Cada variable de la fórmula se representa mediante un par de números, y_i y z_i, que están diseñados de tal manera que solo uno de los dos puede ser parte del subconjunto solución t, lo que representa una asignación de verdadero o falso a esa variable. Los números utilizados se expresan en notación decimal y su estructura se divide en dos partes: la primera codifica la identidad de la variable y la segunda refleja su participación en las cláusulas de la fórmula [1, pp. 320–322].

Siguiendo con las cláusulas, cada columna de la parte derecha de los números representa una. Si un literal aparece en una cláusula, se coloca un 1 en la posición correspondiente de su número asociado. Además, por cada cláusula c_j se agregan dos números adicionales g_j y h_j, que tienen una forma determinada para asegurar que exactamente esa cláusula deba sumar 3 en el objetivo t, lo cual impone que al menos un literal de cada cláusula se evalúe como verdadero [1, pp. 320–322].

El número objetivo t se forma con una secuencia de l unos (para las variables) seguidos de k treses (para las cláusulas), siendo l el número de variables y k el número de cláusulas. Se asegura así que se seleccionen exactamente una representación por variable y que se satisfagan

todas las cláusulas [1, pp. 320–322].

		1	2	3	C_1	C_2	C_3
A	y_1	1	0	0	1	0	0
	z_1	0	0	0	0	0	0
B	y_2	0	0	0	0	0	0
	z_2	0	1	0	0	1	0
C	y_3	0	0	1	0	0	1
	z_3	0	0	0	0	0	0
C_1	g_1				1	0	0
	h_1				1	0	0
C_2	g_2				0	1	0
	h_2				0	1	0
C_3	g_3				0	0	1
	h_3				0	0	1
t		1	1	1	3	3	3

Imagen 6.6: La asignación $A = 1, B = 0$ y $C = 1$ hace satisfacible a la fórmula $(A \vee A \vee A) \wedge (\neg B \vee \neg B \vee \neg B) \wedge (C \vee C \vee C)$, por lo que hay un subconjunto cuya suma es el valor t

Empezamos creando la cabecera de la tabla. Cada columna de la parte izquierda simboliza una variable, mientras que la parte derecha se corresponde con cada una de las cláusulas.

A continuación, se realizan las filas que representan las variables. La parte izquierda simboliza las asignaciones. Si la variable x_i es verdadera, se pondrá un 1 en la fila y_i. Si es falsa, se pondrá un 1 en la fila z_i.

La parte derecha se corresponde con las cláusulas. Si en una cláusula aparece el literal positivo y la asignación de la variable i es verdadera, y_i recibe el valor 1 en la columna perteneciente a esa cláusula. Al contrario, si aparece su literal negado y su asignación es falsa, la fila z_i tiene un 1 en la columna de la cláusula i. Solo en estos casos se pone un 1, ya que son los únicos que hacen verdadera a la cláusula. En el resto, los valores son 0.

```
<td>
    {   litClaus[i].includes(variable) &&
        ((!variable.includes("¬") && valueVar ===1) ||
        (variable.includes("¬") && valueVar ===0)) ? 1 : 0
    }
```

```
    </td>
```

Código 6.10: Si la asignación de la variable (`valueVar`) es 1 (verdadera) y en la cláusula aparece su literal positivo, ponemos un 1 en la cláusula correspondiente. También el resultado es 1 si la asignación de la variable es 0 (falsa) y aparece su literal negado (variable.includes("¬"))

Para las filas de las cláusulas, solo se rellena la parte derecha, que se corresponde con las columnas de las cláusulas. El valor de g_j y h_j dependerá de cuántos literales hacen verdadera a la cláusula j. Su valor es 1 cuando se necesita que la suma total sea 3. En caso de que la cláusula sea falsa, aunque g_j y h_j aporten un 1 cada uno, no será suficiente para sumar 3, ya que la suma total será de 2.

```
    {/* For row g_i -> numberTimes = 3
        For row h_i -> numberTimes = 2 */}
    <td>
        {countTrues[posClause-1] < numberTimes ? 1 : 0}
    </td>
```

Código 6.11: Las filas g y h actúan como elementos auxiliares que ayudan a alcanzar la suma de 3. Se añade el valor 1 en la fila g_i si el número de literales que hace verdadera la cláusula es menor que 3. En el caso de la fila h_i, solo añadimos un 1 si es menor que 2, ya que un 1 lo ha proporcionado la fila g_i ya. Nótese que cuando los 3 literales hacen verdadera la cláusula, no hace falta añadir estos unos auxiliares (el valor de g_i y h_i es 0)

El paso final es representar la suma objetivo t. Este dígito se representa en la última fila y está compuesto por una sucesión de unos seguidos por treses, si la fórmula es satisfacible. Si no lo es, la suma de las cláusulas no dará una serie de treses, ya que para aquellas que son falsas, su resultado será 2. Es decir, las filas auxiliares g_i y h_i han aportado un 1 cada una, pero debido a que ningún literal hace verdadera a la cláusula, no se ha podido sumar el último 1, quedándose la suma en 2.

```
        ...
        {/* Final row -> t */}
        <tr>
            <th colSpan='2' className='doubleBorder'>t</th>
```

```
            {/* Row of 1s */}
            {vars.map((_, i) => (
                {/* The left-hand side is always a succession of ones
                   because there is always a truth value for each variable
                   */}
                <td>{1}</td>
            ))}
            {/* Row of 3s */}
            {litClaus.map((_, i) => (
                {/* If at least one literal makes the clause true, then the
                    result of the addition is 3. Otherwise, the result is
                   2, contributed by rows g and h */}
                <td>
                    {countTrues[i] >= 1 ? 3 : 2}
                </td>
            ))}
        </tr>
    </table>
```

Código 6.12: Si la fórmula es satisfacible, el objetivo t se representa mediante una sucesión de unos, seguidos de treses. Si no es satisfacible, en lugar de solo treses, aparecerán algunos doses en esa segunda parte, indicando que no se ha podido satisfacer alguna cláusula

CAPÍTULO

7 Conclusiones y líneas futuras

7.1 Conclusiones

La complejidad computacional es, paradójicamente, una de las áreas más fundamentales y menos valoradas en el grado de Ingeniería Informática. A menudo, los estudiantes tienden a ignorarla debido a su fuerte componente matemático y abstracto, a causa de que es muy distinta de la programación práctica, que se suele encontrar más atractiva. Sin embargo, esta disciplina tiene un papel clave a la hora de diseñar algoritmos eficientes, una tarea que está presente en casi todas las ramas de la informática. Incluso cuando no se es plenamente consciente, al intentar simplificar o mejorar un algoritmo, ya se está operando dentro del ámbito de la complejidad computacional.

El objetivo de esta aplicación ha sido precisamente acercar esta materia a cualquier persona, aunque especialmente a estudiantes, mediante el desarrollo de visualizaciones intuitivas. Para ello, se ha optado por una aplicación web interactiva en lugar de desarrollarla de una manera más tradicional y menos visual. Una presentación visual cuidada es esencial para captar el interés del usuario, especialmente cuando se trata de una materia densa. De haberse descuidado el aspecto visual o la experiencia de usuario, la utilidad didáctica del proyecto se habría visto disminuida. Por tanto, el propósito final ha sido el de enseñar, pero de manera amigable y accesible.

En el proceso de desarrollo, ha sido especialmente útil el estudio de tecnologías web modernas como React, JavaScript, HTML y CSS. También se ha aprendido el modelo de funcionamiento de las single-page applications, la cual, al basarse en la recarga de únicamente los elementos necesarios, mejora significativamente la fluidez de la experiencia del usuario.

Por otro lado, no todo el camino ha sido sencillo. Algunas secciones del proyecto han supuesto un mayor reto técnico e intelectual. Una de las más complejas ha sido el diseño de los mapas de Karnaugh para expresiones con más de cuatro variables, y que deja margen para futuras

optimizaciones. Además, aunque el dibujo de los grafos para representar las reducciones fue menos complejo, también implicó el estudio del uso de la librería D3.js y el trabajo con SVG para crear las estructuras visuales.

Asimismo, el desarrollo de esta herramienta no habría sido posible sin los conocimientos adquiridos en algunas asignaturas de la carrera. En particular, la teoría de Fundamentos de Electrónica permitió la realización de los mapas de Karnaugh para transformar fórmulas booleanas en su Forma Normal Conjuntiva. Del mismo modo, para la realización de los árboles de sintaxis abstracta (AST) en la introducción de fórmulas, se han aplicado conocimientos adquiridos en la asignatura Procesadores de Lenguajes. Sin lugar a dudas, la asignatura Algoritmia y Complejidad ha sido el pilar conceptual del proyecto, proporcionando la mayor parte de la teoría aplicada a lo largo de todo el proceso. Gracias a la orientación y las sugerencias que han aportado los tutores de esta última asignatura, se han corregido y perfeccionado muchos aspectos del proyecto.

El resultado final es una herramienta educativa interactiva que permite visualizar y comprender ejemplos concretos de reducciones de 3SAT a los problemas NP-completos CLIQUE, VERTEX-COVER, HAMPATH y SUBSET-SUM. A través de explicaciones y gráficas, la página pretende no solo facilitar el aprendizaje de este contenido, sino también fomentar el interés por esta disciplina de la informática teórica. Así pues, el fin de esta página es ofrecer un recurso para docentes y estudiantes que cursen asignaturas relacionadas con esta rama de la complejidad computacional.

7.2 Aportaciones adicionales

♦ 7.2.1. Encuesta

Adicionalmente, se diseñó una encuesta con el fin de recoger la opinión de los alumnos sobre la estructura y utilidad de la página. El número de respuestas obtenidas fue reducido, solo obteniéndose 2. Esta escasa participación puede deberse a que la encuesta fue lanzada una vez finalizado el curso. Aun así, presentamos a continuación un breve resumen de los resultados. Asimismo, se incluye el listado de preguntas que contenía la encuesta en el Apéndice de este

documento.

En la pregunta sobre la utilidad de la visualización de los *gadgets* y la claridad de las explicaciones que acompañan a los problemas, todas las valoraciones se situaron entre 9 y 10, es decir, en un nivel sobresaliente.

Respecto a la claridad de la interfaz, la media alcanzada fue de 9, destacando una preferencia por el modo oscuro (9,5) frente al modo claro (9). En general, los participantes señalaron que la interfaz es sencilla y fácil de usar.

En lo relativo a la utilidad global de la página, la calificación fue unánimemente de 10. Los usuarios coinciden en que esta página web habría sido de gran ayuda durante la asignatura, ya que facilita la comprensión de los problemas al ofrecer ejemplos prácticos de forma más abundante y rápida que el texto tradicional.

Para profundizar en estas valoraciones, se preguntó a los alumnos si habían tenido dificultades al comprender esta parte de la materia, siendo 10 equivalente a "Me costó mucho esfuerzo entender esta parte de la asignatura o no llegué a comprenderla por completo". En este caso, se obtuvo un 10 y un 5 como respuestas, lo que, aunque no representa una amplia muestra (solo 2 respuestas), sí permite reflejar dos perfiles distintos de alumnado.

♦ 7.2.2. Licencias GNU GPL v3 y CC BY-SA 4.0

Para el código, se ha optado por la licencia GNU GPL v3 (Licencia Pública General de GNU versión 3), porque garantiza que el código del proyecto permanezca siempre libre y abierto. Esta licencia protege contra intentos de apropiación privativa y obliga a que cualquier modificación o distribución del software conserve las mismas libertades. De esta forma, se asegura que futuros desarrolladores puedan usar, estudiar, modificar y redistribuir el código bajo las mismas condiciones, favoreciendo la colaboración y la mejora continua del proyecto [12].

Para los textos e imágenes que aparecen en la página web, se ha usado la licencia CC BY-SA 4.0, la cual es compatible en un solo sentido (one-way compatibility) con GNU GPL v3. Esto significa que las obras licenciadas con CC BY-SA 4.0 pueden incorporarse en proyectos bajo

GPL v3, pero no es posible tomar un trabajo bajo GPL v3 y relicenciarlo bajo CC BY-SA 4.0 [13].

Para la aprobación de estas licencias, se ha contactado con la OTRI (Oficina de Transferencia de Resultados de Investigación) de la Universidad de Málaga. Esta nos comunicó que nuestra petición había sido aceptada y, a continuación, se procedió a presentarla telemáticamente ante el Registro General de la Universidad de Málaga.

7.3 Líneas futuras

Algunas mejoras que se pueden implementar en un futuro son:

♦ 7.3.1. Mejora del algoritmo de transformación a FNC

En la versión actual del proyecto, se utilizan mapas de Karnaugh para transformar fórmulas booleanas a su forma normal conjuntiva, incluso para un número superior a 4 variables. Si bien esta técnica es un buen método para fórmulas pequeñas de hasta 4 variables, su escalabilidad y eficiencia disminuyen considerablemente conforme crece la complejidad de las expresiones. Como alternativa, podrían explorarse otros algoritmos, como Quine-McCluskey, el cual es adecuado para casos con más variables [14]. Asimismo, el algoritmo implementado actualmente no garantiza la obtención de la FNC mínima, sino que aplica una estrategia voraz cuyo objetivo es obtener una solución razonablemente buena sin necesidad de alcanzar la óptima. La implementación futura de otras técnicas podría mejorar tanto la eficiencia como la calidad de la transformación [15].

♦ 7.3.2. Ampliación con más problemas NP-Completos

Una de las formas más naturales de extender este proyecto es incorporando más problemas NP-completos. Por ejemplo, 3-COLORING, KNAPSACK o SET PACKING son problemas clásicos e interesantes en teoría de la complejidad, y pueden ser reducidos desde 3SAT. La incor-

poración de estos nuevos problemas también puede servir para mostrar distintas estrategias de construcción de gadgets, lo cual permite una experiencia más variada e instructiva para el usuario.

7.3.3. Implementación de heurísticos para la resolución de problemas

Una posible forma de ampliar este proyecto sería el de crear instancias concretas de problemas y resolverlos utilizando algoritmos heurísticos. Esto permitiría al usuario ver cómo, a pesar de que el problema es NP-completo, es posible encontrar soluciones en tiempos razonables gracias a métodos aproximados. Asimismo, se podría comparar la diferencia de tiempo entre el método heurístico y el algoritmo que desea obtener la solución óptima. Así, este enfoque serviría para mostrar conceptos como la diferencia entre soluciones óptimas y aproximadas.

7.3.4. Estimación del tiempo de ejecución del algoritmo de Karnaugh

Otra posible mejora consiste en medir y mostrar el tiempo que tarda el algoritmo voraz usado en los mapas de Karnaugh, en función del número de variables de la fórmula booleana. Implementar una función que estime este tiempo permitiría valorar la eficiencia del algoritmo y advertir al usuario sobre los tiempos de espera cuando se trabaja con fórmulas de mayor tamaño. Además, esta funcionalidad sería muy útil para evaluar alternativas algorítmicas y proporcionar al usuario más información sobre el proceso.

7.3.5. Traducir toda la página al inglés

Debido a que no hay suficiente tiempo para traducir toda la página, sería interesante replantearse esta mejora si el proyecto va a ser utilizado por usuarios que no sepan español. Esta adición aumentaría el alcance del proyecto y lo haría más accesible para un mayor número de personas.

Bibliografía

[1] M. Sipser, *Introduction to the Theory of Computation*. Cengage Learning, Inc, 2012.

[2] S. A. Cook, "An overview of computational complexity," *ACM Turing award lectures*, p. 1982, 2007. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/1283920.1283938

[3] R. C. Martin, "Iterative and incremental development (iid)," *C++ Report*, pp. 1–6, Apr. 1999, engineering Notebook Column. [Online]. Available: https://condor.depaul.edu/~dmumaugh/readings/handouts/SE477/IIDII.pdf

[4] C. Larman and V. R. Basili, "Iterative and incremental development: A brief history," *IEEE Computer*, vol. 36, no. 6, pp. 47–56, Jun. 2003. [Online]. Available: https://personalpages.bradley.edu/~young/CS690S117old/IID.pdf

[5] A. Banks and E. Porcello, *Learning React: Modern Patterns for Developing React Apps*. O'Reilly Media, Inc, 2020.

[6] D. Flanagan, *JavaScript: The Definitive Guide*, 5th ed. O'Reilly Media, Inc, 2006.

[7] S. Springer, *Node.js: The Comprehensive Guide*. Rheinwerk Computing, 2022, pp. 45–47, 53, 61–62.

[8] J. Niederst Robbins, *Learning Web Design: A Beginner's Guide to HTML, CSS, JavaScript, and Web Graphics*, 4th ed. O'Reilly Media, Inc, 2012.

[9] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, pp. 76–91, 2014. [Online]. Available: https://dl.acm.org/doi/pdf/10.5555/2600239

[10] D. Sinha and E. R. Dougherty, *Introduction to Computer-based Imaging Systems*. SPIE Optical Engineering Press, 1998.

[11] S. Sanyal, "Reduction from sat to 3sat," 2018, [Accessed 05-06-2025]. [Online]. Available: https://cse.iitkgp.ac.in/~palash/2018AlgoDesignAnalysis/SAT-3SAT.pdf

[12] B. Smith, "A quick guide to gplv3," 2007, [Accessed 11-09-2025]. [Online]. Available: https://www.gnu.org/licenses/quick-guide-gplv3.pdf

[13] mike, "Cc by-sa 4.0 now one-way compatible with gplv3," 2015, [Accessed 11-09-2025]. [Online]. Available: https://creativecommons.org/2015/10/08/cc-by-sa-4-0-now-one-way-compatible-with-gplv3/

[14] J. Huang, "Programing implementation of the quine-mccluskey method for minimization of boolean expression," National University of Singapore, Tech. Rep., [Accessed 16-06-2025]. [Online]. Available: https://arxiv.org/pdf/1410.1059

[15] H. S. Edith Hemaspaandra, "Proceedings of the twenty-second international joint conference on artificial intelligence," in *Minimization for Generalized Boolean Formulas*, 2011, [Accessed 16-06-2025]. [Online]. Available: https://www.ijcai.org/Proceedings/11/Papers/102.pdf

Apéndice

A.1 Listado preguntas de la encuesta

REDUCIBILIDADES

1. Los gadgets visuales que acompañan a las reducibilidades entre problemas son intuitivos

En CLIQUE

1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○

No entiendo los gadgets visuales / Es difícil usarlos — Es fácil entender los gadgets visuales / Es fácil usarlos

En VERTEX-COVER

1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○

No entiendo los gadgets visuales / Es difícil usarlos — Es fácil entender los gadgets visuales / Es fácil usarlos

En HAMPATH

1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○

No entiendo los gadgets visuales / Es difícil usarlos — Es fácil entender los gadgets visuales / Es fácil usarlos

En SUBSET-SUM

1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○

No entiendo los gadgets visuales / Es difícil usarlos — Es fácil entender los gadgets visuales / Es fácil usarlos

2. Poder visualizar la solución marcada en el gadget me resulta útil

1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○

Muy poco útil — Extremadamente útil

3. Los textos que acompañan a las reducibilidades son claros

En CLIQUE

1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○

No entiendo ninguna de las explicaciones — Entiendo perfectamente todas las explicaciones

En VERTEX-COVER

1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○

No entiendo ninguna de las explicaciones — Entiendo perfectamente todas las explicaciones

En HAMPATH

1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○

No entiendo ninguna de las explicaciones

Entiendo perfectamente todas las explicaciones

En SUBSET-SUM

1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○

No entiendo ninguna de las explicaciones

Entiendo perfectamente todas las explicaciones

4. Expresa cualquier opinión sobre las reducibilidades

INTERFAZ

1. La interfaz es intuitiva y clara

1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○

No me gusta nada la interfaz / diseño de la página

Me gusta mucho la interfaz / diseño de la página

2. Puntuación modo claro

1	2	3	4	5	6	7	8	9	10
◯	◯	◯	◯	◯	◯	◯	◯	◯	◯

No me gusta/Difícil de ver — Me gusta mucho/Es agradable

3. Puntuación modo oscuro

1	2	3	4	5	6	7	8	9	10
◯	◯	◯	◯	◯	◯	◯	◯	◯	◯

No me gusta/Difícil de ver — Me gusta mucho/Es agradable

4. Expresa cualquier opinión sobre la interfaz

UTILIDAD DEL SITIO WEB

1. Me ha resultado útil, ojalá hubiese estado disponible mientras estudiaba

1	2	3	4	5	6	7	8	9	10
◯	◯	◯	◯	◯	◯	◯	◯	◯	◯

No me hubiera ayudado a estudiar — Me hubiera ayudado mucho para aprobar o mejorar mi nota / Hubiera sido imprescindible en mi estudio

2. Me costó mucho esfuerzo entender esta parte de la asignatura o no llegué a comprenderla al completo

1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○

Me resultó muy fácil entender las reducibilidades y estos problemas

Me costó muchísimo esfuerzo entender las reducibilidades y algunos conceptos todavía no los entiendo

3. Expresa cualquier opinión sobre la utilidad

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática

Bulevar Louis Pasteur, 35

Campus de Teatinos

29071 Málaga