BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN

Reparación de la inferencia entre formas conjuntivas implementando un algoritmo de revisión de creencias en PROLOG

TESIS

PARA OBTENER EL GRADO DE:
MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
Reynold Osuna González

ASESOR DE TESIS:
DR. Guillermo De Ita Luna

CO-ASESOR DE TESIS:
DR. Luis Carlos Altamirano Robles

PUEBLA, PUE. Julio 2022

Dedicatoria

Quisiera dedicar mi trabajo de tesis a Vianey, Alejandro y Josué, mi esposa e hijos, a mis padres Guadalupe y Ricardo, a mis hermanos Victoria y Ricardo y en general mi amada familia que por gracia de Dios es extensa pero muy unida, a cada uno de ellos que siempre creen en mí y que tanto me han dado y apoyado.

Reynold Osuna González

Agradecimientos

Gracias a Dios por la vida que me ha concedido, llena de bendiciones y oportunidades, particularmente en este momento agradezco la oportunidad que me ha dado de volver a estudiar temas que me interesan, gustan y apasionan grandemente.

Agradezco de la manera más profunda al Consejo Nacional de Ciencia y Tecnología (CONACYT) por su apoyo económico otorgado mediante la beca para estudios de Maestría ya que hizo posible que cursara y concluyera esta etapa académica de mi vida.

Agradezco a la Benemérita Universidad Autónoma de Puebla y a la Facultad de Ciencias de la Computación por la oportunidad de continuar mi formación académica.

Gracias a Vianey, mi esposa amada por animarme a estudiar esta Maestría y por su amor y continuo apoyo a lo largo de esta y cada etapa que hemos vivido juntos, gracias por ser también un pilar de mi vida, y gracias a mis hijos por su comprensión en el tiempo dedicado a mis estudios, ustedes son otro pilar en mi vida.

Gracias a mis padres por su dedicación y amor, por su esfuerzo y tiempo dado a mi persona desde pequeño y por tantas cosas que sería imposible escribirlas todas. Gracias por ser un gran ejemplo de vida y por ser siempre un gran pilar y al mismo tiempo un faro en mi vida.

Gracias a mis hermanos por su amor y cariño, por siempre apoyarme y por estar presentes en cada momento importante de mi vida.

Gracias a mis asesores el Dr. Guillermo De Ita Luna y el Dr. Luis Carlos Altamirano Robles por su invaluable apoyo en el desarrollo de este trabajo. A cada uno de los profesores con quienes tuve el gusto de tomar clase, gracias por su tiempo y dedicación, de cada uno adquirí conocimientos que considero de gran importancia y ampliaron mi percepción de lo que comprenden las ciencias de la computación.

A mis compañeros de generación también les agradezco su apoyo y participación durante las clases, puesto que estas interacciones enriquecieron grandemente el desarrollo de estos estudios de postgrado.

Sin duda he tenido la gran fortuna de tener excelentes maestros y profesores en cada etapa académica de mi vida, gracias a cada uno de ellos.

Finalmente, gracias a cada miembro de mi familia, a cada uno de mis amigos, no habría espacio suficiente para mencionarlos a todos pero no sería la persona y profesionista que soy sin al apoyo y cariño que cada uno de ustedes me ha dado a lo largo de mi vida.

Resumen

Para las aplicaciones de inteligencia artificial, es sumamente importante tanto la representación del conocimiento de un ser racional como la revisión de creencias, puesto que sin ella la base de conocimiento de dicho ser permanecería inalterable sin posibilidad de crecer o ser corregida. La lógica proposicional permite representar una base de conocimiento y por medio de la inferencia lógica, es posible determinar si la aparente nueva información forma o no parte de la base de conocimiento.

En este proyecto se considera un operador de revisión de creencias entre formas normales conjuntivas (FNC's). El operador que trabaja dentro del marco de la lógica proposicional hace uso de métodos de deducción automática para revisar si una nueva información se deduce lógicamente de una base de conocimiento.

En el caso de que una nueva información no se infiera de la base de conocimiento, se debe determinar lo mínimo a agregar a la base de conocimiento, para que si se cumpla la inferencia. A este proceso le llamamos reparación de la inferencia lógica. En este proyecto de tesis, proponemos un método lógico para reparar la inferencia lógica entre FNC's. La propuesta de método se implementa a través de un sistema programado con el lenguaje de programación lógica: PROLOG.

Finalmente, se ejemplifica el uso del algoritmo implementado en PROLOG dentro de una aplicación sencilla que ayuda a realizar un diagnóstico asistido por computadora para la cepa de COVID-19 que pueda presentar un paciente basado en síntomas, mostrando la construcción de la base de conocimiento necesaria.

Índice General

Índice de figuras

Índice de tablas

1 Introducción

La humanidad ha creado a lo largo de la historia, herramientas y máquinas que le permiten desarrollar sus actividades en formas más fáciles y eficientes. Cuando en el siglo XVII dio comienzo la revolución industrial, se lograron importantes avances en el diseño y construcción de máquinas que permitieron la automatización de trabajos altamente repetitivos. Sin embargo, la mayoría de estas actividades requiere en menor o mayor grado de una supervisión constante por parte de una o más personas para asegurar que realicen correctamente su función, y en caso de cambios en el entorno, revertirlos o ajustar el funcionamiento de la máquina a las nuevas condiciones; es por ello que aparece el deseo de construir máquinas autónomas capaces de adaptarse a condiciones cambiantes y aprender por cuenta propia, si bien el deseo de replicar el pensamiento y la inteligencia humana ha existido por mucho más tiempo.

Con el desarrollo de la computación en forma teórica como una rama de las matemáticas y la posterior invención de las computadoras digitales, se vislumbró la posibilidad de por primera vez crear "máquinas pensantes". Si bien John McCarthy acuñó el término de ***Inteligencia Artificial* (*IA*)** en la década de 1950, durante un taller en Darmouth College[4], desde la década previa se trabajaba ya con la idea de lograr replicar el pensamiento por medio de computadoras y siendo el trabajo de McCulloch y Pitts, ***"A logical calculus of the ideas immanent in nervous activity"*** [4] [15] que es considerado el primer trabajo del campo.

En la actualidad, la ***IA*** es comúnmente asociada con robots en el imaginario colectivo gracias a obras literarias, artísticas y a las recientes aplicaciones que se hacen eco en los medios de comunicación. Es importante hacer notar que la ***IA*** no se enfoca exclusivamente en aplicaciones computacionales, puesto que también se enfoca en el estudio y entendimiento de la naturaleza del pensamiento inteligente, donde las computadoras son los dispositivos que permiten realizar experimentación sobre este campo [14].

Desde su nacimiento, la ***IA*** ha realizado un intenso estudio y desarrollo de nuevos programas informáticos que han llevado a que, en la actualidad, existan numerosas aplicaciones que utilizan la ***IA***, tanto en la industria como en la vida diaria de gran parte de la población mundial. Sin embargo, no existe a la fecha una ***IA*** de amplio espectro (por llamarla de alguna forma) puesto que cada aplicación se concentra en una actividad o en un reducido conjunto de actividades estrechamente relacionadas entre sí.

Para desempeñarse adecuadamente, un ente inteligente debe de alguna forma conocer su entorno o ambiente y ser también capaz de percibir los cambios que en éste se presenten, para poder actuar acorde a ello.

Por lo anterior, en el desarrollo de dichos entes o aplicaciones de la ***IA*** resulta sumamente importante contar con una representación adecuada del conocimiento de un ser o de un agente racional, y disponer de los medios por los cuales el agente adquiera información constantemente de las condiciones del ambiente, más no basta simplemente adquirir información, también es fundamental contar con un mecanismo que permita al agente

modificar lo que conoce, bien sea para corregir o para ampliar la información que conforma su ***base de conocimiento,*** a este mecanismo se le denomina ***revisión de creencias*** y sin ella, el conocimiento del agente permanecería inalterable sin posibilidad de crecer o ser corregido.

El conocimiento representado en una base de conocimiento puede ser modificado de diversas maneras, siendo 3 las principales: expandir la base de conocimiento, contraer la base de conocimiento y realizar una revisión de la base de conocimiento [11].

La ***revisión de creencias*** consiste en incorporar nuevas creencias a una base de conocimiento ya establecida, con cambios mínimos a la base de conocimiento con el objetivo de preservar las creencias originales y mantener su coherencia [2]. Es por medio de la revisión de creencias que los seres racionales son capaces de adquirir nuevo conocimiento o actualizar el que ya poseen. Para ello, se contrasta la información que se recibe contra lo ya conocido, con la finalidad de determinar si lo que se recibe pertenece ya al conocimiento previo o no, y en caso de no estar contenido y de que la nueva información contradiga lo conocido, se concilia o se modifica el conocimiento previo con el objetivo de conservar la consistencia en la denominada base de conocimiento.

Como se mencionó, la ***IA*** busca desarrollar agentes racionales, y dado que el objeto de estudio de la lógica es el razonamiento [17], es natural que ésta sea estudiada y utilizada dentro de este campo. La lógica proposicional es el sistema lógico más simple y básico que existe [17] y en lo que concierne a la representación del conocimiento, es una de las herramientas que nos permite realizarla mediante un conjunto de enunciados o *sentencias* que pueden ser evaluados lógicamente, es decir, pueden ser evaluadas como falsas o verdaderas [2]; este conjunto de sentencias es lo que conforma la ***base de conocimiento***. Las sentencias se encuentran formadas por proposiciones, representadas por letras del alfabeto, que son relacionadas entre sí a través de operadores lógicos cuya función es la de conectar las oraciones.

Pero la lógica proposicional no solo nos proporciona una forma de representar el conocimiento, también nos provee de un mecanismo de deducción, es decir, de un proceso que nos permite derivar información a partir de aquella contenida en la base de conocimiento y garantizar que, si la información de partida es correcta, la información derivada o ***inferida*** de ella, también lo será. Un uso alternativo de la inferencia lógica es el determinar si una información dada, forma o no parte de la base de conocimiento, es decir, verificar si de la base de conocimiento se infiere tal información.

En el caso de que la información no sea inferida de la base de conocimiento, es posible determinar qué hace falta agregar a ésta para que se cumpla la inferencia. A este proceso le llamamos ***reparación de la inferencia lógica***. Es importante destacar que la implicación lógica es todo un reto para el razonamiento automático puesto que se trata de un problema perteneciente a la clase Co-NP completo [10], razón por la que se busca obtener algoritmos que permitan implementarla de forma práctica y poder simular el proceso de razonamiento en seres racionales.

Utilizando un algoritmo de revisión de creencias, es posible realizar la reparación de la inferencia lógica, y para este trabajo se procedió a utilizar el diseñado por los autores: De Ita Luna y Bello López [2], que trabaja sobre cadenas falsificantes de ***Formas Normales Conjuntivas*** (***FNC***'s), siendo estás la forma de representación tanto de la base de conocimiento como de la nueva información.

El algoritmo en cuestión parte del diseño de un operador de revisión de creencias denominado ***reducción de independencia*** entre formas normales conjuntivas, mediante el uso de métodos de deducción automática utilizando la lógica proposicional que se presenta por fórmulas en FNC. Se construyen las asignaciones falsificantes de dichas FNC's mediante cadenas de 0, 1 y *, siendo éstas las denominadas ***cadenas falsificantes*** [2].

En este proyecto de tesis se implementa el algoritmo de revisión de creencias previamente mencionado para reparar la inferencia lógica entre FNC's, automatizada a través de un sistema implementado en el lenguaje de programación lógica PROLOG.

1.1 Planteamiento del Problema

Dadas una ***Base de Conocimiento*** (KB, por su nombre en inglés 'Knowledge Base') inicial en ***Forma Normal Conjuntiva (FNC)*** y una nueva información ϕ también en ***FNC***, determinar si la ***KB*** infiere a la nueva información ϕ, denotado como KB $\models \phi$, y si no fuese el caso, reparar – agregar una mínima información a la KB para que la inferencia se cumpla. Este proceso de revisión y agregación se reconoce como revisión de creencias. Tanto la revisión de la inferencia lógica como la Revisión de Creencias se implementaron mediante un sistema en PROLOG.

1.2 Objetivos generales y específicos

1.2.1 Objetivo general

Implementar en PROLOG un algoritmo para decidir la inferencia lógica proposicional entre formas conjuntivas.

1.2.2 Objetivos específicos

- Determinar si dos formas conjuntivas FA y FB se infieren, esto es: FA $\models$ FB.
- Cuando se cumple la inferencia FA $\models$ FB, el resultado del predicado red(cad_fals(FB),cad_fals(FA)) deberá ser la cadena vacía.
- Cuando no se cumple que FA $\models$ FB, el resultado del predicado red(cad_fals(FB),cad_fals(FA)) deberán ser las cláusulas S que hacen falta en FA para que efectivamente se cumpla que: (S $\cup$ FA) $\models$ FB.

2 Preliminares

2.1 Estado del arte.

El proceso por medio del cual se puede realizar un cambio en las creencias es un tema por demás interesante y ampliamente estudiado, existiendo diversas propuestas sobre cómo puede ser realizado este cambio y habiendo múltiples trabajos y propuestas de operadores con dicho propósito, pero cuando se realiza búsqueda sobre la implementación de dichos operadores la información resulta no ser tan abundante.

No obstante, y si bien no se encontraron muchos trabajo de implementación de operadores, concretamente operadores de revisión de creencias, sí ha sido posible encontrar algunos trabajos que aunque de forma limitada tratan parcialmente sobre la implementación de los operadores de cambios de creencias.

An Implementation of Consistency-Based Multi-Agent Belief Change using ASP. [21]

Este artículo del XXXX de Paul Vicol, James Delgrande y Torsten Schaub gira en torno a trabajos previos de los dos últimos autores sobre el cambio de creencias de un agente en un entorno multi agente donde cada uno adquiere información proveniente de los demás. Para ello recurren a una representación del entorno en forma de grafo, donde los vértices representan a los agentes y las aristas representan comunicaciones entre ellos por medio de las cuales intercambian información. Cada agente como es de esperarse cuenta con una base de creencias que será compartida con los demás agentes de modo que cada uno pueda incorporar a su propia base de creencias tanta información de las creencias de los demás como sea posible conservando su consistencia, llamando a este enfoque Maximización y estando definido a detalle en un artículo previo.

El sistema implementado es denominado Equibel y está compuesto de dos partes: el núcleo de la aplicación es la capa encargada de realizar la operación Maximización de creencias, estando implementada en ASP, mientras que una capa externa programada en Python realiza trabajos de post-procesamiento de la información resultante del cambio de creencias además de proporcionar una interfase interactiva que permite utilizar el núcleo en ASP y realizar experimentación sobre grafos predefinidos.

Si bien se describe brevemente cómo se codifica un grafo en ASP y se especifican los predicados que utilizan para ello, así como aquellos necesarios para realizar la operación de maximización, no se entra en detalles sobre el diseño y construcción de estos predicados ni en las razones elegidas para utilizar dicho lenguaje, enfocándose

más bien en ilustrar cómo se cumple con el proceso establecido por el operador implementado.

Especifican la creación de un módulo *equivel* para Python que permite a los usuarios realizar operaciones de revisión y unificación de creencias por medio de la definición de fórmulas que representan las creencias y su sistema se encarga de crear el grafo necesario para realizar el proceso antes descrito. Para la operación de revisión de creencias recurren a crear un grafo de dos agentes, donde la creencia del primer agente corresponde a la base de conocimiento K y la creencia del segundo agente corresponde a la información ϕ a ser revisada. Aplicando su método de compartir información entre agentes, el segundo agente incorpora en ϕ tanta información de K como sea posible sin perder consistencia, entregando como resultado de la revisión de creencias las creencias finales del segundo agente.

GenC: A Fast Tool for Applications Involving Belief Revision [22]

Los autores Aaron Hunter y John Agapeyev reconocen que si bien la teoría de revisión de creencias es un tema ampliamente estudiado, no se ha puesto énfasis en desarrollar e implementar estas teorías en sistemas prácticos, y que los pocos trabajos de implementación existentes se encuentran sumamente limitados en su enfoque de aplicación además de ser sistemas a los que les toma mucho tiempo entregar soluciones cuando los casos de prueba o trabajo son grandes, debido a que ciertamente, la revisión de creencias es un problema complejo y establecen como desarrollar un solucionador de revisión de creencias en el marco AGM más eficiente que los ya existentes.

Su desarrollo es enfocado en operadores denominados de *Distancia Parametrizada (Parametrised difference o PD),* una generalización del operador Dalal. Este operador realiza una revisión de $K * \phi$ entregando un conjunto de creencias que son satisfechas por todos los modelos que se encuentran mínimamente distantes de los modelos de K usando la distancia de Hamming. Los operadores PD ordenan los símbolos en el vocabulario, estableciendo precedencia a símbolos particulares.

La aplicación ha sido realizada en C++ utilizando la librería OpenMP que permite procesamiento en paralelo. Se permite al usuario mediante una consola de comandos ingresar la información correspondiente a K, a ϕ y como algo opcional un orden de precedencia a las variables involucradas. Alternativamente pueden ser K y ϕ cargados desde un archivo externo que contenga la información en forma de cláusulas en FNC o FND. Para establecer el orden de las variables, el archivo que las contiene debe contener el listado de las mismas en orden descendente en cuanto a la prioridad.

BRL: A Toolkit for Learning How an Agent Performs Belief Revision [23]

Los autores Aaron Hunter y Konstantin Boyarinov plantean en este trabajo que en un entorno multiagente, un agente inteligente no solo debería ser capaz de realizar una

revisión de sus creencias si no observar y aprender de otros agentes cómo éstos han realizado sus propias revisiones de creencias en el pasado, para poder prever como realizarán en el futuro nuevas revisiones, siendo esta la idea central de lo que denominan Belief Revision Learner (BRL).

BRL le permite a un agente identificar o aprender el operador de revisión de creencias utilizado por otro agente al dividir el problema en dos escenarios. El primero escenario consiste en que se conoce por completo las creencias K iniciales del agente, la información ϕ a ser revisada el las creencias K' que resultaron de la revisión, para lo cual se utiliza un enfoque de fuerza bruta aplicando los operadores de revisión de creencias implementados y señalando aquellos cuya salida corresponda exactamente con K'. En este módulo emplean GenC implementado con anterioridad. Si bien no se dan detalles sobre los operadores implementados, se ha recurrido nuevamente a C++ para ello.

Estos tres trabajos a pesar de no ser los únicos explorados, son los que más claramente cuentan con un enfoque en la implementación de operadores, y aunque existen numerosos trabajos con propuestas sobre operadores de revisión de creencias, la mayoría de los trabajos en el campo se enfocan en los aspectos teóricos del proceso y en la definición formal de los operadores, siendo no tan comunes trabajos de implementación de algoritmos u operadores, algo que no deja de llamar la atención dada la importancia que tiene no solo la revisión de creencias si no el cambio de creencias en general.

En aquellos trabajos que incluyen una implementación del operador de cambio de creencias, no se puede observar un común denominador en cuanto a los lenguajes de programación utilizados, siendo tal vez lo más común recurrir a C++ para las implementaciones directas de los algoritmos o algún lenguaje de programación lógica, como ASP utilizado en el primer trabajo reseñado o PROLOG en alguna de sus diversas variantes con base a que las creencias se representan por medio cláusulas lógicas.

2.2 Lógica proposicional.

En las aplicaciones de Inteligencia Artificial (IA) se busca elaborar agentes que sean capaces de razonar en forma similar a como lo hace un ser humano, de forma que se debe comenzar por adquirir un entendimiento sobre cómo se lleva a cabo el proceso de razonamiento en los humanos.

De acuerdo a la Real Academia Española, se entiende por lógica a la “Ciencia que expone las leyes, modos y formas de las proposiciones en relación con su verdad o falsedad”. Puede decirse que la lógica tuvo origen en las leyes de pensamiento desarrolladas por Aristóteles, en su intento por establecer una manera correcta de pensar [4], y que su estudio dentro del campo de la filosofía ha sido continuo a lo largo de la historia, pero su desarrollo matemático fue iniciado por George Boole al definir la lógica proposicional (también conocida como Booleana), en 1847 [4].

Mediante el uso de la lógica proposicional es posible representar conocimiento, utilizando proposiciones para ello, es decir, afirmaciones que pueden ser evaluadas como verdaderas o falsas y que serán representadas por medio de símbolos y conectivas u operadores (lógicos) que expresen relaciones entre los símbolos [5]. Cuando se relacionan símbolos mediante operadores se forman cláusulas, que de igual forma pueden ser evaluadas como verdaderas o falsas, y que a su vez, pueden relacionarse con otros símbolos o cláusulas, y es de esta forma que se construyen nuevas oraciones o creencias.

Como ya se ha dicho, los posibles valores que puede tomar una proposición son sólo dos: verdadero o falso, así que el conjunto de valores posibles es {V,F}. Para evaluar una cláusula, es decir, conocer si a su vez es verdadera o falsa, se realiza asignación de valores para cada símbolo incluido en ella (a esto se le llama una ***interpretación***) [18] y se aplica el valor que corresponde a la relación dictada por el o los conectores lógicos que las relacionan.

Dado que el número de posibles valores que puede tomar cada símbolo, el número de interpretaciones posibles para una cláusula está dada por 2^n , siendo n el número de símbolos que contiene la cláusula. Cuando una asignación vuelve verdadera a la cláusula, se dice que dicha asignación es un ***modelo*** de la cláusula.

Si bien existen muchos operadores lógicos, es posible expresar unos en términos de otros puesto que algunos son redundantes, por lo que es posible utilizar un conjunto reducido de ellos en la construcción de la información (creencias) a representarse mediante lógica proposicional.

Mediante el conjunto de conectores lógicos formado por $\wedge$ (conjunción), $\vee$ (disyunción) y $\neg$ (negación), es posible representar todo enunciado de la lógica proposicional. Los enunciados denotados en Forma Normal Conjuntiva (FNC) se forman de cláusulas relacionadas entre sí por exclusivamente el operador conjunción, y donde la estructura interna de las cláusulas es de literales (una proposición p o la negación de p) relacionadas exclusivamente por el operador disyunción.

Finalmente, utilizando la lógica proposicional es posible también establecer relaciones entre dos sucesos o creencias, según el orden en que éstos sucedan, es decir se establece que una creencia es consecuencia de otra. A este proceso se le conoce como inferencia o implicación lógica. La inferencia lógica (denotada por el operador $\models$) también sirve para corroborar si una cláusula que represente una aparente nueva información (identificada con la letra griega ϕ) debe ser agregada a la base de conocimiento (representada por la letra griega $\mathbf{K}$) o si ya se encuentra contenida en ella de forma explícita, o si puede ser deducida a partir de ella. Este proceso es denominado Revisión de la Inferencia Lógica [5].

Para este trabajo, tanto la base de conocimiento $\mathbf{K}$ como toda nueva información ϕ será representada en lógica proposicional con enunciados expresados únicamente en Forma Normal Conjuntiva.

2.3 Revisión de Creencias.

La base de conocimiento K de un agente racional puede ser modificada de diversas formas cuando se confronta con nueva información, como por ejemplo, K puede ser expandida al agregar nueva información, cuidando que la K resultante sea consistente. K puede ser contraída eliminando de la K original la información inconsistente con la nueva información. O puede llevarse a cabo un proceso de revisión de creencias, en el cual se agregue la nueva información a la K y eliminando la información antigua que provoca la inconsistencia [11].

Cuando se recibe nueva información ϕ que es consistente con K, y que no es parte de K, es suficiente con agregar ϕ. Sin embargo, cuando la nueva información es inconsistente con K se debe aplicar una conciliación que permita agregar el nuevo conocimiento ϕ sin comprometer la consistencia de K. Se denota al proceso de revisión de creencias entre K y ϕ, como: $\kappa * \phi$.

La revisión de creencias busca describir el proceso que debe llevar a cabo un agente racional para agregar nueva información a su base de conocimiento, siempre tratando de minimizar los cambios y cuidando su consistencia, es decir, que el conocimiento existente y el conocimiento a agregar no sean contradictorios [6]. Es ampliamente considerado como una propiedad fundamental de la revisión de creencias al principio de persistencia de Dalal, consisten en que se debe conservar la mayor cantidad de conocimiento previo posible [16].

Alchourron, Gärdenfors y Makinson, en su artículo *On the Logic of Theory Change: Partial Meet Contraction and Revision Functions* [8] establecieron un conjunto de postulados (conocidos como postulados AGM), que se pueden tomar como base para este proceso. Posteriormente Katsuno y Mendelzon, en su artículo *Propositional knowledge base revision and minimal change* [9], reformulan los postulados AGM en términos de fórmulas, que se identifican como postulados KM (Tabla 1).

Postulado KM 1	$\kappa * \phi \vDash \phi$
Postulado KM 2	Si $\kappa \wedge \phi$ es satisfactible, entonces $\kappa \wedge \phi \equiv \kappa * \phi$
Postulado KM 3	Si ϕ es satisfactible, entonces $\kappa * \phi$ también es satisfactible.
Postulado KM 4	Si $\kappa_1 \equiv \kappa_2$ y $\phi_1 \equiv \phi_2$, entonces $\kappa_1 * \phi_1 \equiv \kappa_2 * \phi_2$
Postulado KM 5	$(\kappa * p_1) \wedge p_2 \vDash \kappa * (p_1 \wedge p_2)$
Postulado KM 6	Si $(\kappa * p_1) \wedge p_2$ es satisfactible, entonces $\kappa * (p_1 \wedge p_2) \vDash (\kappa * p_1) \wedge p_2$

Tabla 1. Postulados KM

En estos postulados, el símbolo * representa al proceso de revisión de creencias, cuyo resultado es también una fórmula proposicional.

Existen diversas propuestas de operadores para la revisión de creencias que cumplen con los postulados KM. Sin embargo, García, De Ita y Zacarías presentan en su artículo: *Model-based Algorithm for Belief Revisions between Normal Conjunctive Forms* [7], una propuesta en la cual partiendo de una base de conocimiento K y una nueva información ϕ, ambas en FNC, se aplica un operador llamado de independencia: Ind(ϕ,K), y cuyo resultado es también una FNC. Ind(ϕ,K) permite determinar si la nueva información ϕ ya se encuentra incorporada en K, y si no es así, agregar aquellas cláusulas que le hacen falta de modo que K' contenga a ϕ y sea consistente, siendo K' la base de conocimiento modificada.

2.4 PROLOG.

PROLOG es el lenguaje de programación lógica más utilizado en el área de inteligencia artificial [4]. PROLOG surgió de un proyecto enfocado en el procesamiento de un lenguaje natural, concretamente el lenguaje Francés, en el que fue desarrollada una versión preliminar de PROLOG a principios de los años 70's. Su nombre proviene de la abreviatura de *PROgrammation en LOGique* [3].

El objetivo buscado era realizar deducciones basadas en textos escritos en lenguaje natural, el idioma Francés para ser precisos [3], y en un principio fue ampliamente utilizado para escribir sistemas expertos de diversos ámbitos [4].

Un programa escrito en PROLOG consiste en un conjunto de cláusulas positivas. Las cláusulas pueden ser de dos tipos: reglas y hechos. Una cláusula puede contener un encabezado y un cuerpo cuando son reglas y únicamente un encabezado cuando son hechos. En PROLOG no existen instrucciones de control, ya que la ejecución de un programa está basada en la unificación y en el backtracking.

PROLOG no genera todas las respuestas posibles de un subobjetivo antes de pasar al siguiente, en vez de ello genera una respuesta y la evalúa, con una expectativa de obtener la respuesta cuando ésta haya sido explorada hasta el final, realizando una búsqueda primero en profundidad. Esto sirve como punto de elección, al cual regresar si al finalizar la búsqueda en profundidad no se ha encontrado la solución, entonces elige otra rama a explorar en profundidad en búsqueda de una solución [4].

3 Algoritmo de Revisión de Creencias.

Se considerará como entrada una base de conocimiento K y nueva información ϕ. Gran parte de los algoritmos de revisión de creencias requieren el cálculo del conjunto de modelos de K y de ϕ, lo que es equivalente a calcular Sat(K) y Sat(ϕ), que como se sabe, es un problema NP-completo [1].

En el algoritmo a implementar en este trabajo de tesis (propuesto en De Ita et al. [2]), tanto K como ϕ se representan mediante Formas Normales Conjuntivas (FNC's). El algoritmo, no requiere el cálculo de Sat(K) ni de Sat(ϕ), sino que se trabaja utilizando asignaciones falsificantes tanto para K como para ϕ. Así es posible calcular en tiempo lineal con respecto al tamaño de K y ϕ, los conjuntos Fals(K) y Fals(ϕ), que son utilizados para determinar si efectivamente la inferencia se cumple: si conocemos Fals(K) y Fals(ϕ), entonces (Fals(ϕ) $\subseteq$ Fals(K)) si y solo sí, K $\models \phi$ [1].

De esta forma, si definimos al conjunto S como el resultado de (Fals(ϕ) - Fals(K)), S será igual al conjunto vacío siempre que K $\models \phi$, mientras que si el conjunto S es diferente del conjunto vacío, S contendrá aquellas asignaciones que falsifican a ϕ pero que no falsifican a K ; a partir de estas asignaciones podemos obtener la correspondiente FNC de S, denotada por F_s, siendo Fals(F_s) = S, y tal que (K' = K * ϕ = K $\wedge$ S), y se cumplirá que: K' $\models \phi$ [1].

Para obtener Fals(F_s), se propone la implementación de un nuevo operador lógico, denominado ***Reducción de Independencia***: *Ind(ϕ_i , C_j),* que opera sobre dos cláusulas (una ϕ_i perteneciente a ϕ y C_j perteneciente a K. Este operador permite trabajar en tiempo lineal con respecto al número de variables involucradas y es la base del algoritmo de revisión de creencias.

El operador *Ind(ϕ_i , C_j)* realiza una evaluación de las cláusulas ϕ_i y C_j para generar un resultado, habiendo 3 posibilidades:

a. La cláusula ϕ_i y la cláusula C_j son independientes, en cuyo caso el resultado del operador deberá ser la misma cláusula ϕ_i.
b. La cláusula ϕ_i es en efecto inferida de la cláusula C_j, por lo que el resultado del operador será el conjunto vacío Ø.
c. Para cualquier otro caso, el resultado deberá ser el conjunto S, es decir aquellas asignaciones que falsifican a ϕ_i pero que no falsifican a C_j.

Este deberá ser un proceso iterativo, puesto que al resultado: Sij = *Ind(ϕ_i , C_j)* será la entrada para realizar *Ind(S_{ij} , C_{j+1})* hasta llegar a la cláusula C_n de K. Así mismo, si ϕ está compuesta por m cláusulas, deberá realizarse este proceso para cada una de ellas. Es importante hacer notar que este proceso iterativo puede desencadenar un crecimiento exponencial de cláusulas en F_s puesto que S puede contener más de una cadena o asignación falsificante.

Por ejemplo, sea A la FNC formada por (p ∨ q) ∧ (¬p ∨ r), y B la FNC formada por (p ∨ ¬r), se quiere decidir si FA ⊨ FB, que es el problema central de inferencia proposicional.

Para realizar la inferencia lógica, primero se construyen el conjunto de asignaciones que falsifican tanto a FA como a FB. Tales conjuntos de asignaciones son representados mediante lo que se denomina como cadenas falsificantes. Una cadena falsificante es aquella cuyos valores de las literales vuelven falsa a la cláusula.

Retomando el ejemplo, las literales involucradas en ambas fórmulas son p, q y r, aunque no aparecen todas las literales en cada cláusula. Por lo que se codificarán cadenas de 0,1,* en las posiciones: [p,q,r]. Cuando una literal no aparece en una cláusula, su lugar es ocupado por un asterisco.

Bajo el método propuesto en este trabajo, decidir si FA ⊨ FB se realiza como sigue:

- Se genera la cadena falsificante para B (p ∨ ¬r), que se denomina como cad_fals(FB)= [0,*,1].

- Se generan las cadenas falsificantes para A (p ∨ q) ∧ (¬p ∨ r), que es denominada como cad_fals(FA)= [[0,0,*],[1,*,0]].
- Se aplica el método de reducción de independencia ind(cad_fals(FA),cad_fals(FB)):

- $ind(FB_1,FA_1) = ind([0,*,1],[0,0,*])$

  Se evalúa si [0,*,1] y [0,0,*] son independientes, en este caso no son independientes así que pasa a la siguiente evaluación.

  Se evalúa si [0,*,1] es subsumida por [0,0,*] condición que no se cumple por lo que debe pasar a construir el conjunto S_{11}:

  ind([0,*,1],[0,0,*]) = [0,1,1]

- FB está formado por 2 elementos, por lo que el proceso debe iterarse una segunda vez, ésta vez tomando como parámetros al conjunto S_{11} obtenido y a la cadena falsificante FA_2:

- $ind(S_{11},FA_2) = ind([0,1,1],[1,*,0])$

  Se evalúa si [0,1,1] y [1,*,0] son independientes; en esta evaluación se determina que en efecto son independientes al contar con literales complementadas (la correspondiente a p), por lo que el resultado del operador deberá ser la misma cláusula del conjunto S_{11}. Entonces,

  ind([0,1,1],[1,*,0]) = [0,1,1]

El resultado obtenido por el operador es diferente a la cadena vacía, por lo tanto, se concluye que FA no infiere a FB y la cadena [0,0,1] es la información faltante para que FA infiera a FB. La cadena falsificante [0,0,1] codifica a la cláusula (p ⋁ q ⋁ ¬r), lo que a su vez determina que se cumple que: (p ⋁ q) ⋀ (¬p ⋁ r) ⋀ (p ⋁ q ⋁ ¬r) ⊨ (p ⋁ ¬r). Así, la nueva KB se conformará por: (p ⋁ q) ⋀ (¬p ⋁ r) ⋀ (p ⋁ q ⋁ ¬r).

3.1 Implementación del algoritmo.

El algoritmo base a implementar realiza la revisión de creencias entre FNC's mediante un operador lógico denominado ***Independencia*** que puede construir una o más nuevas FNC's.

La definición de este operador es la siguiente:

$$Ind\ (\phi, \kappa) = \begin{cases} \phi & Si\ \phi\ y\ \kappa\ son\ independientes \\ conjunto\ vacío & Si\ \phi\ es\ subsumida\ por\ \kappa \\ conjunto\ de\ cláusulas\ independientes & En\ otro\ caso \end{cases}$$

El operador Ind, se basa en el trabajo del operador **Independencia** $Ind(\phi_i\ ,\ C_j)$ sobre dos cláusulas, revisando primero si la cláusula ϕ_i correspondiente a la nueva información y la cláusula C_j correspondiente a la base de conocimiento son independientes, significando esto que ninguna parte del nuevo conocimiento es inferido del conocimiento previo por lo que se debería agregar completamente la nueva información a la base de conocimiento, resultando entonces la cláusula ϕ_i sin modificación alguna.

Si ambas cláusulas no son independientes, el siguiente paso es revisar si la cláusula ϕ_i es subsumida por C_j, lo que significaría que en efecto la inferencia se cumple y no requiere ser reparada, dando como resultado un conjunto vacío.

Finalmente, cuando no se trata de ninguno de los casos previos, lo que procede es construir un conjunto de cláusulas independientes de KB (mediante un segundo operador denominado ***reducción de independencia***) que corresponderían a las cláusulas faltantes en KB para que la inferencia sea efectivamente cumplida.

Como se observa, el operador descrito trabaja sobre dos cláusulas, la primera perteneciente a la nueva información y la segunda a la base de conocimiento, obteniendo como resultado un conjunto, que puede ser vacío o en caso de no serlo, contiene las cláusulas que hacen falta a KB para reparar la inferencia. Se debe notar que cuando la KB está compuesta por más de una cláusula, no es posible definir si es necesario agregarle las cláusulas resultantes para reparar la inferencia con solo aplicar el operador entre dos cláusulas puesto que cabe la posibilidad que las cláusulas obtenidas si puedan ser inferidas por el resto de la información conocida. Así que a las cláusulas del conjunto resultante se les debe aplicar el mismo operador con la siguiente cláusula perteneciente a la base de conocimiento, proceso que debe

repetirse hasta haber alcanzado la última cláusula de la base de conocimiento. Es en este punto, que se puede determinar cuál es el conjunto de cláusulas resultante que son necesarias para reparar la inferencia.

A su vez, la información ϕ puede estar compuesta por más de una cláusula, así que el proceso anteriormente descrito debe repetirse sobre cada cláusula de ϕ.

Si al finalizar este proceso, el resultado es el conjunto vacío, no es necesario reparar la inferencia, mientras que si se obtienen cláusulas nuevas, éstas deben ser adicionadas a KB para reparar la inferencia lógica.

4 Predicados en PROLOG

En principio, el planteamiento del proceso a realizar parecería claro y que requeriría pocos predicados para ser llevado a cabo:

1. Un predicado que obtuviese las cadenas falsificantes.
2. Un predicado que revisase la independencia entre dos cláusulas.
3. Un predicado que permitiera aplicar la reducción de independencia entre dos cláusulas.
4. Un predicado que adicionara las cláusulas obtenidas (si alguna) a la base de conocimiento para reparar la inferencia.

Sin embargo, al comenzar la implementación del algoritmo elegido, se observó que la principal dificultad que entraña es que el aplicar el operador independencia entre dos cláusulas no arroja siempre como resultado una única cláusula, además de que cuando la información Φ ó KB está formada por más de una cláusula, se debe "encadenar" el uso del operador de forma que se aplique a todas las cláusulas involucradas, incluyendo las que resultan generadas durante el proceso y no es posible conocer a priori el número de cláusulas que arrojará como resultado final.

4.1.1 Cadenas Falsificantes

Como se mencionó previamente, el operador Independencia trabaja sobre asignaciones o cadenas falsificantes de las FNC's que conforman tanto a Φ como a la KB, por lo que el primer predicado a implementar es aquel que reciba como entrada una cadena que represente a la FNC y devuelva su cadena falsificante. Las FNC se codifican por medio de una lista que puede contener únicamente los valores 1 para literales positivos, 0 para literales negativos y * como un comodín que representa aquellas literales no presentes en la cláusula. Una cadena falsificante invierte el signo de las proposiciones en una cláusula, 0 para lietrales positivas, 1 para literales negativas, y * para variables proposicionales que no aparezcan en la cláusula. El comodín * indica que no es relevante para la evaluación de la cláusula el valor que una literal tome, puesto que no está contenida en la FNC, así que en la cadena falsificante permanecerá representado por el mismo comodín. Así, para obtener las cadenas falsificantes se definió el predicado ***comp***:

```
%comp(Clausula,CadenaFalsificante) Forma el vector complemento de una asignación
comp([],[]).
comp([1|L],[0|L1]) :- comp(L,L1),!.
comp([0|L],[1|L1]) :- comp(L,L1),!.
comp([*|L],[*|L1]) :- comp(L,L1),!.
```

En ***comp(Clausula,CadenaFalsificante)*** se recorrerá la lista *Clausula* hasta su último elemento, mientras se coloca en la cláusula *CadenaFalsificante* el valor complementado correspondiente, una vez que se llega al final de Clausula termina la ejecución del predicado.

4.1.2 Independencia entre cadenas

Una vez obtenidas las cadenas falsificantes, debemos contar con un predicado que determine si dos de ellas son independientes, para ello se construyó ***indep***:

```
%indep(ClausulaA,ClausulaB)
indep([],[]) :- !,fail.
indep([0|_],[1|_]) :- true,!.
indep([1|_],[0|_]) :- true,!.
indep([_|L],[_|L1]) :- indep(L,L1).
```

Dos cadenas son independientes entre sí, cuando en ambas está presente al menos un mismo literal en forma complementaria, por lo que el predicado ***indep(ClausulaA,ClausulaB)*** debe comparar el valor contenido en iguales posiciones de las dos listas o cadenas falsificantes en busca de esta complementación: si la encuentra, en cualquier posición, termina ejecución y devuelve el valor verdadero, mientras que si no lo encuentra las recorrerá hasta el final, en cuyo caso devuelve el valor falso.

4.1.3 Subsumir

Cuando las cadenas no son independientes, existe la posibilidad que la cláusula ϕ_i sea subsumida por la cláusula C_j, por lo que ahora requerimos un predicado que evalúe esta posibilidad sobre las cadenas, lo que se hace con el predicado ***subsume***:

```
% subsume(A,B)
subsume([],[]) :- true,!.
subsume([X|L],[X|L1]) :- subsume(L,L1),!.
subsume([_|L],[*|L1]) :- subsume(L,L1),!.
subsume([1|_],[0|_]):- fail.
subsume([0|_],[1|_]):- fail.
```

Si A es subconjunto de B, entonces B subsume a A. Esto lo podemos revisar de la siguiente forma: si para cada posición de A, B cuenta con el mismo valor o si en A existe valor (0,1) y en B existe *, B subsume a A. Este predicado nos devolverá nuevamente valores de verdadero o falso al final de la ejecución.

4.1.4 Operador Independencia entre cláusulas.

Cuando no se cumple que dos cláusulas sean independientes, ni la segunda subsuma a la primera, se debe construir como resultado del operador de independencia un conjunto de

cadenas independientes. Pero, ¿cuántas cláusulas independientes pueden y deben obtenerse? La respuesta reside en aquellas posiciones de la cláusula ϕ_i que contengan el carácter comodín * y que en la cláusula C_j contengan ya sea un 1 ó un 0. Por cada posición en donde se presente esta situación, se generará una cláusula independiente, es decir que se debe identificar el número de literales no contenidas en ϕ_i y que sí están contenidas en C_j, y este será el número de cláusulas independientes resultantes. Esto nos lleva a requerir un predicado que obtenga este número, al que llamamos ***diferencia***:

```
% diferencia(A,B,N)
diferencia([],[],0).
diferencia([1|L],[_|L1],N) :- diferencia(L,L1,N).
diferencia([0|L],[_|L1],N) :- diferencia(L,L1,N).
diferencia([*|L],[*|L1],N) :- diferencia(L,L1,N).
diferencia([*|L],[0|L1],N) :- diferencia(L,L1,N1), N is N1 + 1.
diferencia([*|L],[1|L1],N) :- diferencia(L,L1,N1), N is N1 + 1.
```

El predicado diferencia(A,B,N) recorre las listas de entrada A y B comparando los elementos en iguales posiciones. Cuando A contiene un valor distinto de *, sólo avanza en la lista haciendo una llamada recursiva a sí misma, con el resto de ambas listas y sin incrementar el valor N. Cuando encuentra que la posición correspondiente de A contiene el comodín y en B se encuentra 1, o 0, incrementa el valor de N y hace una llamada recursiva a sí misma. De esta forma obtenemos el número de cláusulas independientes a generar.

Ahora surge otra pregunta, ¿cómo se construyen las cláusulas independientes?. Primero se debe comparar cada posición de ambas cadenas y aplicar el siguiente proceso:

- Cuando en ambas cadenas, se tiene el mismo valor, en las nuevas cadenas, se conserva dicho valor (ya sea 1, 0 o *).

- Cuando en la cadena A exista un asterisco y en B exista un valor, la cadena hasta ese momento construida se duplicará.

- En la primer cadena, la posición correspondiente al asterisco encontrado en A, contendrá el valor complementado de la posición correspondiente en la cláusula B (haciéndola con esto, independiente de B) y se completa la cadena con los valores restantes de la cadena A.

- Para la segunda cadena generada, se colocará el mismo valor contenido en B, completando la cadena con el resto de A y se continuará evaluando posición a posición la diferencia o coincidencia de valores entre el resto de A y el resto de B. Esto quiere decir, por cada asterisco en A, donde B contenga 1 o 0, se generan dos cadenas.

Para realizar el proceso descrito, se diseñó el predicado ***construir3***:

```
%construir3(A,B,C,RestoA,RestoB).
construir3([],[],[],[],[]).

construir3([1|L],[1|L1],C,RA,RB):- construir3(L,L1,C1,RA1,RB1),
        La = [1|C1], C = La,
        RA = [1|RA1],
        RB = [1|RB1].

construir3([0|L],[0|L1],C,RA,RB):- construir3(L,L1,C1,RA1,RB1),
        La = [0|C1], C = La,
        RA = [0|RA1],
        RB = [0|RB1].

construir3([1|L],[*|L1],C,RA,RB):- construir3(L,L1,C1,RA1,RB1),
        La = [1|C1], C = La,
        RA = [1|RA1],
        RB = [*|RB1].

construir3([0|L],[*|L1],C,RA,RB):- construir3(L,L1,C1,RA1,RB1),
        La = [0|C1], C = La,
        RA = [0|RA1],
        RB = [*|RB1].

construir3([*|L],[*|L1],C,RA,RB):- construir3(L,L1,C1,RA1,RB1),
        La = [*|C1], C = La,
        RA = [*|RA1],
        RB = [*|RB1].

construir3([*|L],[0|L1],C,RA,RB):- concatenar([1],L,C),
        RA = [0|L],
        RB = [0|L1],!.

construir3([*|L],[1|L1],C,RA,RB):- concatenar([0],L,C),
        RA = [1|L],
        RB = [1|L1],!.
```

Siguiendo el procedimiento, el predicado ***construir3(A,B,C,RestoA,RestoB)*** trabaja comparando posición por posición las cadenas A y B al tiempo que construye la cadena independiente de forma duplicada (argumentos La y La1) y se llama de forma recursiva a sí mismo con el objetivo de continuar comparando cada posición de las cadenas.

Cuando encuentra la condición de "bifurcación", complementa el valor correspondiente de B en la cadena en construcción y copia en ella el resto de la cadena A (los valores aún no comparados de A con B) en la cláusula independiente generada y la entrega en el argumento C. Al mismo tiempo, termina la construcción de la segunda cadena insertando el mismo valor contenido en la posición correspondiente de B y copiando el resto de la cadena A. Esto nos

arroja una cadena independiente de B y una cadena no independiente de B, a la que se debe aplicar nuevamente el predicado ***construir3***.

Es por esto, que se requiere un nuevo predicado que haga llamados a construir3 de forma iterativa. ¿Cuántas veces se debe iterar el predicado? Justamente el número obtenido mediante el predicado ***diferencia***. El predicado en cuestión se llama ***itera_construir3***:

```
% itera_construir3(X,A,B,Clausulas_Phi)
itera_construir3(0,_,_,[]).

itera_construir3(X,A,B,Cs_Phi) :- X1 is X-1,
        construir3(A,B,C,RA,RB), itera_construir3(X1,RA,RB,Cs_Phi1),
        L = [C|Cs_Phi1], Cs_Phi = L.
```

Este predicado tiene por argumentos ***X***, que es el número de veces a iterar el predicado, cláusulas ***A*** y ***B*** y finalmente el argumento ***Cs_phi***, que corresponde a una lista donde se van almacenando las cláusulas independientes construidas al iterar ***construir3***.

Se han definido hasta ahora los predicados que nos permiten construir el resultado del operador independencia: ***indep(ClausulaA,ClausulaB)***, ***subsume(A,B)*** e ***itera_construir3(X,A,B,Clausulas_Phi)*** y los predicados auxiliares para éste último, por lo que podemos construir ahora un predicado que haga uso de ellos para obtener el resultado del operador independencia entre dos cláusulas, ***red_indep_clausula***:

```
% red_indep_clausula(A,B,Resultado)
red_indep_clausula(A,_,Resultado) :- es_vacia(A), Resultado = A.

red_indep_clausula(A,B,Resultado) :- subsume(A,B), longitud(A,Long),
        cadena_vacia(Long,Resultado).
red_indep_clausula(A,B,Resultado) :- indep(A,B), Resultado = A.

red_indep_clausula(A,B,Resultado) :- diferencia(A,B,X), X > 1,
        itera_construir3(X,A,B,Resultado).

red_indep_clausula(A,B,Resultado) :- diferencia(A,B,X), X = 1,
        construir3(A,B,Resultado,_,_).
```

Este predicado sigue la definición del operador y entrega un resultado dependiendo de qué situación se presente:

- Si A es subsumida por B, devolverá la cadena vacía
- Si A es independiente de B, devolverá la misma cadena A
- En otro caso, aplicará la construcción de cláusulas independientes de B y devolverá el conjunto S generado.

Una vez construido el predicado ***red_indep_clausula(A,B,Resultado),*** se pudo apreciar un problema en el aspecto práctico del mismo. Cuando la cláusula de entrada A corresponde a la cadena vacía (representada por una cadena que únicamente contiene asteriscos) no se considera subsumida por B ni independiente de B, por lo que el predicado entregaba un conjunto de cláusulas independientes de B.

Para corregir este problema, se creó un predicado auxiliar: ***es_vacia(A):***

```
%es_vacia(A)

es_vacia([]) :- !, true.

es_vacia([*|L]) :- es_vacia(L).
```

Es una cláusula de funcionamiento muy sencillo puesto que recorre cada posición de la cadena de entrada y valida si está compuesta por sólo asteriscos, que es la representación de la cláusula vacía. Utilizando dicho predicado, se le agregó un cuarto caso a ***red_indep_clausula***, en el que simplemente nos devuelve la misma cadena vacía cuando A sea vacía.

4.1.5 Construyendo el conjunto S

Hasta ahora, los predicados construidos nos permiten aplicar el operador de independencia entre una única cláusula de ϕ y una única cláusula de *K,* con lo que se obtendría un conjunto S correspondiente exclusivamente la revisión de si la cláusula ϕ evaluada se infiere o no de la cláusula *K,* que sería la respuesta final si una base de conocimiento y la nueva información se componen exclusivamente de una cláusula cada una. Sin embargo, una base de conocimiento difícilmente está compuesta por una única cláusula, y la información nueva que se recibe tampoco está restringida a ser de cláusula única, ¿Qué sucede si ϕ y la KB, o ambas, están formadas por más de una cláusula?

Cuando el resultado de ***red_indep_clausula*** entre una cláusula ϕ y la primera cláusula de KB, es un conjunto que contiene más de una cláusula independiente, se debe aplicar el operador independencia entre esas cláusulas obtenidas y las cláusulas restantes de la KB. Así mismo, cuando ϕ está formada por más de una cláusula, a cada una de ellas se les debe aplicar el operador independencia con respecto a las cláusulas de KB (en esto consiste el proceso iterativo descrito en el algoritmo).

Con el objetivo de cubrir este escenario, se recurre al predicado ***itera_red_idep2***:

```
%itera_red_indep2(A,B,Cadenas_Resultantes)

itera_red_indep2(A,B,Respuesta) :- es_cadena(A),es_cadena(B),
        red_indep_clausula(A,B,Respuesta),!.
```

```
itera_red_indep2(A,[X|[]],Respuesta) :-
        es_cadena(A),not(es_cadena([X|[]])),
        red_indep_clausula(A,X,Respuesta).
itera_red_indep2(A,[X|B],Respuesta) :-
        es_cadena(A),not(es_cadena([X|B])),
        red_indep_clausula(A,X,Phi),itera_red_indep2(Phi,B,Respuesta),
        print(Respuesta).

itera_red_indep2([Y|[]],B,Respuesta) :-
        not(es_cadena([Y|[]])),es_cadena(B),
        red_indep_clausula(Y,B,Respuesta).
itera_red_indep2([Y|A],B,Respuesta) :-
        not(es_cadena([Y|A])),es_cadena(B),
        itera_red_indep2(Y,B,PHI),
        itera_red_indep2(A,B,Resultado),
        L = [PHI|Resultado], Respuesta = L.

itera_red_indep2([Y|A],B,Respuesta) :-
        not(es_cadena([Y|A])),
        not(es_cadena(B)),
        itera_red_indep2(Y,B,Respuesta1),
        itera_red_indep2(A,B,Respuesta2),
        L = [Respuesta1|Respuesta2],
        Respuesta = L.
itera_red_indep2([Y|[]],B,Respuesta) :-
        not(es_cadena([Y|[]])),
        not(es_cadena(B)),
        itera_red_indep2(Y,B,Respuesta),
        print(Respuesta).

itera_red_indep2([],_,[]) :- true.
```

Podemos entonces identificar 4 escenarios, en función de los cuales el predicado debe llevar a cabo un proceso diferente:

- Cuando los argumentos A y B son cláusulas únicas.
- Cuando el argumento A es cláusula única y B está formado por más de una cláusula.
- Cuando el argumento A contiene más de una cláusula y B contiene sólo una cláusula.
- Cuando los argumentos A y B contienen más de una cláusula cada uno.

Así, este predicado primero revisa si los argumentos de entrada, A y B, corresponden a dos únicas cláusulas. Para ello hace uso del predicado auxiliar es_cadena(A), que revisa si el argumento de entrada es una cadena de valores, o una lista de listas. Si se cumple que ambos argumentos sean cadenas, se aplica directamente el predicado red_indep_clausula a ambas, entregando un resultado en el argumento Respuesta.

Cuando se encuentre que A es una única cláusula, pero B esté compuesto por una lista de cláusulas, se hace un llamado a red_indep_clausula entre la cláusula A y la primera cláusula contenida en B, almacenando su resultado en un argumento Phi, posteriormente llamando en forma recursiva a itera_red_indep2, usando como argumentos las cláusulas contenidas en ϕ y las cláusulas restantes de B. Esta recursión termina en el momento en que se ha alcanzado la última cláusula de B, momento en que se llama directamente a red_indep_clausula entre 2 únicas cláusulas.

Tenemos un tercer caso, que es cuando el argumento A no es una única cláusula, pero B sí lo es y se sigue un procedimiento muy similar al caso previo pero con la diferencia de hacer dos llamados recursivos a itera_red_indep2: el primero corresponde a utilizar el primer elemento del argumento A con la cláusula única B, y almacenar su resultado en Phi, y llamarse a si misma utilizando como argumentos el resto de elementos de A con la única cláusula B. Los resultados de ambos llamados se unificarán en una única lista conteniendo las cláusulas resultantes.

El siguiente caso del predicado corresponde a cuando ambos argumentos, A y B, son un conjunto de cláusulas (ambos son listas de listas). Esto nos lleva a hacer llamadas recursivas de itera_red_indep2: el primer llamado es con el primer elemento de la lista A con toda la lista B, mientras que el segundo llamado corresponde al resto de la lista A con toda la lista B, construyendo una lista de listas donde se almacenan los resultados obtenidos.

Finalmente cerramos el predicado con los casos base, con los cuales sale de la recursión, es decir, cuando se han recorrido los argumentos y solo tenemos listas vacías.

4.1.6 Reparación de la inferencia

Con los predicados implementados al momento, es posible obtener las cláusulas que hacen falta en la *KB* para que la inferencia entre *KB* y Φ sea correcta, es decir, sea reparada la inferencia, no obstante, debido al manejo de listas que implementa PROLOG, emerge un desafío que resulta particularmente complicado de superar directamente.

Si bien se cuenta con un predicado dedicado a concatenar listas, que sirve tanto para generar las cadenas falsificantes, como para agregar nuevas cláusulas a una lista de listas, esto entraña un problema en cuanto al formato en que se puede obtener el conjunto de cláusulas: cuando se obtiene un conjunto de cláusulas a las que se debe seguir aplicando el operador de independencia, cada una de ellas puede resultar a su vez en un nuevo conjunto de cláusulas, que se van adicionando a una lista general que almacena todas ellas una vez terminado el proceso, generando una lista con listas anidadas.

Esto se ilustra con la tabla 2 que muestra la construcción del conjunto S correspondiente a realizar la reducción de independencia entre una KB compuesta de 5 cláusulas y un ϕ compuesta por una única cláusula y de la figura 1 correspondiente a la captura de pantalla del resultado obtenido por el predicado ***itera_red_indep*** de las mismas KB y ϕ, que obtiene

de forma correcta las cláusulas [1,0,0,0] y [1,1,0,0] (la cadena [*,*,*,*] es vacía) aunque en forma de una lista de listas anidadas:

	KB K					
ϕ	* 1 * 1	1 * 1 0	* 0 0 1	0 * * *	1 0 1 1	***S***
1 * * *	1 0 * *	1 0 0 *	1 0 0 0	1 0 0 0	1 0 0 0	**1 0 0 0**
		1 0 1 1	1 0 1 1	1 0 1 1	* * * *	* * * *
	1 1 * 0	1 1 0 0	1 1 0 0	1 1 0 0	1 1 0 0	**1 1 0 0**

Tabla 2. Generación del conjunto S

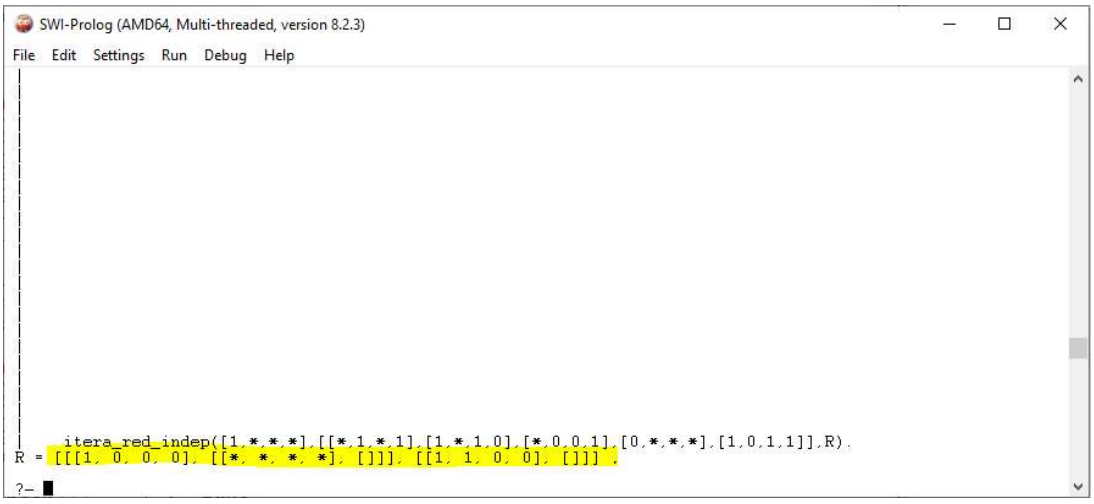

Figura 1. Lista anidada. El mismo conjunto S de la Tabla 2 obtenido mediante itera_red_indep en PROLOG. En amarillo se resalta que las cadenas son las necesarias, sin embargo, se encuentran en una lista de listas (anidada).

Obtener las cláusulas en una lista anidada es inconveniente debido a que para reparar la inferencia lógica es necesario agregarlas a KB por lo que se requeriría un procesamiento adicional para extraer cada una, eliminar del resultado las cadenas vacías y ya que se trata de un conjunto, eliminar cualquier elemento (cadena) que pudiese estar duplicada, antes de poder agregar a KB y concluir con la reparación de la inferencia.

Un enfoque alternativo consiste en hacer uso de un archivo temporal de texto en el que cada cláusula del conjunto S sea escrita conforme se vayan generando y una vez concluido el proceso de reducción de independencia, leer directamente de este archivo las cláusulas que deben agregarse a KB reparando así la inferencia, siendo esta la solución ocupada.

Para realizar este proceso se requirieron predicados cuyas funciones son crear el archivo temporal de texto ***crear_archivo***:

```
% Crear archivo de texto
crea_archivo :- open('C:/temp_prolog1/cadenas1.txt',write,Stream),
                close(Stream).
```

Un predicado que escriba la cláusula en el archivo temporal ***escribe_en_archivo***:

```
% Escribe el término C en archivo temporal cadenas1.txt
escribe_en_archivo(C) :- open('C:/temp_prolog1/cadenas1.txt',append,Stream),
                         write(Stream,' '), write(Stream,C), write(Stream,'.'),
                         close(Stream).
```

Un predicado que sea capaz de leer cada cláusula del archivo temporal y las coloque dentro de una lista no anidada ***leer_archivo***:

```
leer_archivo(Stream,[]) :- at_end_of_stream(Stream).

leer_archivo(Stream,[X|L]) :-  read(Stream,X), print(Stream),
                               leer_archivo(Stream,L).
```

También se requiere un predicado que lea las cláusulas contenidas en el archivo temporal cadenas1.txt ***leer_resultado*** y coloca la información en un término ***Respuesta***:

```
leer_resultado(Respuesta) :-
                open('C:/temp_prolog1/cadenas1.txt',read,Stream),
                leer_archivo(Stream,Lineas),
                close(Stream), print('Cerramos Stream'),
                Respuesta = Lineas,
                write(Lineas),write(Respuesta),nl.
```

Ahora que contamos con una forma de obtener el conjunto S en una lista no anidada, se define el predicado **correr_reduccion**, que se encargará de hacer uso de todos los predicados hasta ahora definidos para justamente generar dicho conjunto S, además de que remueve cualquier cadena formada únicamente por asteriscos y aquellas cadenas que pudiesen repetirse:

```
correr_reduccion(A,B,Resultado) :- exists_directory('c:/temp_prolog1'),
                exists_file('C:/temp_prolog1/cadenas1.txt'),
                delete_file('C:/temp_prolog1/cadenas1.txt'),
                itera_red_indep(A,B,_),
                leer_resultado(Resultado2),
                depurar_respuesta(Resultado2,Resultado),
                print(Resultado).
```

```
correr_reduccion(A,B,Resultado) :- exists_directory('c:/temp_prolog1'),
                not(exists_file('C:/temp_prolog1/cadenas1.txt')),
                itera_red_indep(A,B,_),
                leer_resultado(Resultado2),
                depurar_respuesta(Resultado2,Resultado),
                print(Resultado).

correr_reduccion(A,B,Resultado) :- not(exists_directory('c:/temp_prolog1')),
                make_directory('c:/temp_prolog1'),itera_red_indep(A,B,_),
                leer_resultado(Resultado2),
                depurar_respuesta(Resultado2,Resultado), print('Salida final: '),
                print(Resultado).
```

Por último, se define el predicado ***reparar_base*** con el que obtendremos la KB reparada; este predicado complementa primero tanto a ϕ como a KB y hace el llamado al proceso de correr la reducción, unifica los conjuntos KB y S y a este conjunto KB' le aplica nuevamente una complementación, de esta manera se repara efectivamente la inferencia lógica, de tal forma que KB' $\models \phi$:

```
reparar_base(A,B,Resultado) :-
                complementar(A,Acomp), complementar(B,Bcomp),
                not(es_cadena(B)),
                correr_reduccion(Acomp,Bcomp,Resultado1),
                union_conjuntos(Bcomp,Resultado1,Resultado2),
                complementar(Resultado2,Resultado),
                print('La Kb reparada es: '),
                print(Resultado), escribe_salida(Resultado).

reparar_base(A,B,Resultado) :-
                complementar(A,Acomp), complementar(B,Bcomp),
                es_cadena(B),
                correr_reduccion(Acomp,Bcomp,Resultado1),
                union_conjuntos([Bcomp],Resultado1,Resultado2),
                complementar(Resultado2,Resultado),
                print('La Kb reparada es: '),
                print(Resultado), escribe_salida(Resultado).
```

4.2 Interfase Gráfica de Usuario (GUI)

Habiendo ya desarrollado en PROLOG los predicados necesarios para llevar a cabo la reducción de independencia con éxito, se procedió a generar una Interfase Gráfica de Usuario (GUI, por sus siglas en inglés). Existen múltiples herramientas para este propósito y la elección de una en particular dependió única y exclusivamente de la familiaridad previa de quien desarrolló este trabajo con el lenguaje de programación Turbo Pascal, de modo que la

GUI se realizó utilizando el entorno integrado de desarrollo (o IDE por sus siglas en inglés) llamado Lazarus.

4.2.1 Lazarus

Lazarus es un IDE que utiliza Free Pascal, un compilador open source para el lenguaje de programación de alto nivel Pascal. Es una plataforma clónica del IDE Delphi que es la evolución de Turbo Pascal, un lenguaje de programación de alto nivel, perteneciente a la empresa Embarcadero Technologies quien adquirió los derechos de su desarrollador Borland Software Corporation durante la década de 1990.

Lazarus cuenta con un entorno de desarrollo visual que permite diseñar fácilmente aplicaciones GUI mediante el uso de controles (botones, cuadros de texto, etc.) que deben ser agregados en formularios. Posteriormente, se escribe el código correspondiente a los eventos (como hacer click en un botón) que puedan tener los controles. Contar con un entorno de desarrollo visual libera al desarrollador de escribir el código correspondiente a los elementos visuales y enfocarse exclusivamente en la funcionalidad de la aplicación.

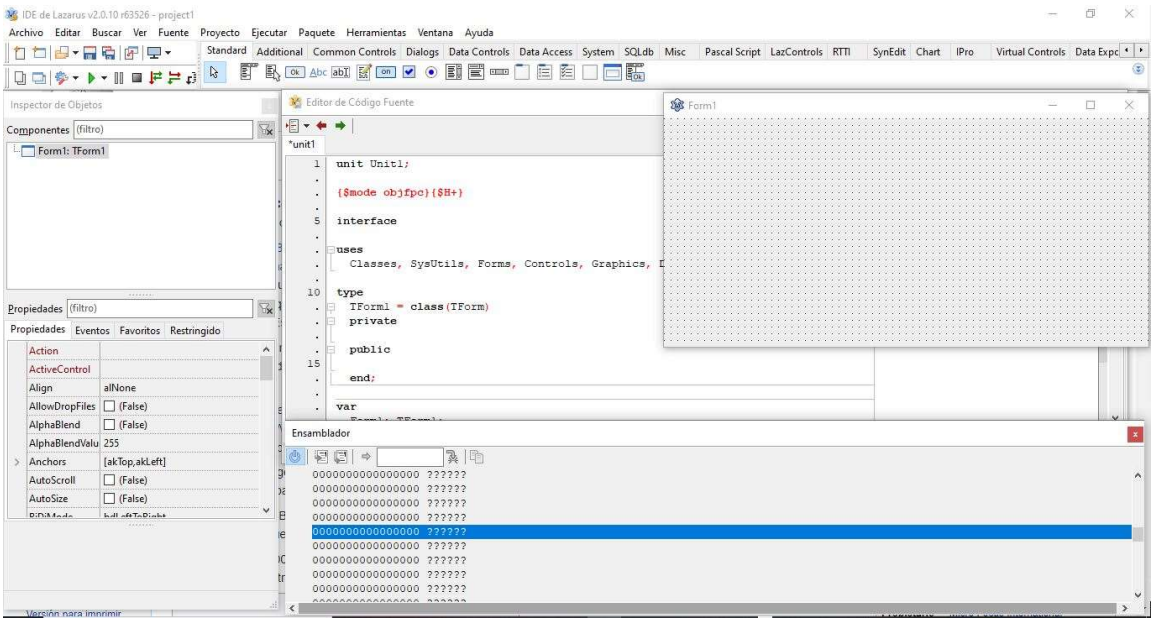

Figura 2. Entorno Integrado de Desarrollo Lazarus

4.2.2 Uso de la GUI

La función de la GUI es proveer una interacción sencilla entre el usuario y el código desarrollado en PROLOG para realizar la reparación de la inferencia por medio de la reducción de independencia. Así, la GUI cuenta con un único menú que permite cargar tanto la base de conocimiento como la nueva información a través de archivos de texto previamente creados.

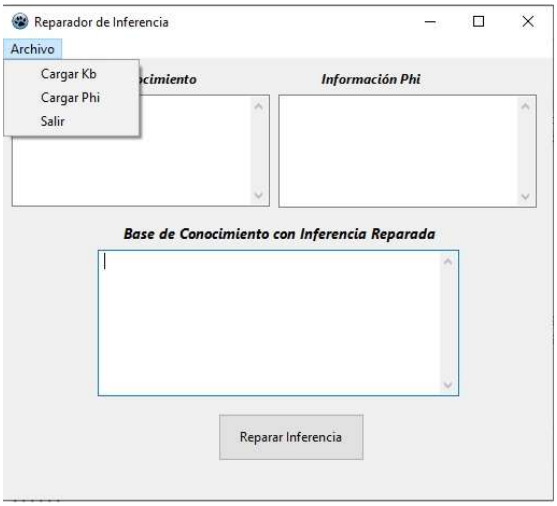

Figura 3. GUI creada para reparar la inferencia lógica.

Cuando se carga un archivo de texto, el contenido de éste se observará en la ventana correspondiente. Posteriormente basta oprimir el botón "Reparar Inferencia" para que se ejecute el predicado reparar_base en PROLOG y se obtenga la KB que infiera efectivamente la información ϕ.

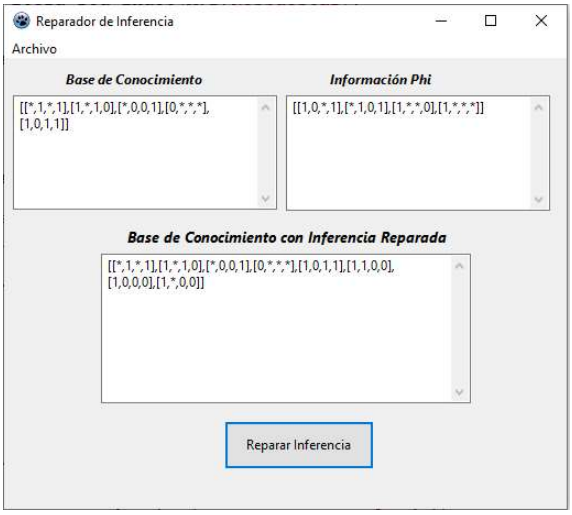

Figura 4. Ejemplo de KB reparada.

4.2.3 Comunicación entre GUI y PROLOG

La GUI no procesa internamente la información de la base de conocimiento ni de la información ϕ que debe inferir si no que debe ejecutar la consulta en PROLOG. Para poder interactuar con PROLOG, la GUI primero genera una carpeta temporal donde a su vez crea un archivo con el código fuente que utilizará PROLOG, es decir, los predicados desarrollados durante este trabajo de tesis.

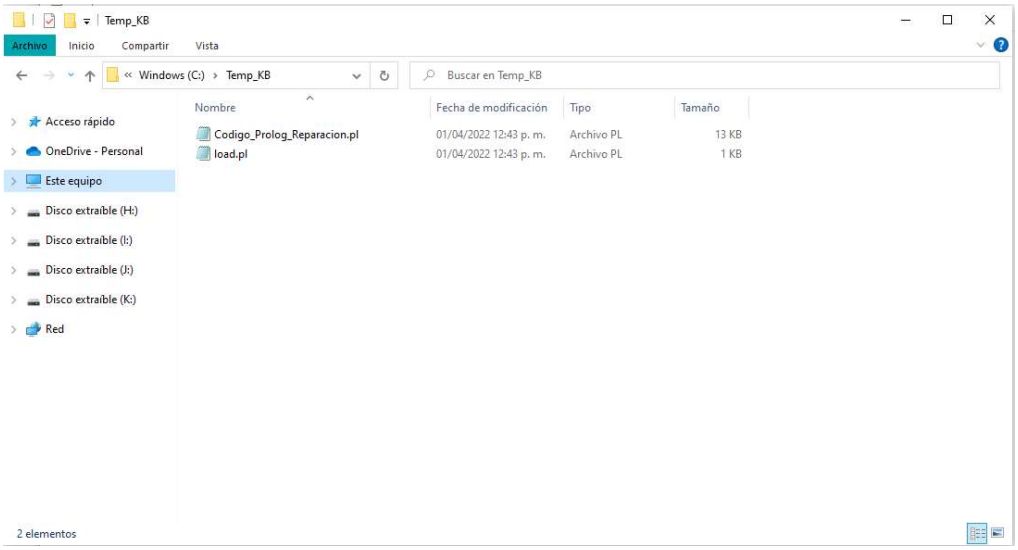

Figura 5. Archivos de Texto. Código fuente de PROLOG y Consulta

Después, con la información leída desde los archivos correspondientes a la base de conocimiento y la información ϕ, se crea un archivo que indica la consulta a realizarse en PROLOG.

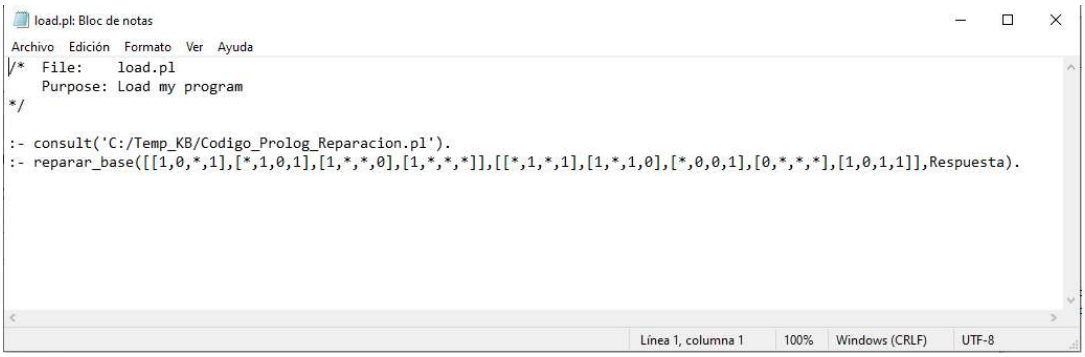

Fig 6. Load.pl. Archivo que ejecuta la consulta en PROLOG

Cuando se han creado ambos archivos, la GUI hace un llamado por medio de la consola de Windows a PROLOG, que ejecuta la consulta indicada. En cuanto el procedimiento termina, se cierra la ejecución de PROLOG y se procede a leer el archivo de texto *salida.txt* creado durante la ejecución del predicado **reparar_base**, presentando el resultado en el recuadro de texto correspondiente, tal y como se muestra en la figura 4 previamente presentada.

5 Ejemplo de aplicación del algoritmo implementado.

Ahora que el algoritmo de revisión de creencias ha sido implementado en PROLOG, cualquier aplicación que requiera realizar una revisión de creencias podría recurrir a nuestro sistema implementado en PROLOG, tal como se hace con la GUI mostrada al final del capítulo anterior, independientemente de su propósito o el lenguaje de programación en que haya sido hecha, aunque sí hay que establecer que tanto la base de conocimiento como la información que requiera ser revisada deben de codificarse en el formato ya establecido.

El predicado **correr_reduccion** toma como argumentos de entrada tanto para la KB como para ϕ, una lista que contiene una única cadena de 1´s, 0´s y *´s separados por comas que codifican a una única cláusula o una lista de listas, donde cada lista también es una cadena de los caracteres previamente mencionados que codifican cada una de las cláusulas que pueda contener la KB o ϕ, en FNC.

Por lo que para poder usar los predicados desarrollados se requiere partir de la FNC de la información.

Por ejemplo, sea la FNC siguiente, la que corresponde a la KB:

$$(p \vee \neg q \vee s) \wedge (q \vee r \vee \neg t) \wedge (\neg p \vee r \vee \neg s) \wedge (q \vee \neg s \vee t) \wedge (p \vee \neg r \vee t)$$

Se observa que la fórmula tiene 5 literales: p, q, r, s, t. Cada literal positiva es reemplazada por el número 1, mientras que las literales negativas son reemplazadas por un 0. Si una literal no aparece en la cláusula, su lugar será ocupado por el carácter comodín, que en este caso es representado por el asterisco: *.

$$(1 \vee 0 \vee * \vee 1 \vee *) \wedge (* \vee 1 \vee 1 \vee * \vee 0) \wedge (0 \vee * \vee 1 \vee 0 \vee *) \wedge$$

$$(* \vee 1 \vee * \vee 0 \vee 1) \wedge (1 \vee * \vee 0 \vee * \vee 1)$$

Cada operador y (∧) , o (∨), será reemplazado por una coma, obteniendo la siguiente codificación para la fórmula de entrada:

(1,0,*,1,*),(*,1,1,*,0),(0,*,1,0,*),(*,1,*,0,1),(1,*,0,*,1)

PROLOG identifica como lista aquella información contenida dentro de un paréntesis cuadrado ([ ]), así que deben reemplazarse los paréntesis por éstos y agrupar todos los términos dentro de otro par de paréntesis cuadrados, puesto que se trata de una lista de listas.

[[1,0,*,1,*],[*,1,1,*,0],[0,*,1,0,*],[*,1,*,0,1],[1,*,0,*,1]]

Así, aplicando el proceso descrito, la FNC siguiente correspondiente a ϕ:

$$(p \vee r) \wedge (q \vee s \vee \neg t)$$

Sería codificada por la siguiente cadena:

[[1,*,1,*,*],[*,1,*,1,0]]

Una vez que la KB y la fórmula ϕ se encuentran codificadas por una lista de listas, cualquier aplicación podrá hacer uso de los predicados desarrollados en PROLOG para realizar el proceso de revisión de creencias.

5.1 Diagnóstico médico basado en síntomas asistido por computadora

Usualmente, cuando un médico consulta a un paciente realiza una exploración, consistente en dos partes: un interrogatorio (conocido como anamnesis) mediante el cual recopila información que le permite construir una historia clínica. Información relevante como es la edad, estatura y peso del paciente, además le solicita conocer los síntomas que lo han llevado a buscar atención médica. La siguiente parte consiste en una inspección del paciente en busca de ciertos signos o manifestaciones objetivas físicas o químicas que permitan al médico, realizar un diagnóstico [19]. La experiencia cotidiana nos dice que para afecciones comunes, los análisis clínicos especializados no son solicitados, pues usualmente basta al médico con este procedimiento para que indique el tratamiento a seguir para la recuperación del paciente, sin embargo, ante la persistencia de los síntomas, ante un incremento en la gravedad de ellos, o si dentro de las posibles afecciones se encuentra una de gran peligrosidad, el médico puede solicitar a su criterio se realicen análisis clínicos adicionales.

Existen síntomas muy comunes y no exclusivos de una afección en particular, como pueden ser el dolor de cabeza y la fiebre, mientras que existen síntomas que pueden apuntar muy claramente hacia una enfermedad en particular. Así mismo, existen enfermedades que pueden compartir varios otros síntomas y, sin embargo, ante la misma enfermedad no todas las personas presentan exactamente los mismos síntomas.

El médico realiza entonces el diagnóstico de la enfermedad basándose en su conocimiento y experiencia y puede llegar a asistirse de su literatura en caso de alguna duda o un síntoma inusual. Mediante el algoritmo de revisión de creencias implementado en este trabajo de tesis, puede desarrollarse un sistema de asistencia de diagnóstico por computadora que ayude al médico en estos casos, o como se presenta en el ejemplo siguiente, cuando se presenta una enfermedad nueva como ha sido el caso del SARS-CoV-2 que surgió en el año 2019.

5.2 SARS-CoV-2

Hacia finales del año 2019 en China, específicamente en la región de Wuhan comenzaron a dispararse casos de una nueva neumonía en su población y no tardó mucho tiempo para que esta enfermedad se dispersara fuera de la región, e incluso del país, llegando la enfermedad a todo el mundo en poco tiempo [12]. La nueva enfermedad fue declarada una pandemia global por la Organización Mundial de la Salud el 11 de marzo de 2020 [13].

El virus causante de esta enfermedad pertenece a un grupo de virus denominados Coronavirus, siendo identificado con el nombre de SARS-CoV-2 aunque es comúnmente referido simplemente como COVID-19 por el año en que comenzó a difundirse y aunque no es el único coronavirus capaz de infectar al ser humano y causarle enfermedades con resultados mortales, puesto que previamente ha habido brotes de enfermedades respiratorias como SARS y MERS causadas por diferentes coronavirus, el SARS-CoV-2 ha probado ser más difícil de contener.

Los virus como todo organismo son susceptibles a mutaciones que pueden provocar pequeñas alteraciones en el mismo, y si bien la mayoría de ellas no alteran de forma significativa al virus, algunas mutaciones lo alteran de una forma suficientemente significativa como para clasificarse como una variante o cepa del virus. Así, diferentes cepas del virus tienen el potencial de causar diferentes síntomas de la enfermedad que provoca en las personas.

Las diferentes cepas llevan por nombre una letra griega. A la fecha La Organización Mundial de la Salud (OMS) ha identificado varias cepas del virus a las que ha dado seguimiento por ser potencialmente más peligrosas y contagiosas, destacando tres, que son la cepa original denominada Alfa, la variante Delta y la variante Omicron [20].

Al tratarse de una enfermedad tan reciente, no existe suficiente información para contar con una sintomatología definitiva que permita diagnosticarla sin ayuda de un análisis PCR que busca específicamente por el virus en la persona, sin embargo la identificación de la cepa no es un estudio estándar que se realice en cada prueba PCR y aunque las diferentes cepas pueden compartir algunos síntomas, hay síntomas que se pueden presentar más frecuentemente dependiendo de la cepa de Covid-19 que la persona haya adquirido.

Finalmente, se debe mencionar que la importancia de diagnosticar la cepa radica en que cada cepa tiene diferentes índices de pacientes que desarrollan complicaciones, por lo que conocer la cepa causante de la enfermedad puede ayudar al médico en decidir si un paciente requiere cuidados especiales adicionales o no.

5.3 Construcción de la base de conocimiento de síntomas de SARS-CoV-2

Por lo anterior, es que se presenta como ejemplo de aplicación al algoritmo de revisión de creencias, una aplicación que recoge los síntomas más comúnmente reportados por las personas que han sido contagiadas por Covid-19 y diagnosticadas por medio de la prueba PCR con la cepa confirmada.

Como previamente se ha establecido, la representación del conocimiento es fundamental para la aplicación y el primer paso para desarrollar la aplicación deberá ser construir la base de conocimiento de la cual se partirá.

Lo primero que se realiza es definir los síntomas más comúnmente presentados por los pacientes de Covid-19 y sus cepas, que podemos encontrar en la tabla 3 y asociarlos con la cepa de SARS-CoV-2 correspondiente.

Síntomas más frecuentes del Covid-19
Tos
Congestión Nasal
Fatiga
Dolor de Garganta
Pérdida del Apetito
Pérdida del Olfato
Estornudos
Dolor de Cabeza
Dolor Muscular
Fiebre
Pérdida en el gusto
Falta de Aliento
Dificultad para respirar
Diarrea

Tabla 3. Síntomas más frecuentes de la enfermedad causada por SARS-CoV-2

Estadísticamente se observa en estudios que la tos, el dolor de cabeza y la fiebre son síntomas comunes a la enfermedad ocasionada por las 3 cepas del virus, mientras que otros síntomas se presentan más frecuentemente en la enfermedad que causa cada una de las cepas. Esta información se muestra en la tabla 4.

Si se busca el síntoma más común asociado a cada cepa, se encuentra que, para la variante Alfa, se trata de la pérdida del olfato, para Delta es la falta de aliento y para Omicron, se trata del dolor de garganta. Si es considerado que la presencia de los 3 síntomas comunes a todas las cepas más el síntoma más característico de cada una indica una probabilidad de 50% de indicar la cepa, se puede iniciar la construcción de la base de conocimiento utilizando la lógica proposicional en FNC:

$$(Tos \wedge Pérdida\ del\ Olfato \wedge Dolor\ de\ Cabeza \wedge Fiebre) \rightarrow Alfa50$$

$$(Tos \wedge Dolor\ de\ Cabeza \wedge Fiebre \wedge Falta\ de\ Aliento) \rightarrow Delta50$$

$$(Tos \wedge Dolor\ de\ Cabeza \wedge Fiebre \wedge Pérdida\ en\ el\ gusto) \rightarrow Omicron50$$

Sintoma	Alfa	Delta	Omicron
Tos	x	x	x
Congestion o escurrimiento Nasal		x	x
Fatiga	x		x
Dolor de Garganta		x	x
Pérdida del Apetito	x		x
Pérdida del Olfato	x		
Estornudos			x
Dolor de Cabeza	x	x	x
Dolor Muscular		x	x
Fiebre	x	x	x
Pérdida en el gusto			x
Falta de Aliento	x		
Dificultad para respirar	x	x	
Diarrea	x		

Tabla 4. Síntomas asociados a la cepa causante de la enfermedad.

Ahora partiendo de la implicación de los síntomas, es posible agregar que, si se presenta un síntoma adicional menos frecuente como son diarrea para la variante Alfa, dolor muscular para la variante delta y estornudos frecuentes para la variante omicron, es posible identificar con mayor certeza la cepa causante de la enfermedad:

$$(Alfa50) \wedge (Diarrea) \rightarrow Alfa70$$

$$(Delta50) \wedge (Dolor\ Muscular) \rightarrow Delta70$$

$$(Omicron50) \wedge (Estornudos) \rightarrow Omicron70$$

También existen síntomas que son compartidos por a lo más dos variantes, así que si se presenta alguno de esos síntomas, podría incrementarse la certeza del diagnóstico:

$$(Alfa70) \wedge (Fatiga \vee Pérdida\ del\ Apetito) \rightarrow Alfa90$$

que distribuyendo pasa a ser:

((Alfa70 ∧ Fatiga) ⇾ Alfa90) ∧ ((Alfa70 ∧ Perdida del Apetito) ⇾ Alfa90)

(Delta70) ∧ (Congestion Nasal ∨ Dolor de Garganta) ⇾ Delta90

que distribuyendo pasa a ser:

((Delta70 ∧ Congestion Nasal) ⇾ Delta90) ∧ ((Delta70 ∧ Dolor de Garganta) ⇾ Delta90)

(Omicron70) ∧ (Congestion Nasal ∨ Fatiga ∨ Dolor de Garganta ∨ Pérdida del Apetito) ⇾ Omicron90

que distribuyendo pasa a ser:

((Omicron70 ∧ Congestion Nasal) ⇾ Omicron90) ∧ ((Omicron70 ∧ Fatiga) ⇾ Omicron90) ∧ ((Omicron70 ∧ Dolor de Garganta) ⇾ Omicron90) ∧ ((Omicron70 ∧ Pérdida del Apetito) ⇾ Omicron90)

(Dificultad para respirar) ∧ (Alfa50 ∨ Delta50 ∨ Omicron50) ⇾ Internar

que distribuyendo pasa a ser:

((Dificultad para respirar ∧ Alfa50) ⇾ Internar) ∧ ((Dificultad para respirar ∧ Delta50) ⇾ Internar) ∧ ((Dificultad para respirar ∧ Omicron50) ⇾ Internar)

Ahora se debe transformar las implicaciones en cláusulas, con lo que obtenemos:

(¬Tos ∨ ¬Pérdida del Olfato ∨ ¬Dolor de Cabeza ∨ ¬Fiebre ∨ Alfa50)

(¬Tos ∨ ¬Dolor de Cabeza ∨ ¬Fiebre ∨ ¬Falta de Aliento ∨ Delta50)

(¬Tos ∨ ¬Dolor de Cabeza ∨ ¬Fiebre ∨ ¬Pérdida en el gusto ∨ Omicron50)

(¬Alfa50 ∨ ¬Diarrea ∨ Alfa70)

(¬Delta50 ∨ ¬Dolor Muscular ∨ Delta70)

(¬Omicron50 ∨ ¬Estornudos ∨ Omicron70)

(¬Alfa70 ∨ ¬Fatiga ∨ Alfa90) ∧ (¬Alfa70 ∨ ¬Perdida del Apetito ∨ Alfa90)

(¬Delta70 ∨ ¬Congestion Nasal ∨ Delta90) ∧ (¬Delta70 ∨ ¬Dolor de Garganta ∨ Delta90)

(¬Omicron70 ∨ ¬Congestion Nasal ∨ Omicron90) ∧ (¬Omicron70 ∨ ¬Fatiga ∨ Omicron90) ∧ (¬Omicron70 ∨ ¬Dolor de Garganta ∨ Omicron90) ∧ (¬Omicron70 ∨ ¬Pérdida del Apetito ∨ Omicron90)

(¬Dificultad para respirar ∨¬Alfa50 ∨Internar) ∧(¬Dificultad para respirar ∨¬Delta50 ∨Internar) ∧(¬Dificultad para respirar ∧¬Omicron50 ∨Internar)

Lo siguiente es generar las cadenas falsificantes correspondientes a cada cláusula. Para ello se sustituirán los nombres de los síntomas y los diagnósticos por literales, como se observa en la tabla 5, tenemos 24 literales.

Variable	Literal
Tos	X1
Congestión Nasal	X2
Fatiga	X3
Dolor de Garganta	X4
Pérdida del Apetito	X5
Pérdida del Olfato	X6
Estornudos	X7
Dolor de Cabeza	X8
Dolor Muscular	X9
Fiebre	X10
Pérdida en el gusto	X11
Falta de Aliento	X12
Dificultad para respirar	X13
Diarrea	X14
Alfa50	X15
Alfa70	X16
Alfa90	X17
Delta50	X18
Delta70	X19
Delta90	X20
Omicron50	X21
Omicron70	X22
Omicron90	X23
Internar	X24

Tabla 5. Asignación de literales

Como las cláusulas se codifican vía cadenas falsificantes, para cada literal positivo de una cláusula se colocará el valor 0, mientras que para cada literal negativo de una cláusula se colocará el valor 1, y para todos aquellos literales que no aparecen en la cláusula se utiliza el

*. Así, las cadenas falsificantes correspondientes a la base de conocimiento definida se pueden observar en la tabla 6.

	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12	X13	X14	X15	X16	X17	X18	X19	X20	X21	X22	X23	X24
C 1	1	*	*	*	*	1	*	1	*	1	*	*	*	*	0	*	*	*	*	*	*	*	*	*
C 2	1	*	*	*	*	*	*	1	*	1	*	1	*	*	*	*	*	0	*	*	*	*	*	*
C 3	1	*	*	*	*	*	*	1	*	1	1	*	*	*	*	*	*	*	*	*	0	*	*	*
C 4	*	*	*	*	*	*	*	*	*	*	*	*	*	1	1	0	*	*	*	*	*	*	*	*
C 5	*	*	*	*	*	*	*	*	1	*	*	*	*	*	*	*	*	1	0	*	*	*	*	*
C 6	*	*	*	*	*	*	1	*	*	*	*	*	*	*	*	*	*	*	*	*	1	0	*	*
C 7	*	*	1	*	*	*	*	*	*	*	*	*	*	*	*	1	0	*	*	*	*	*	*	*
C 8	*	*	*	*	1	*	*	*	*	*	*	*	*	*	*	1	0	*	*	*	*	*	*	*
C 9	*	1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	1	0	*	*	*	*
C 10	*	*	*	1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	1	0	*	*	*	*
C 11	*	1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	1	0	*
C 12	*	*	1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	1	0	*
C 13	*	*	*	1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	1	0	*
C 14	*	*	*	*	1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	1	0	*
C 15	*	*	*	*	*	*	*	*	*	*	*	*	1	*	1	*	*	*	*	*	*	*	*	0
C 16	*	*	*	*	*	*	*	*	*	*	*	*	1	*	*	1	*	*	*	*	*	*	*	0
C 17	*	*	*	*	*	*	*	*	*	*	*	*	1	*	*	*	1	*	*	*	*	*	*	0

Tabla 6. Cláusulas en formato de cadena falsificante.

Como se puede observar en la tabla 5 de la página previa (tabla 4,), los literales 1 a 14 corresponden a síntomas, mientras que del 15 al 24 corresponden a un diagnóstico o implicación. Si se tiene información únicamente de síntomas, es posible aplicar el operador independencia entre la cadena falsificante correspondiente a la consulta con cada una de las cláusulas de la base de conocimiento para encontrar cuál es la información faltante (síntomas) que haga cumplir la correspondiente inferencia.

De esta forma, es posible sugerir los síntomas no reportados por el paciente al médico en busca de determinar la variante causante de la enfermedad.

5.4 Aplicación para diagnóstico asistido por computadora de cepa de SARS-CoV-2

Para desarrollar la aplicación en cuestión nuevamente se recurrió al IDE Lazarus, con la finalidad de desarrollar una GUI fácil y rápidamente. En el diseño se presenta un formulario que aloja el grupo de 14 síntomas definidos y de los que pueden ser seleccionados los síntomas que manifiesta el paciente, un botón para iniciar la consulta y un cuadro de texto donde se despliegan aquellos síntomas que se deben indagar para poder sugerir la cepa causante de la enfermedad.

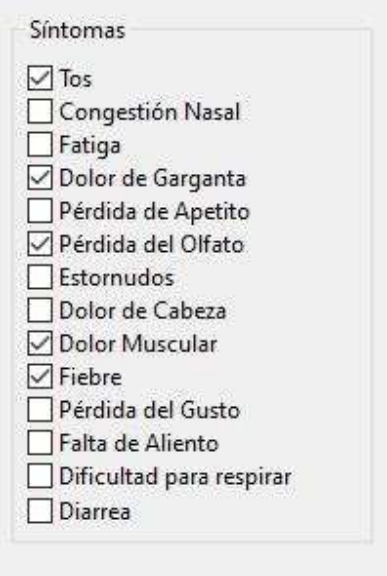

Figura 7. Recuadro de selección de síntomas de la aplicación.

La aplicación realiza el mismo proceso que la GUI diseñada para reparar la inferencia lógica con una diferencia: el proceso iterativo no se realiza a toda la base de conocimiento, si no que se realiza la reparación de inferencia cláusula por cláusula, obteniendo así las cláusulas que faltan para que se cumpla el consecuente de cada una de ellas, es decir, el diagnóstico (figura 9).

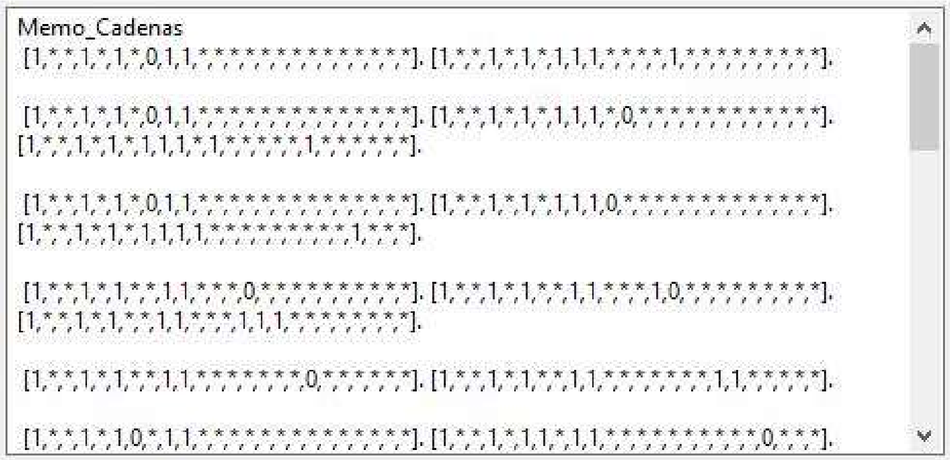

Figura 8. Cadenas resultantes al aplicar el operador Independencia a cada cadena de la base de conocimiento.

Adicionalmente, a cada cadena o cláusula resultante, debe de restársele la cláusula de la base de conocimiento correspondiente, con el fin de obtener la información (síntomas) faltante para que se cumpla el diagnóstico correspondiente.

Entre los predicados desarrollados previamente se encuentra **amenosb**, que realiza esta resta entre cadenas:

```
%Restar cadenas A-B:
amenosb([],[],[]).
amenosb([X|L],[X|L1],[*|Z]) :- amenosb(L,L1,Z),!.
amenosb([X|L],[_|L1],[X|Z]) :- amenosb(L,L1,Z),!.
```

Sin embargo, para que la aplicación hecha en Lazarus sea capaz de obtener los resultados dados por el predicado, éstos deberán ser escritos en un archivo de texto desde donde pueda leerlos. Con este propósito se desarrolló el predicado **obtener_Z:**

```
/*Para obtener el resultado de amenosb y escribirlo en un archivo de texto*/
obtener_Z(A,B,Z) :- amenosb(A,B,Z), imprime_Z(Z), print(Z).
```

Que como se puede observar, hace uso de un tercer predicado, **imprime_Z**, encargado de escribir en un archivo temporal de texto el resultado de **amenosb**:

```
imprime_Z(Z) :-  exists_file('C:/Temp_Diagnostico/amenosb.txt'),
                 delete_file('C:/Temp_Diagnostico/amenosb.txt'),
                 open('C:/Temp_Diagnostico/amenosb.txt',write,Stream),
                 write(Stream,Z), close(Stream).
```

```
imprime_Z(Z) :-  not(exists_file('C:/Temp_Diagnostico/amenosb.txt')),
                 open('C:/Temp_Diagnostico/amenosb.txt',write,Stream),
                 write(Stream,Z), close(Stream).
```

Una vez realizada la resta entre cadenas, la aplicación procede a interpretar los resultados obtenidos para cada cláusula de la base de conocimiento y procede a indicar en el cuadro de texto los síntomas faltantes que ayuden a identificar la identidad de la cepa que infectó al paciente, lo que se muestra en el cuadro de texto de resultados de la aplicación, como puede observarse en la figura T.

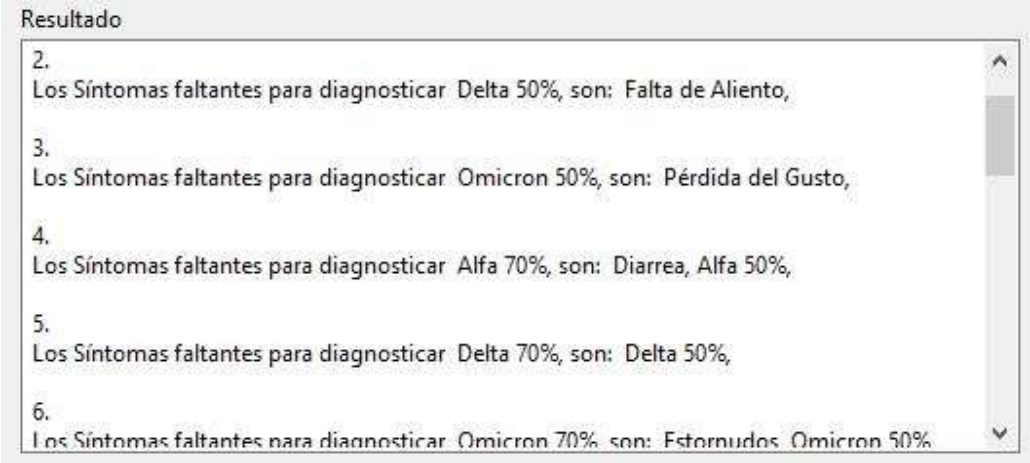

Figura 9. Interpretación: Cuadro de texto con lista de síntomas a buscar para obtener un diagnóstico.

En caso de que, con los síntomas reportados por el paciente, no sean obtenidos síntomas faltantes para indicar un diagnóstico, lo que correspondería a que la información consultada efectivamente se infiere de la sentencia, aparece un mensaje que indica que no falta ningún síntoma para diagnosticar el consecuente correspondiente, tal y como se observa en la figura 10.

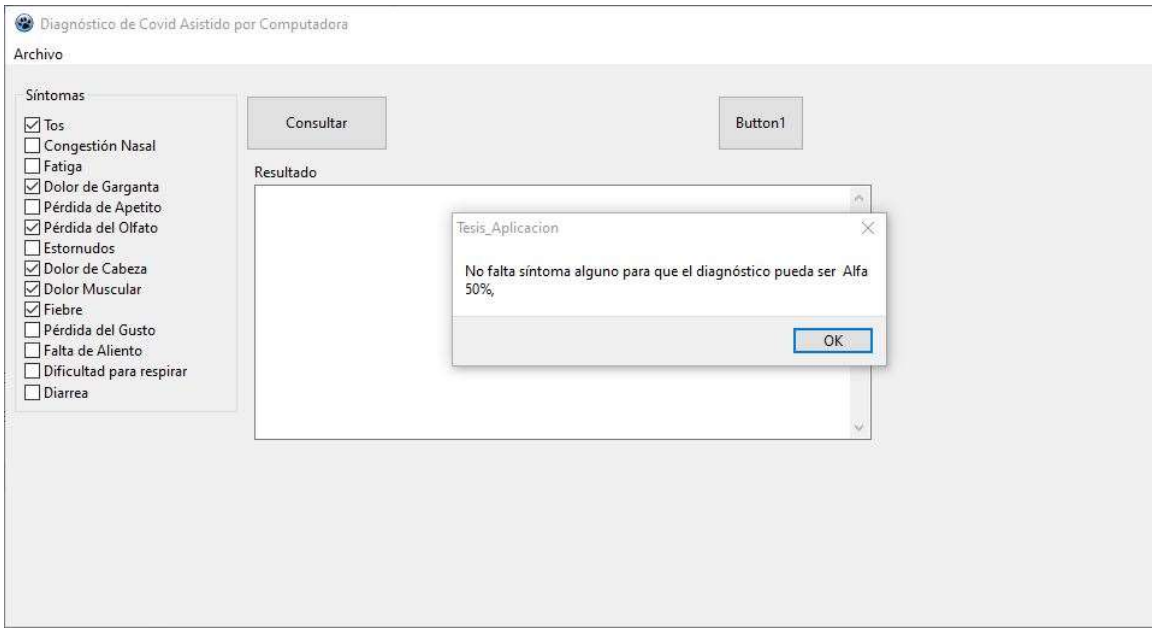

Figura 10. Posible diagnóstico en base a los síntomas reportados.

6 Conclusiones.

Si bien el campo de la Inteligencia Artificial es relativamente nuevo, con menos de cien años de existencia, ha presentado en años recientes avances significativos, despertando cada vez más interés en su estudio y llevando a que se presente en la actualidad un gran número de aplicaciones en la vida cotidiana. Sin embargo, aún se encuentra lejos de conseguir la construcción de una máquina 'inteligente' puesto que faltan muchos retos por superar.

Uno de estos retos es el de automatizar el razonamiento para crear agentes que sean capaces de actuar racionalmente. Para ello es necesario dotarles de la capacidad de aprendizaje, es decir que deben ser capaces no solo de almacenar o representar conocimiento, además deberían poder acrecentarlo e incluso modificarlo cuando se encuentre ante nueva información que complemente o incluso contradiga lo que ya conoce, de forma que la base de conocimiento resultante conserve su consistencia.

Dado que la simple acción de agregar cada nueva información a una base de conocimiento no es algo adecuado, puesto que puede provocar que ésta se vuelva inconsistente o puede hacerla crecer con información redundante. Para determinar si la información es o no contenida en la base de conocimiento, es necesario corroborar si lo que desea agregar puede ser inferido o no de lo ya conocido y para ello es posible valerse de la inferencia lógica.

La revisión de creencias es una de las formas principales en que un agente modifica su comportamiento y, aunque existen diversas propuestas para el proceso de revisión de creencias, las más comunes se enfocan en la búsqueda de asignaciones que satisfagan a la base de conocimiento, es decir sus modelos, lo que es un problema Co-NP Completo.

De lo anterior se desprende la importancia que tiene el desarrollo de métodos que permitan a agentes racionales modificar sus creencias, es decir, lo que el agente sabe, como el algoritmo presentado por De Ita et al. [2], que fue la base para este trabajo.

Al implementar dicho algoritmo en PROLOG, fue posible utilizarlo como un método de reparación de una inferencia lógica. Sin embargo, no es la única forma de aplicar el algoritmo, y como ejemplo, se ha desarrollado una breve aplicación orientada a un diagnóstico asistido por computadora enfocada en el aspecto médico.

Durante la etapa de elaboración del estado del arte se pudo constatar que a pesar de existir numerosos trabajos en el tema de cambios de creencias, la implementación de las propuestas ha sido un tanto desdeñado lo que podría dificultar la aplicación de estos procesos en aplicaciones prácticas, por lo que el trabajo de implementación realizado en esta tesis puede servir de base para desarrollar agentes o aplicaciones inteligentes.

El mayor desafío durante el desarrollo de los predicados necesarios es que se trata de un proceso iterativo lo que provoca listas de cláusulas anidadas en PROLOG. Si bien se trabajó en un principio en una solución directa en PROLOG para desanidar dichas listas, se constató que utilizar archivos de texto temporales constituía una solución más rápida y práctica por lo que se recurrió a ellos para obtener los resultados en forma de una lista no anidada de cláusulas, mediante la escritura y lectura de ellos.

Al respecto de la comunicación entre la aplicación que hace uso de los predicados desarrollados en PROLOG, en este trabajo se realiza mediante scripts que son ejecutados en un sistema operativo WINDOWS, este mismo proceso podría llegar a ser utilizado en un sistema operativo diferente que permita de la misma forma correr scripts desde una consola de comandos, ajustando el formato de los archivos y comandos al sistema operativo en uso. Una alternativa más elegante a esto es contar con PROLOG embebido dentro de la aplicación programada en el lenguaje de elección, sin embargo, al encontrarse fuera del alcance de este trabajo, no se exploró esta posibilidad.

Ambas aplicaciones han sido desarrolladas en un lenguaje de alto nivel llamado Lazarus, lo que demuestra, que es posible utilizar cualquier otro lenguaje de programación para desarrollar una interfase o incluso diseñar un agente racional que únicamente haga uso de PROLOG y del código fuente desarrollado en este trabajo para su proceso de revisión de creencias, pero que reciba información a través de sensores y reaccione por medio de actuadores controlados directamente desde un programa diferente.

Bibliografía

[1] De Ita, G., Marcial-Romero, R., Bello, P., & Contreras, M. (2018). *Belief revision between conjunctive normal forms* [JB]. Journal of Intelligent & Fuzzy Systems, 34(5), 3155–3164. https://doi.org/10.3233/JIFS-169499

[2] De Ita, G., Bello, P. *An Algorithm to Belief Revision and to Verify Consistency of a Knowledge Base*. IEEE LATIN AMERICA TRANSACTIONS, VOL. 19, NO. 11, NOVEMBER 2021

[3] Alain Colmerauer and Philippe Roussel. 1996. *The birth of Prolog. History of programming languages*---II. Association for Computing Machinery, New York, NY, USA, 331–367. https://doi.org/10.1145/234286.1057820

[4] Russel, S.J. Norvig, P.. (2004). *Inteligencia Artificial, un enfoque moderno*. Madrid: Pearson Educación, S.A.

[5] Bello L., P.. (2018). *Modelando las normas con revisión de creencias e inferencia*. Elementos, 112, 9-13.

[6] Liberatore, P. (1997). *The complexity of iterated belief revision*. In Lecture Notes in Computer Science (pp. 276–290). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-62222-5_51

[7] De Ita, G., Zacarias Flores, F., & García García, A. D. (2017). *Model-based Algorithm for Belief Revisions between Normal Conjunctive Forms*. Computación y Sistemas, 21(3). https://doi.org/10.13053/cys-21-3-2442.

[8] Alchourrón, C. E., Gärdenfors, P., & Makinson, D. (1985). *On the logic of theory change: Partial meet contraction and revision functions*. Journal of Symbolic Logic, 50(2), 510–530. https://doi.org/10.2307/2274239

[9] Katsuno, H., & Mendelzon, A. O. (1991). *Propositional knowledge base revision and minimal change*. Artificial Intelligence, 52(3), 263–294. https://doi.org/10.1016/0004-3702(91)90069-v

[10] Liberatore P. *The complexity of iterated belief revision*. Lecture Notes in Computer Science, 1186:276{292, 1997}

[11] Gärdenfors, Peter. (1992). *Belief Revision: an introduction*. 10.1017/CBO9780511526664.001. https://www.researchgate.net/publication/260134902_Belief_Revision_an_introduction

[12] Zhan T, Tang Y, Han Z, *et al*. *Clinical characteristics of 195 cases of COVID-19 with gastrointestinal symptoms*. Turk J Gastroenterol. 2021; 32(2): 148-154.

[13] Fung, T.S., Liu, D.X. (2021) *Similarities and Dissimilarities of COVID-19 and Other Coronavirus Diseases*. Annual Review of Microbiology. 75(1): 19-47.

[14] Buchanan, B. G. (2005). *A (Very) Brief History of Artificial Intelligence*. AI Magazine, 26(4), 53. https://doi.org/10.1609/aimag.v26i4.1848

[15] McCulloch, W.S., Pitts, W. *A logical calculus of the ideas immanent in nervous activity*. Bulletin of Mathematical Biophysics 5, 115–133 (1943). https://doi.org/10.1007/BF02478259

[16] Flouris, George & Plexousakis, Dimitris. (2001). *Belief Revision in Propositional Knowledge Bases*.

[17] Gamut, L. T. F., & Durán, C. (2002). *Introducción a la lógica*. Buenos Aires: Eudeba.

[18] Oviedo, & Labra Gayo, Jose & Ana, G & Fernández, I. (2022). *Universidad de Lógica Proposicional para Informática*.

[19] Surós Batlló, A., & Surós Batlló, J. (2001). *Semiología médica y técnica exploratoria* (8th ed.). Masson.

[20] World Health Organization. (2022). *Seguimiento de las variantes del SARS-COV-2*. World Health Organization. Retrieved May 9, 2022, from https://www.who.int/es/activities/tracking-SARS-CoV-2-variants

[21] Vicol, Paul & Delgrande, James & Schaub, Torsten. (2015). An Implementation of Consistency-Based Multi-Agent Belief Change using ASP. 10.1007/978-3-319-23264-5_40.

[22] Hunter, Aaron & Agapeyev, John. (2020). GenC: A Fast Tool for Applications Involving Belief Revision. 5090-5092. 10.24963/ijcai.2020/725.

[23] Hunter, A. and Boyarinov, K. (2022). BRL: A Toolkit for Learning How an Agent Performs Belief Revision. In Proceedings of the 14th International Conference on Agents and Artificial Intelligence - Volume 3: ICAART, ISBN 978-989-758-547-0; ISSN 2184-433X, pages 753-756. DOI: 10.5220/0010899100003116