Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Computación

Tesis para obtener el título de:

Licenciado en Ciencias de la Computación

Algoritmo para la consistencia de una base de conocimiento

Presenta:

Carlos Aviña Zamora

Asesor:

M.C. Pedro Bello López

Noviembre de 2015

AGRADECIMIENTOS

A ti dios mío por haberme puesto en este camino. Porque sé que siempre caminas a mi lado. Todo llega en su momento y si en esta etapa de mi vida estoy presentando este trabajo es porque tú así lo quisiste.

Por supuesto a mis padres, muy especialmente a mi madre que con mucho esfuerzo y sacrificio me dieron estudios, me enseño que el trabajo y la constancia es una herramienta poderosa. Por el apoyo incondicional que siempre recibí, por la confianza que siempre me ha otorgado, por las palabras de aliento que nunca me faltaron.

A mis hermanos que en todo momento están conmigo, apoyando siempre mis acciones.

A mi asesor, que con mucha paciencia me apoyo en todo momento con su conocimiento y experiencia.

Contenido

Introducción

El presente trabajo muestra la forma de procesar la información cuando una entidad se expone a nueva información y esta entidad debe asimilar la nueva información para poder realizar sus actividades, sin entrar en contradicción con la información almacenada. Las entidades en cuestión pueden ser vehículos autotripulados, robots, sondas de exploración, autómatas, etcétera. Y puede ser visto como un “aprendizaje” de la experiencia de la entidad.

Para el tratamiento de la información utilizamos una serie conceptos preliminares explicados ampliamente en el capítulo uno de este trabajo. La información se organiza en clausulas y se muestra gráficamente utilizando grafos, a partir de dichas clausulas se pueden escribir o interpretar el conocimiento en expresiones lógicas, estas expresiones lógicas se pueden analizar para determinar la validez de un razonamiento. Cualquier expresión lógica se puede procesar utilizando el método de la cerradura transitiva (a,b) $\rightarrow$ (-a,b)(-b,a) para simplificar la expresión y mostrarla en su forma más simple. Con la teoría de revisión de creencias permite agregar nueva información a la base de conocimientos (expansión) primaria o en su defecto retirar (contracción) clausulas. Se debe priorizar la información más reciente y en caso de que entre en conflicto la base de conocimiento con la nueva información, se deberá de modificar las clausulas a fin de que este nuevo conocimiento sea viable. En particular las bases de conocimiento es modelada utilizando la 2FNC y utilizando la cerradura transitiva para determinar si la formula expresada como conocimiento es satisfactible o no, cada formula se lee en su 2FNC y se construye un grupo de elementos denominados TX que contienen las variables alcanzables según una variable de entrada, si por lo menos una literal xi o –xi es consistente y esto se cumple para cada literal, entonces la formula modelada es satisfactible.

Capítulo 1. Conceptos preliminares

En este capítulo introduciremos los conceptos básicos de la teoría de grafos, conceptos a través de los cuales se representa la información en una colección finita de literales, hablamos también de Satisfactibilidad termino que permite garantizar la computabilidad del programa, la forma normal conjuntiva que permitirá convertir cualquier expresión lógica a una forma lógica conjuntiva para que pueda ser procesada. Anexamos como ejemplo la revisión de creencias como aplicación donde se utiliza de forma natural bases de conocimiento.

1.1 Teoría de grafos

El grafo es una es una red de elementos conectados entre sí. En la actualidad se le considera una de las herramientas más utilizadas para modelar problemas reales, ya que, generalmente, algunos componentes de estos problemas interactúan, lo que fácilmente puede representarse en un grafo [3].

El grafo podría explicarse como la red de carreteras de un país (figura 1.1), que tiene un conjunto de ciudades conectadas a través de caminos. No todas las poblaciones tienen la misma cantidad de vías, y no necesariamente se puede llegar directamente de una ciudad a otra.

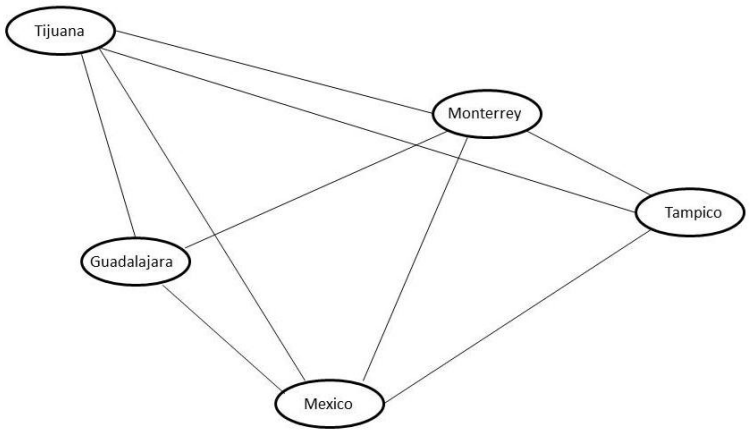


Figura 1.1 Imagen que muestra el aspecto visual de un grafo.

El grafo se considera como la estructura más general dentro de la clasificación tradicional de estructuras de datos, ya que es la generalización de una estructura jerárquica, la cual, a su vez, resulta ser la generalización de la estructura lineal. Cada elemento del grafo mantiene una relación de N:M con respecto a los demás elementos [3].

1.1.1 Terminología utilizada en grafos

Grafo: conjunto de nodos y arcos o de vértices y aristas.

Nodo: elemento básico de información de un grafo.

Arco: liga que une dos nodos del grafo.

Nodos adyacentes: dos nodos son adyacentes si hay un arco que los conecte.

Camino: secuencia de nodos, en la que cada par de nodos son adyacentes.

Camino simple: es un camino en el que todos los nodos contenidos son diferentes, con la posible excepción del primero y el último, que podrían ser el mismo.

Grafo no dirigido: los arcos en el grafo no tienen una dirección particular, es decir, son bidireccionales. Por ejemplo: si los nodos A y B son adyacentes, es igual ir de A a B que de B a A.

Grafo dirigido (Dígrafo): Los arcos en el grafo tienen una dirección asociada. El primer elemento del arco es el origen y el segundo es considerado el destino. Ejemplo el arco A B es diferente del arco B A.

Grafo ponderado: cada arco del grafo tiene asociado un peso o valor. Generalmente el peso está relacionado con costos, distancias y similares.

Ciclo: Es un grafo dirigido el ciclo es un camino donde el nodo de inicio y el nodo de terminación es el mismo.

Grafo conectado: un grafo no dirigido es conectado si hay un camino de cualquier nodo hacia cualquier otro en el grafo.

Digrafo fuertemente conectado: un grafo dirigido se considera fuertemente conectado si hay un camino desde cualquier nodo hacia cualquier otro.

Digrafo débilmente conectado: un grafo dirigido se considera débilmente conectado si su grafo no dirigido correspondiente está conectado, pero no hay caminos para llegar de cualquier nodo hacia cualquier otro.

Grados de un grafo: es el máximo grado de sus nodos, donde este se define como la cantidad de arcos que inciden en ese nodo. En el caso de dígrafos se distingue entre el grado_entrada y el grado_salida. El primero define la cantidad de arcos en los que el nodo es el destino y el segundo es la cantidad de arcos donde es el origen.

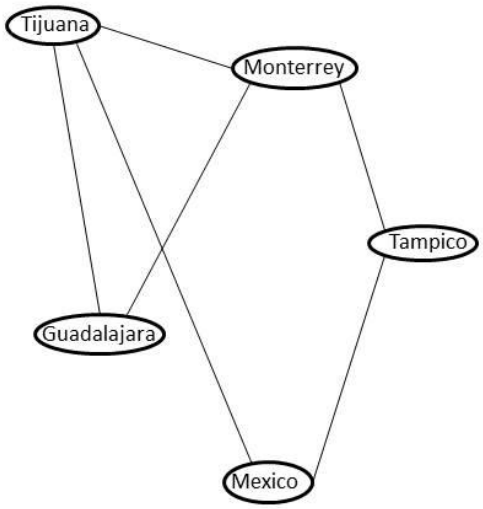
1.1.2 Matriz de adyacencias

Una de las representaciones más sencillas de un grafo es la denominada matriz de adyacencias, que consiste básicamente en un arreglo bidimensional de tamaño n x

n donde n es la máxima cantidad de nodos en el grafo. Cada casilla de la matriz será llenada con verdadero o falso, según exista o no un arco que conecte los dos nodos involucrados (figura 1.2). Para el caso de los grafos no dirigidos, la matriz será simétrica; sin embargo, esto no ocurre en los dígrafos, donde se debe considerar la dirección explicita de cada uno de los arcos[3].

Para el caso de los grafos ponderados, la matriz puede llenarse con el peso asociado a cada uno de los arcos.

La principal ventaja de este tipo de representación es su simplicidad, ya que resulta relativamente fácil realizar cada una de las operaciones descritas para un grafo. Desafortunadamente, esta representación tiene una fuerte desventaja: está limitada a un número máximo de vértices en el grafo, lo que provoca que, en cierto momento, sea imposible almacenar más información o que, por el contrario, si se supone un número máximo muy grande, se desperdicie una gran cantidad de memoria. Para el caso de un grafo no dirigido el desperdicio seria mayor, ya que al ser simétrica la matriz, la información esta duplicada.



	Tijuana	Tampico	México	Monterrey	Guadalajara
Tijuana	F	F	V	V	V
Tampico	F	F	V	V	F
México	V	V	F	F	F
Monterrey	V	V	F	F	V
Guadalajara	V	F	F	V	F

Figura 1.2 Ejemplo de una matriz de adyacencias.

1.1.3 Lista de adyacencias

En esta representación, generalmente dinámica, se trata de evitar el problema de restricción de espacio. Básicamente consiste en definir una lista encadenada de nodos y, para cada uno de ellos, encadenar una lista con los nodos adyacentes. La idea es similar a la de la matriz; sin embargo, no define las casillas de los nodos que no tienen un arco común, ahorrando un poco de espacio.

En este tipo de representación se desperdicia mucho espacio cuando se almacena un grafo no dirigido ya que, al igual que en la matriz de adyacencias, la información de cada arco esta duplicada; pues hay un arco de un nodo a otro y viceversa. Sin embargo, esta representación es la más utilizada cuando se tiene que almacenar un grafo disperso, donde la cantidad de arcos es mucho menor que el cuadrado del número de nodos(es decir que existen pocas conexiones). Esta tipo de representación se muestra en la figura 1.3

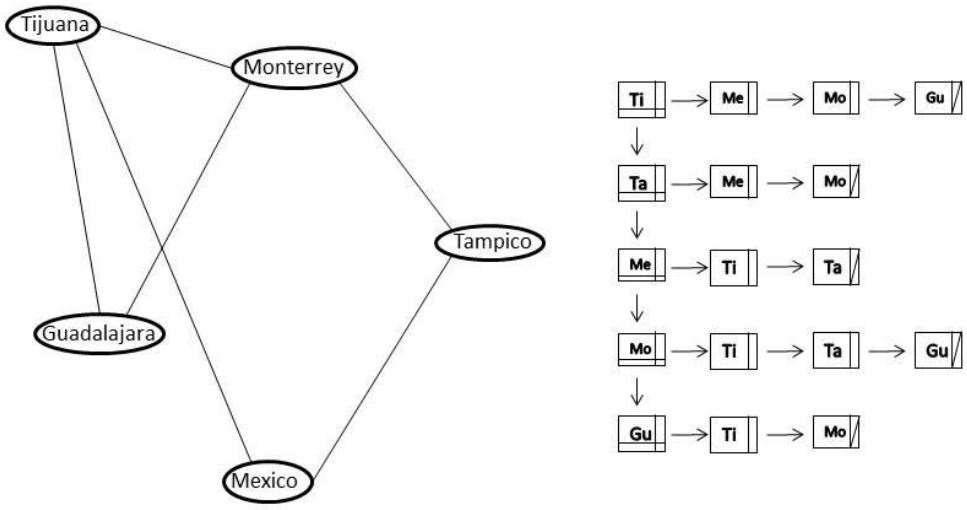


Figura 1.3 Ejemplo de un grafo en listas de adyacencias.

1.1.4 ¿Cómo se puede recorrer un grafo?

Al igual que en los árboles, los grafos requieren algoritmos especializados para desplegar la información contenida en ellos. Básicamente se manejan dos algoritmos que sirven para recorrer un grafo, y lo hacen siguiendo un orden diferente, dependiendo de la representación seleccionada para el grafo. Dado el grado de complejidad de la representación de grafos, estos algoritmos se apoyan en estructuras de datos auxiliares; primero en anchura (breadth first), utiliza una fila; y el segundo, llamado primero en profundidad (Deep first), emplea una pila.

Búsqueda en anchura

La búsqueda en anchura (o búsqueda en amplitud). Llamada Breadth First Search en inglés, es un algoritmo usado para recorrer o buscar elementos en una estructura de datos como los árboles y los grafos. Su procedimiento consiste en ir visitando todos los nodos de un nivel antes de proceder con el siguiente nivel tal y como se observa en la imagen 1.4. De modo que lo primera que hará será visitar la raíz, luego los hijos de la raíz, luego los hijos de cada uno de estos hijos y así sucesivamente.

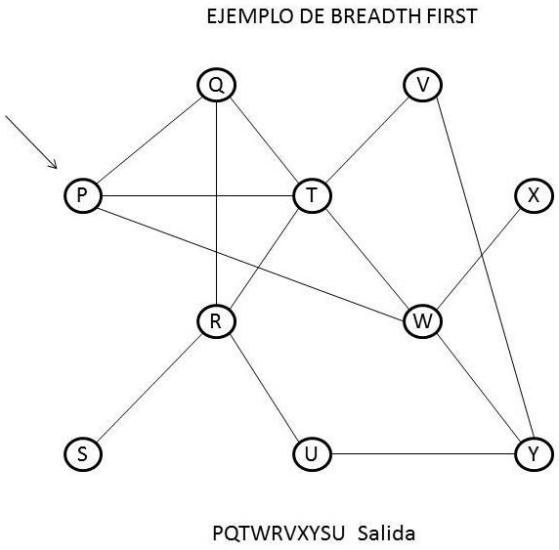


Figura 1.4 Ejemplo del recorrido del algoritmo Breadth First.

Algoritmo: Primero en anchura (breadh first)

Entrada: nodo inicial, nodo buscado

Salida: nodo buscado o mensaje de estado

Algoritmo

-inicializar todos los nodos del grafo en el estado: en espera.

-para cada nodo del grafo:

-si el estado es en espera, entonces:

-insertar el nodo en una fila (cambiar su estado a listo).

-mientras la fila de nodos listos no este vacia:

-sacar un nodo de la fila y procesarlo.

-cambiar su estado a procesado

-meter en la fila todos los nodos vecinos del nodo recién sacado cuyo estado sea en espera. Cambiar el estado a listo.

Búsqueda en profundidad

La búsqueda en profundidad, llamada Deph First Search en inglés, es un algoritmo usado para recorrer o buscar elementos en un árbol o un grafo. Su procedimiento consiste en visitar todos los nodos de forma ordenada pero no uniforme en un camino concreto, dejando caminos sin visitar en su proceso. Una vez llega al final del camino vuelve atrás hasta que encuentra una bifurcación que no ha explorado, y repite el proceso hasta acabar el árbol (esto se conoce como backtracking). Como se muestra en figura 1.5

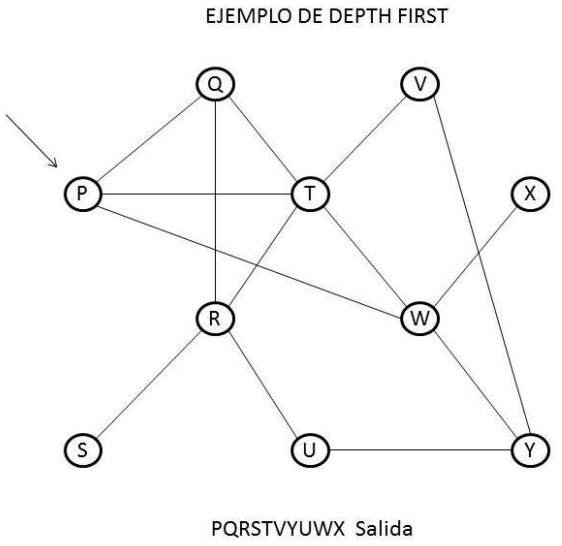


Figura 1.5 Ejemplo Del recorrido del algoritmo Depth First.

Algoritmo: Primero en profundidad (deepth first)

Entrada: nodo inicial, nodo buscado

Salida: nodo buscado o mensaje de estado

Algoritmo

-inicializar todos los nodos del grafo en el estado: en espera.

-para cada nodo del grafo:

-si el estado es en espera, entonces:

-insertar el nodo en una pila (cambiar su estado a listo).

-mientras la pila de nodos listos no este vacía:

-sacar un nodo de la pila y procesarlo.
-cambiar su estado a procesado
-meter en la pila todos los nodos vecinos del nodo recién sacado
Cuyo estado sea en espera o listo.
Cambiar el estado de estos nodos a listo.

1.1.5 Cerradura transitiva en un dígrafo

Otra aplicación muy importante es de los grafos consiste en determinar si para dos nodos cualesquiera de un dígrafo hay una trayectoria que los una. Esta operación es muy útil en problemas de ruteo y conectividad.

El cálculo de la cerradura transitiva se basa en el siguiente concepto: Si hay una trayectoria entre los nodos n i – n j y una trayectoria entre los nodos n j – n k, entonces existe una trayectoria desde n i hasta n k.[3]

Existen varios algoritmos que permiten calcular la cerradura. Sin embargo, uno de los más comunes es el Warshall, que trabaja con la representación del grafo en una matriz de adyacencias. Una versión de este algoritmo es:

Algoritmo: Cerradura transitiva

Si se supone la existencia de un grafo con n nodos vértices:
–Para cada una de las columnas de la matriz (j)
-para cada uno de sus renglones (I)
-Si la casilla A[j,i] es verdadera (hay conexión entre el nodo i y el nodo j) entonces
-para cada una de las columnas de la matriz dentro del renglón i
-si la casilla A[i,k]es verdadera (hay conexión entre el nodo i y el nodo k)
Entonces:
-la casilla A[j,k] es verdadera (existe una trayectoria desde j hasta k).

1.2 Satisfactibilidad (Complejidad computacional)

Es una rama de la computación que se centra en la clasificación de los problemas computacionales de acuerdo a su dificultad inherente y en relación entre dichas clases de complejidad.

Un problema se cataloga como “Inherentemente difícil” si la solución requiere de una cantidad significativa de recursos computacionales, sin importar el algoritmo utilizado. La teoría de la complejidad introduce modelos de cómputo matemático para el estudio de estos problemas y la cuantificación de recursos necesarios para resolverlos, como tiempo y memoria.

Cuando se desarrolla un algoritmo se procede a estimar la cantidad de tiempo de procesador que requiere el algoritmo, así como también la cantidad de memoria, esto se realiza en función del tamaño de la entrada (n) contabilizando la cantidad de operaciones contenidas en el algoritmo y se multiplica por la constante del tiempo que la computadora tarda en realizar una operación cf(n).[2]

En 1965 Jack R. Edmonds definió un “buen” algoritmo como uno con un tiempo de ejecución acotado por un polinomio, es decir con un tiempo de ejecución polinómico. Esto condujo al surgimiento de uno de los conceptos más importantes de la teoría de la complejidad computacional la NP-completitud y su pregunta fundamental, si P=NP. En efecto, uno de los más sorprendentes aspectos de la teoría de complejidad computacional es que una frontera delgada parece separar los problemas *fáciles* de los *difíciles*.

1.2.1 Problemas de decisión

Por conveniencia, la teoría de la complejidad es diseñada especialmente para aplicarse a problemas de decisión. Particularmente se tiene que distinguir entre dos tipos de problemas. Un tipo es el así llamado problema de decisión el cual

requiere de una respuesta si o no y el otro tipo es el problema de optimización que solicita una solución que optimice alguna función objetivo.

En estos términos, un problema de decisión PD puede usarse como medio para reconocer un lenguaje S que es un subconjunto de palabras de un alfabeto D. En la práctica, el PD intenta reconocer algún conjunto específico de objetos matemáticos, tales como: formulas lógicas verdaderas, graficas que tienen una cierta propiedad, etcétera. Pero esto requiere el codificar adecuadamente las estructuras subyacentes de los objetos para cualquier instancia del problema, de tal forma que finalmente, se puede construir un predicado que a un solo con respuestas Si o No obtenga cualquier otra información que se desee. [2]

En forma abstracta, un problema de decisión PD, puede describirse como: PD: < D, S>, donde: D – Dominio de valores posibles y $S \subseteq D$ – son las instancias que resuelven el problemas

El planteamiento, del PD será:

$$\forall\, x\ (x \in D);\ PD(x) = \begin{cases} \text{Si} & \text{si } x \in S \\ \text{No} & \text{si } x \notin S \end{cases}$$

El formato que usaremos para especificar problemas de decisión, consistirá en dos partes; en la primera parte se especifica una instancia genérica del problema en términos de sus componentes o parámetros. Puede verse esta parte como la presentación de una lista de variables libres indicando los valores tipo que cada variable podrá tomar, sean estos conjuntos, graficas, funciones, etcétera. La segunda parte establece una pregunta en términos de la instancia genérica que

espera solo una respuesta de Si o de No. Una instancia particular o muchas veces solo diremos una *instancia*, se obtiene de la instancia genérica por asignar valores particulares del tipo especificado a cada uno de los parámetros del problema.

Por ejemplo el problema del agente viajero el cual consiste en visitar un determinado número de ciudades minimizando la longitud total del recorrido que pasa por cada uno de los vértices de una gráfica sin tocar más de una vez cada vértice, puede plantearse de la siguiente manera:

Instancia: Sea un conjunto finito C de ciudades, donde C = {c1, c2, …cm}, se define una distancia $d(ci,cj) \in Z+$ para todo par de ciudades ci,cj en C, y una cota $B \in Z+$ (Z+ denota los enteros positivos).

Pregunta: ¿Existirá un *recorrido* que visita a todas las ciudades en C sin visitar dos veces una misma ciudad y cuya longitud total del recorrido no sea mayor que B?. En otras palabras, se busca encontrar una permutación Π de {1, …,m} tal que

$$\Sigma_{i=1,\ldots,m-1} d(c_{\Pi(i)},c_{\Pi(i+1)})+d(c_{\Pi(m)},c_{\Pi(1)}) \leq B$$

Un ejemplo de una instancia específica es:

C = {c1,c2,c3,c4}; d(c1,c2) = 5; d(c1,c3) = 5; d(c1,c4) = 8; d(c2,c3) = 6; d(c2,c4) = 12; d(c3,c4) = 9; con cota: B = 29 y la pregunta a resolver seria: ¿Existirá una permutación Π de {1,…,m} tal que: $\Sigma_{i=1\ldots,3} d(c_{\Pi(i)},c_{\Pi(i+1)})+d(c_{\Pi(4)},c_{\Pi(1)}) \leq 29$?

Otro ejemplo de planteamiento de un problema de decisión, lo ilustraremos con el problema de coloración de una gráfica.

Instancia: Una gráfica G = (V,E), donde V es el conjunto de nodos o vértices y E-conjunto de aristas entre los vértices. Una cota $K \in Z+$, con $K \leq |V|$.

Pregunta ¿Sera G k coloreable?, es decir, existirá una función f: V → {1,2,..,k} tal que $f(u) \neq f(v)$ siempre que $(u,v) \in V$.

1.2.2 Problemas de optimización

Muchos de los primeros problemas que al inicio de los 70´s se demostró que eran NP-difíciles, eran problemas de optimización (por ejemplo, los problemas de la cubierta de vértices, la coloración de los vértices de un grafo, el problema del agente viajero, etcétera).

Un problema de optimización combinatoria Π es un problema de minimización o un problema de maximización y se compone de las tres partes siguientes:

1.- un conjunto D de instancias

2.- para cada instancia $I \in D$, un conjunto finito S(I) de soluciones candidatas para I, y

3.- una función m que le asigna a cada instancia $I \in D$ y cada solución candidata $\sigma \in S(I)$ un numero racional positivo $m(I,\sigma)$, llamado el valor de la solución para σ.

Si Π es un problema de minimización (maximización), entonces una solución óptima para una instancia $I \in D$, es una solución candidata $\sigma' \in S(I)$, tal que, para toda otra solución candidata $\sigma \in S(I)$, $m(I, \sigma') \leq m(I, \sigma)(m(I, \sigma') \geq m(I, \sigma))$.

Se usara la notación de OPT(I) para denotar el valor $m(I, \sigma')$ de la solución óptima para I.

Un *algoritmo* es un procedimiento general que trabaja paso a paso y en número finito de estos, resuelve un problema. Un algoritmo resuelve un problema si puede aplicarse a cualquier instancia *I* del problema y garantiza que siempre produce una solución para esta instancia. Podemos pensar en un algoritmo como un programa de computadora o como la descripción de la función de transición de una máquina de Turing.

En general, se desea encontrar un algoritmo más eficiente que resuelva un problema dado. La noción de eficiencia tiene que ver con los varios recursos computacionales necesarios para ejecutar el algoritmo. A un que el recurso dominante es el tiempo, por tal, normalmente el algoritmo más eficiente que resuelve un problema PD es aquel que se ejecuta más rápidamente. [2]

Los requerimientos de tiempo en la ejecución de un algoritmo, se expresan en términos de una sola variable: la longitud o tamaño de la instancia del problema, que refleja la cantidad de datos de entrada y la magnitud de cada uno de estos.

Para definir la longitud de una instancia *I* de un problema de decisión PD, denotado como Long(*I*) necesitamos primero definir la longitud de cada uno de los posibles parámetros que pueden aparecer en la instancia. Así, para todo $n \in Z$, se define su longitud (magnitud) como:

(*) $long(n) = 1 + [\log(|n| + 1)]$.

Donde $|x|$ es la función que da como resultado al entero $z \in Z$ inmediatamente superior a x, y log denota la función logaritmo en base dos. En la expresión (*) el primer 1 indica que se requiere un bit para almacenar el signo de n, por tanto (*) expresa el número de bits necesarios para codificar al número n en binario.

En general, la long(x) de un numero x, será el número de bits que la maquina necesita para representar en su memoria a tal número. En caso de que x sea una cadena de símbolos, si presupone que cada símbolo puede representarse en la memoria de la computadora usando una localidad de memoria, entonces se considera la long(x) = $|x|$ = no. de símbolos que componen a x (incluyendo al carácter blanco como simbolo).

Se extiende el dominio de aplicación de la función long, para considerar diferentes tipos de datos. Por ejemplo, la longitud de un arreglo v con m elementos, donde v[i] se usa para referenciar al elemento i del arreglo, es:

$long(v) = \Sigma_{i=1,\ldots,m} long(v[i])$

Si M es un tipo de dato matriz en Z^{nxm}, entonces:

$long(M) = \Sigma_{i=1,\ldots,n}\Sigma_{j=1,\ldots,m}long(m[i,j])$

Es decir, de acuerdo al tipo de dato a considerar la longitud del dato será la sumatoria de las longitudes de cada uno de sus componentes de manera recursiva, hasta llegar a los componentes escalares.

Dada una instancia *I* de un problema de decisión PD con m datos de entrada; (d1,d2,…,dm) se define la longitud de la instancia como: $long(I) = \Sigma_{i=1,\ldots,m}long(d_i)$ – con long(di) siendo la longitud de cada dato de entrada.

Por ejemplo, si se tuviera una instancia I del problema del agente viajero, se requiere codificar las ciudades y las distancias entre estas. Si una instancia especifica tiene m ciudades, se necesitan m etiquetas para diferenciar las ciudades c_i y m(m-1)/2 números d_j, que podemos suponer enteros positivos, que definen las distancias entre cualquier par de ciudades, y un número entero para indicar la cota B, en total se tendría que:

$long(I) = \Sigma_{i=1,\ldots,m}long(c_i)+\Sigma_{j=1,\ldots,m(m-1)/2}long(d_j)+long(B)$

La función de complejidad de tiempo de un algoritmo, expresa los requerimientos de tiempo que un determinado modelo de computación necesita al ejecutar el algoritmo para resolver cada posible instancia del problema. Suponiendo que las operaciones aritméticas son ejecutadas en tiempo constante, por ejemplo, independiente del tamaño de los números que envuelve. Esto no es correcto en general, pero es posible presuponerlo en este contexto. El modelo de computación básico, para el análisis de las complejidades de tiempo y espacio de los algoritmos, son las llamadas máquinas de Turing.

Un punto clave a observar es que el problema de decisión puede no ser más difícil que su correspondiente problema de optimización. Por ejemplo, si se puede encontrar en tiempo polinomial un recorrido con longitud mínima para el problema del agente viajero PAV, entonces también se puede resolver el problema de decisión asociado en tiempo polinomial.

Por ejemplo, supongamos que existe un algoritmo AP de complejidad polinomial en tiempo para la solución del problema de decisión PAV. Si todos los pesos de las aristas son enteros positivos y el valor mayor de los pesos de las aristas es p y el valor menor es q, entonces claramente la solución óptima estará entre q*n y p*n, siendo n el número de vértices. Aplicando la técnica de búsqueda binaria se puede hallar el recorrido óptimo en a lo más [log(p*n-q*n)] + 1 llamados del algoritmo AP, lo que resulta en un proceso con complejidad polinomial en tiempo.

Ahora, si existe un procedimiento que puede hallar el valor óptimo en tiempo polinomial para el PAV, entonces en particular se puede contestar si existe un recorrido menor a una cota B también en tiempo polinomial, basta con comparar el óptimo hallado contra la cota definidad.

1.3 Forma Normal Conjuntiva (FNC)

A menudo es necesario transformar una fórmula en otra, sobre todo transformar una fórmula a su “forma normal”. Esto se consigue transformando la fórmula en otra equivalente y repitiendo el proceso hasta conseguir la forma deseada.

Se mostraran las reglas necesarias para transformar sintácticamente una fórmula en una forma normal más adecuada para la demostración automática y que conserva la semántica de la fórmula original.

Gracias a las leyes asociativas los paréntesis en (F∨(G∨H)) o en ((F∨G)∨H) pueden eliminarse, es decir, de puede escribir (F∨G∨H). En general se va poder escribir sin ambigüedad una fórmula D= (F1∨F2∨. . . ∨ Fn) donde F1∨F2∨. . .∨ Fn son fórmulas. La fórmula D es cierta cuando lo es al menos una de las Fi, en caso contrario D es falsa. La fórmula D recibe el nombre de disyunción de las fórmulas F1∨F2∨. . . ∨ Fn. De modo análogo se puede escribir C=(F1^F2^. . .^Fn) que es cierta cuando lo son F1, F2,. . . ,Fn, si alguna de las Fi es falsa también lo es C. La

fórmula C se llama conjunción de F1, F2,. . . ,Fn. El orden en que aparecen los Fi es indiferente debido a la ley conmutativa.

Una Fórmula F se dice que está en forma normal conjuntiva si y sólo si F es de la forma F=(F1^F2^. . . ^Fn), donde cada Fi es una disyunción de literales[4].

1.3.1 Átomos y literales:

Definición.- Un átomo es una variable proposicional (p.e. p, q, . . .).
Definición.- Un literal es un átomo o su negación (p.e. p,¬p, q,¬q, . . .).
Notación: L, L1, L2, . . . representarán literales.

1.3.2 Forma normal conjuntiva:

Definición.- Una fórmula está en forma normal conjuntiva (FNC) si es una conjunción de disyunciones de literales; es decir, es de la forma:
(L1,1 ∨ … _ L1,n1) ^… ^(Lm,1 … Lm,nm).
Ejemplos: (¬p ∨ q) ^ (¬q ∨ p) está en FNC.
(¬p ∨ q) ^(q →p) no está en FNC.

Definición.- Una fórmula G es una forma normal conjuntiva (FNC) de la fórmula F si G está en forma normal conjuntiva y es equivalente a F.

Ejemplo: Una FNC de ¬(p ^ (q → r)) es (¬p ∨ q) ^ (¬p ∨ ¬r).

1.3.3 Algoritmo de cálculo de forma normal conjuntiva

Algoritmo: Aplicando a una fórmula F los siguientes pasos se obtiene una forma normal conjuntiva de F, FNC(F):

1. Eliminar los bicondicionales usando la equivalencia

$A \leftrightarrow B \cong (A \rightarrow B) \wedge (B \rightarrow A)$ (1)

2. Eliminar los condicionales usando la equivalencia

$A \rightarrow B \cong \neg A \vee B$ (2)

3. Interiorizar las negaciones usando las equivalencias

$\neg(A \wedge B) \cong \neg A \vee \neg B$ (3)

$\neg(A \vee B) \cong \neg A \wedge \neg B$ (4)

$\neg\neg A \cong A$ (5)

4. Interiorizar las disyunciones usando las equivalencias

$A \vee (B \wedge C) \cong (A \vee B) \wedge (A \vee C)$ (6)

$(A \wedge B) \vee C \cong (A \vee C) \wedge (B \vee C)$ (7)

1.3.4 Ejemplos de cálculo de forma normal conjuntiva

Ejemplo de cálculo de una FNC de $\neg(p \wedge (q \rightarrow r))$:

$\neg(p \wedge (q \rightarrow r)) \cong \neg(p \wedge (\neg q \vee r))$ [por (2)]

$\cong \neg p \vee \neg(\neg q \vee r)$ [por (3)]

$\cong \neg p \vee (\neg\neg q \wedge \neg r)$ [por (4)]

$\cong \neg p \vee (q \wedge \neg r)$ [por (5)]

$\cong (\neg p \vee q) \wedge (\neg p \vee \neg r)$ [por (6)]

Ejemplo de cálculo de una FNC de $(p \rightarrow q) \vee (q \rightarrow p)$:

$(p \rightarrow q) \vee (q \rightarrow p)$

$\cong (\neg p \vee q) \vee (\neg q \vee p)$ [por (2)]

$\cong \neg p \vee q \vee \neg q \vee p$

1.3.5 Cálculo de forma normal conjuntiva

Ejemplo de cálculo de una FNC de $(p \leftrightarrow q) \rightarrow r$:

$(p \leftrightarrow q) \rightarrow r$

$\cong (p \rightarrow q) \wedge (q \rightarrow p) \rightarrow r$

$\cong \neg((p \rightarrow q) \wedge (q \rightarrow p)) \vee r$

$\cong \neg((\neg p \vee q) \wedge (\neg q \vee p)) \vee r$

$\cong (\neg(\neg p \vee q) \vee \neg(\neg q \vee p)) \vee r$

$\cong ((\neg\neg p \wedge \neg q) \vee (\neg\neg q \wedge \neg p)) \vee r$

$\cong ((p \wedge \neg q) \vee (q \wedge \neg p)) \vee r$

$\cong (((p \wedge \neg q) \vee q) \wedge ((p \wedge \neg q) \vee \neg p)) \vee r$

$\cong (((p \vee q) \wedge (\neg q \vee q)) \wedge ((p \vee \neg p) \wedge (\neg q \vee \neg p))) \vee r$

$\cong (((p \vee q) \wedge (\neg q \vee q)) \vee r) \wedge (((p \vee \neg p) \wedge (\neg q \vee \neg p)) \vee r)$

$\cong (((p \vee q) \vee r) \wedge ((\neg q \vee q) \vee r)) \wedge (((p \vee \neg p) \vee r) \wedge ((\neg q \vee \neg p) \vee r))$

$\cong (p \vee q \vee r) \wedge (\neg q \vee q \vee r) \wedge (p \vee \neg p \vee r) \wedge (\neg q \vee \neg p \vee r)$

$\cong (p \vee q \vee r) \wedge (\neg q \vee \neg p \vee r)$

1.3.6 Procedimiento de decisión de validez mediante FNC

Literales complementarios:

El complementario de un literal L es $L^c = \{_p{}^{\neg p}$ si $L = \neg p$ Si $L = :p\}$

Propiedades de reducción de tautologías:

F1 ^… ^Fn es una tautología syss F1, . . . , Fn lo son.

L1 ∨…∨Ln es una tautología syss {L1, . . . , Ln} contiene algún par de literales complementarios (i.e. existen i, j tales que Li = Lc

Algoritmo de decisión de tautologías mediante FNC

Entrada: Una fórmula F.

Procedimiento:

FORMA NORMAL CONJUNTIVA

1. Calcular una FNC de F.

2. Decidir si cada una de las disyunciones de la FNC tiene algún par de literales complementarios.

1.3.7 Ejemplos de decisión de validez mediante FNC

$\neg(p \wedge (q \rightarrow r))$ no es tautología:

$FNC(\neg(p \wedge (q \rightarrow r))) = (\neg p \vee q) \wedge (\neg p \vee \neg r)$

Contramodelos de $\neg(p \wedge (q \rightarrow r))$:

I_1 tal que $I_1(p) = 1$ y $I_1(q) = 0$

I_2 tal que $I_2(p) = 1$ y $I_2(r) = 1$

$(p \rightarrow q) \vee (q \rightarrow p)$ es tautología:

$FNC((p \rightarrow q) \vee (q \rightarrow p)) = \neg p \vee q \vee \neg q \vee p$

$(p \leftrightarrow q) \rightarrow r$ no es tautología:

$FNC((p \leftrightarrow q) \rightarrow r) = (p \vee q \vee r) \wedge (\neg q \vee \neg p \vee r)$

Contramodelos de $(p \leftrightarrow q) \rightarrow r$:

I_1 tal que $I_1(p) = 0$, $I_1(q) = 0$ y $I_1(r) = 0$

I_2 tal que $I_2(p) = 1$, $I_2(q) = 1$ y $I_2(r) = 0$

1.4 Revisión de creencias

Una teoría de revisión de creencias es, ante todo, una estructura formal, a la cual luego pueden darse distintas aplicaciones. La función básica de dicha estructura formal es ofrecer instrucciones acerca de cómo debe cambiar una base de datos cuando nos enfrentamos con nueva información. La nueva información puede entrar en conflicto con la que teníamos almacenada antes, y, en ese caso, si deseamos mantener la consistencia, nos vemos obligados a eliminar algunos elementos previos del sistema. Es deseable que los cambios no se efectúen de cualquier manera, sino de manera racional. Se trata, por tanto, de una teoría normativa que nos indica en cada caso cuál es la manera óptima de proceder.

Los estudios sobre dinámicas de creencias conforman un campo de investigación interdisciplinario, y relativamente nuevo, que se nutre de aportes provenientes de disciplinas tan diversas como la epistemología, la lógica, la inteligencia artificial, y, en menor grado, la psicología cognitiva. Una teoría de revisión de creencias es, ante todo, una estructura formal, a la cual luego pueden darse distintas aplicaciones. La función básica de dicha estructura formal es, dicho muy sucintamente, ofrecer instrucciones acerca de cómo debe cambiar una base de datos cuando nos enfrentamos con nueva información. La nueva información puede entrar en conflicto con la que teníamos almacenada antes, y, en ese caso, si queremos mantener la consistencia, nos vemos obligados a eliminar algunos elementos previos del sistema. Ahora bien, es deseable que los cambios no se efectúen de cualquier manera, sino de manera racional. Se trata, por tanto, de una teoría normativa que nos indica en cada caso cuál es la manera óptima de proceder.

En cierto sentido el nombre “revisión de creencias” no es demasiado feliz. Las “creencias” a las que refiere la teoría no son, en primera instancia, sino elementos de una estructura formal, a los que después podemos interpretar de diferente manera. Dichas “creencias”, pues, pueden aludir en última instancia tanto a

entidades mentales de un sujeto real, como a elementos de una base de datos en una computadora, o a hipótesis de una teoría científica (si es que queremos aplicar este modelo al cambio teórico en la ciencia), o tal vez a elementos de un problema de teoría de la decisión. Justamente, uno de los rasgos interesantes de las teorías de revisión de creencias es su versatilidad. Como corresponde a una propuesta surgida en un terreno interdisciplinario, podemos buscar en ella respuestas a cuestiones estrictamente filosóficas y teóricas, pero a la vez constituye una herramienta con múltiples aplicaciones posibles. Retomaré este punto más adelante. Entre los pioneros en estudios sobre dinámicas epistémicas debemos mencionar sin dudas a Isaac Levi, quien a fines de los años sesenta sentó las bases conceptuales de lo que más adelante sería una teoría axiomatizada. La axiomatización más famosa, y ya clásica, se debe básicamente a (Carlos Alchourrón, Peter Gärderfors, y David Makinson, 1985; también Gärdenfors, 1988). Dicha axiomatización, bautizada “teoría AGM” por las iniciales de los tres autores, es considerada hoy en día como la posición estándar u ortodoxa en dinámica de creencias (y punto de referencia obligado de todo estudio sobre el tema). En la próxima sección delinearé, muy esquemáticamente, los aspectos más importantes de AGM.

Capítulo 2. Modelo de detección de Clausulas no satisfactibles

Los algoritmos son parte fundamental de los programas ya que indican paso a paso que actividades realizar para la resolución de un problema, inicia con los datos de entrada, datos de salida que es propiamente la resolución del problema y posteriormente viene lo que es el algoritmo, en cada paso realiza una actividad para obtener un resultado, un algoritmo se compone de un inicio y final y cada paso se numera para seguir un orden meticuloso. Una vez obtenido el algoritmo se realiza su correspondiente diagrama de flujo, que es la diagramación del algoritmo, este diagrama permite analizar la forma que fluye el flujo de información en el programa. Por último se prueba la valides del algoritmo utilizando para ello la prueba de escritorio que consiste en dar un seguimiento al algoritmo utilizando valores arbitrarios para los datos de entrada, se toman las variables y de acuerdo a las instrucciones se realizan las operaciones correspondientes al finalizar el algoritmo comparamos los resultados obtenidos con los resultados esperados.

2.1 Planteamiento del problema

El presente trabajo pretende mostrar cómo se evalúa una base de conocimiento (clausulas) cuando se expone a nueva información la base de conocimiento, esta información puede estar: 1) acorde con la información de la base de conocimiento o bien 2) estar en contradicción con la base de conocimiento. La información nueva que se adquiere se debe de priorizar y se debe incorporar a la base de conocimiento, es así como se plantea un método para validar la base de conocimiento, es decir, se determina si la base de conocimiento formada por clausulas en 2FNC es satisfactible o no. El presente trabajo se limitara a leer una base de conocimiento almacenada en un archivo de texto y evaluara las clausulas para determinar si la base de conocimiento es consistente o no es consistente (lo

que permite determinar si la base es satisfactible o no) utilizando la fórmula de la cerradura transitiva.

2.2 Algoritmo para la evaluación de una base de conocimiento

En las siguientes imágenes observaremos el funcionamiento del algoritmo para el ejemplo

F= {(-x3,x1)(x1,-x2)(x1,x3)(-x2,x3)(-x1,-x3)}

Esta función contiene 5 clausulas, 5 variables y en la imagen 2.1 podemos observar la forma de almacenarse la clausulas en un archivo simple de block de notas.

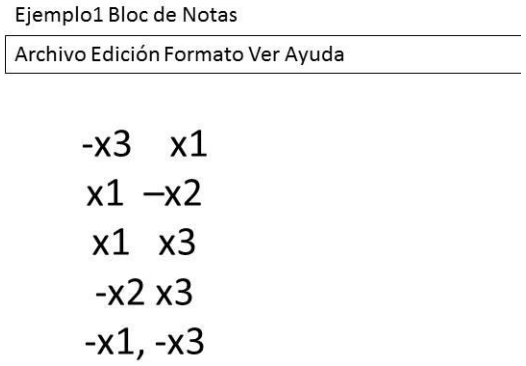


Figura 2.1 La imagen muestra el archivo de block de notas.

El algoritmo lee el archivo y coloca las clausulas en un arreglo llamado arreglouno, también genera el arreglodos que contiene el arreglouno al cual se le aplico la formula (a,b) $\rightarrow$ (-a,b)(-b,a) y por ultimo arreglotres en el cual se mapean las variables del arreglouno como lo muestra la figura 2.2.

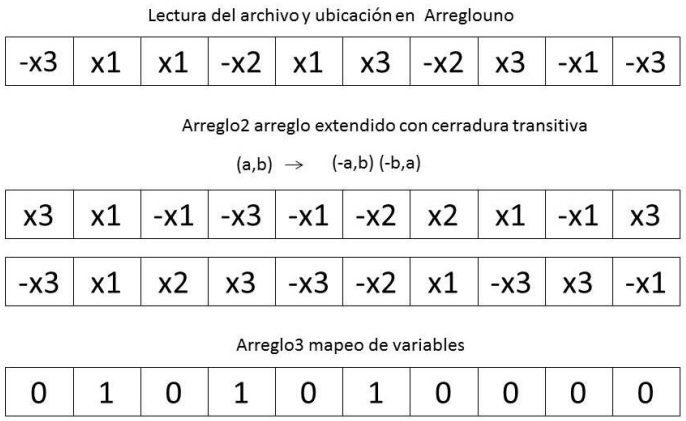


Figura 2.2 La imagen muestra los contenidos de arreglo uno, arreglo dos y arreglo tres.

Una vez mapeadas las variables y obtenido el arreglo extendido se identifican los Tx en el arreglodos y se almacenan en una lista de adyacencias como se muestra en la figura 2.3

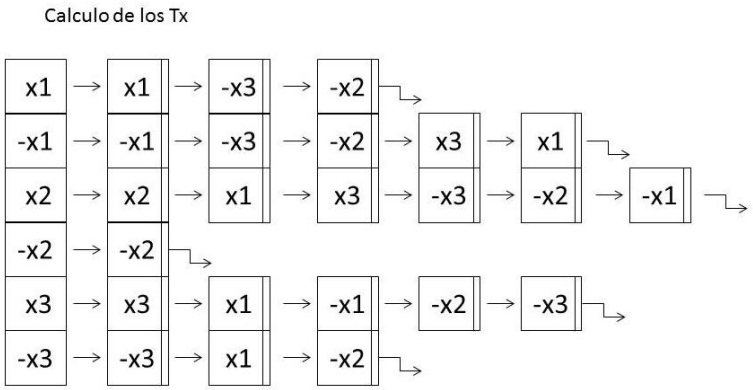


Figura 2.3 muestra la ubicación de los Tx en una lista de adyacencias.

En la figura 2.4 se muestra el flujo principal de la aplicación desarrollada, donde se recibe el archivo de entrada, se transforma cada clausula utilizando la cerradura transitiva y se almacenan en una lista ligada para cada literal xi y –xi. Después se realiza el cálculo de los TX's para poder determinar si las literales tiene o no

contradicción para lo cual se mapean en un arreglo unidimensional de ceros y unos, aquí si para alguna xi tiene contradicción entonces se marca con un uno y tendremos que esa literal es inconsistente. Si en este arreglo se tienen un uno para la variable xi y su negación, entonces la formula no es satisfactible y por tanto la base de conocimiento no es consistente.

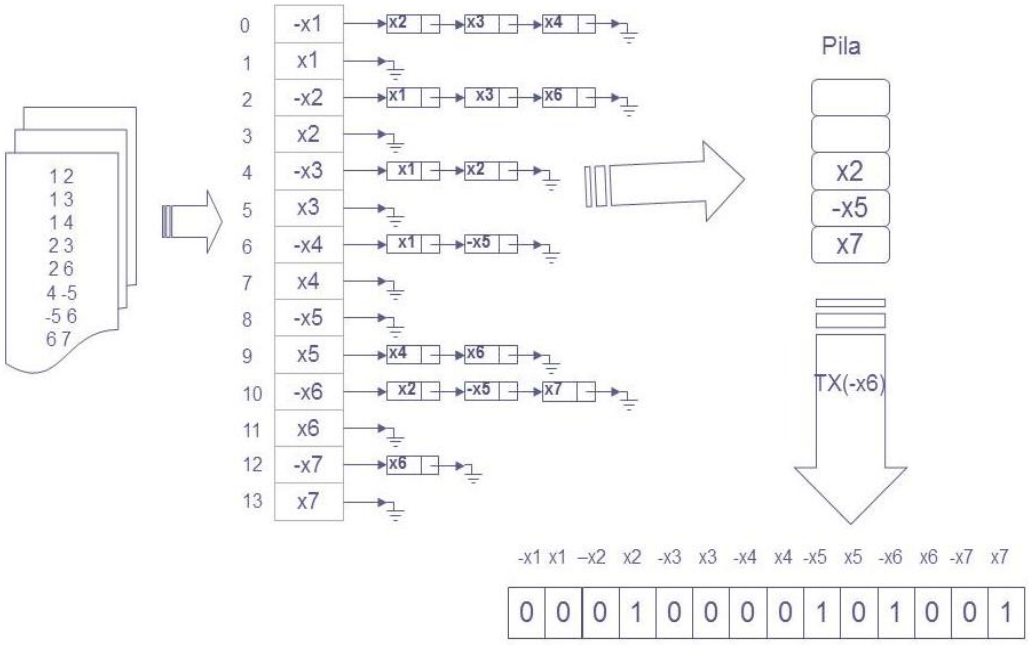


Figura 2.4 Imagen que muestra el aspecto visual de un grafo.

El algoritmo principal del cálculo de la satisfactibilidad de la formula 2FNC mapeada como una base de conocimiento se muestra en la figura 2.5, donde se recibe como entrada un archivo con los datos de la formula (base de conocimiento) y se transforma dicha fórmula usando la cerradura transitiva, el punto central del proceso es el cálculo de los TX´s para cada literal xi.

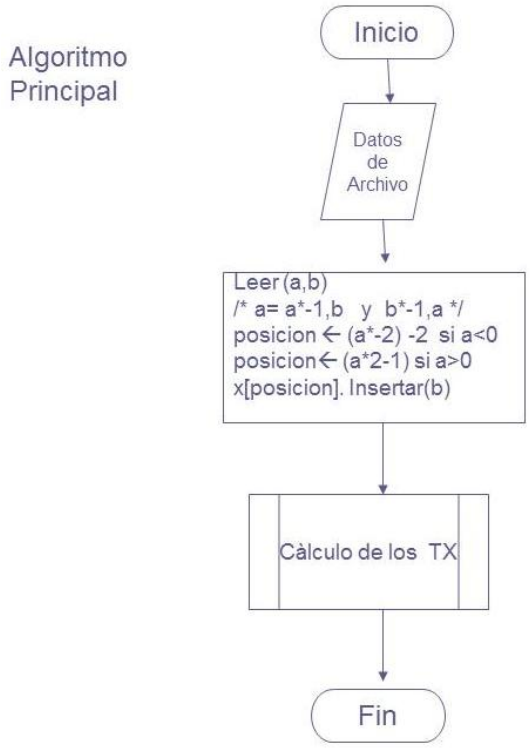


Figura 2.5 Algoritmo del programa

El punto central del proceso de revisión de la consistencia de la base de conocimiento es el cálculo de los así llamados conjuntos TX´s (ver figura 2.6), tomando como base la lista ligada donde se encuentra almacenada la formula transformada usando la cerradura transitiva, se almacena en una lista ligada auxiliar cada TX[x_i], donde se inserta x_i y se recorre la lista ligada teniendo como origen x_i y se agrega cada literal destino x_j que no esté ya en el conjunto TX[x_i], de forma recurrente se agrega cada x_j destino al conjunto TX[x_i], si en este conjunto aparece x_i y $-x_i$ (x_i negada) entonces este conjunto TX[x_i] es inconsistente y se marca con un uno. Si su correspondiente TX[$-x_i$] resulta inconsistente entonces se marca con un uno. Al finalizar el cálculo de los TX's y se revisa si TX[x_i]=1 y TX[$-x_i$]=1 entonces la base de conocimiento es no satisfactibe, en caso contrario si alguno de los dos TX's es igual a cero, entonces la base de conocimiento es satisfactible.

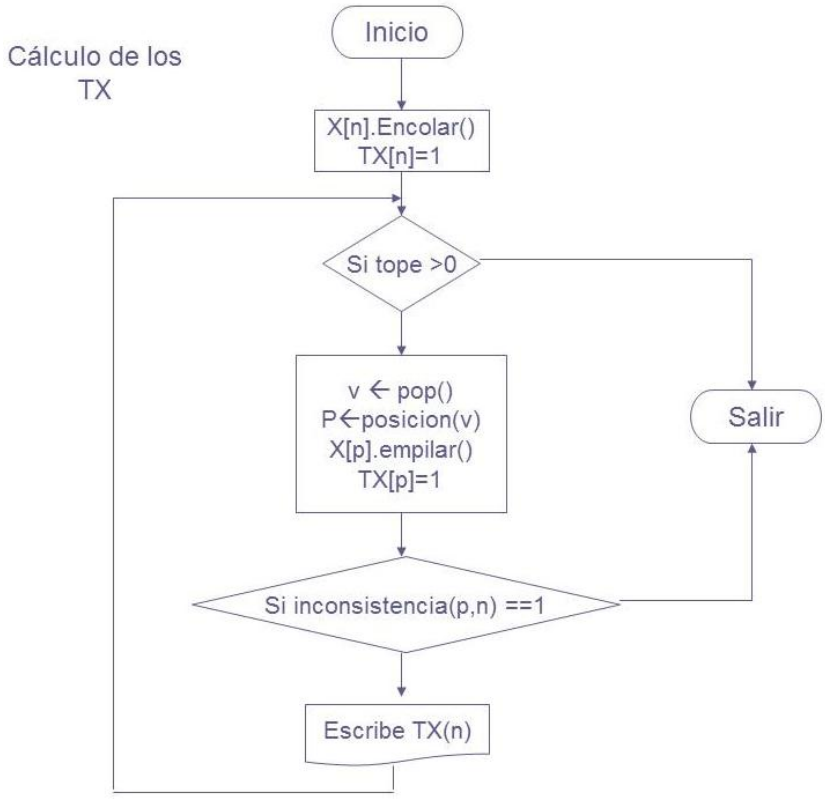


Figura 2.6 continuación del algoritmo

Código principal de la aplicación para determinar si la base de conocimiento es satisfactible

```
public static void main(String[] ar)
{
        int sat;
  Sat T=new Sat();
  System.out.println("Clausulas: ");
  T.Lee("exe.txt"); //Archivo de entrada
  T.escribeClausulas();
  System.out.println("_____");
  System.out.println("Listas ligadas: ");
  T.mostrarListas(T.L);
  System.out.println("_____");
  System.out.println("Calculo de TX's: ");*/
  sat=T.calculaTX();
 /T.mostrarTX(T.TX);
  System.out.println("_____");
```

```
   if(sat==0)System.out.println("La Base de conocimiento es SATISFACTIBLE");
   else System.out.println("La Base de conocimiento NO es SATISFACTIBLE");
 }
```

El código anterior representa código en java de la aplicación desarrollada donde se recibe como entrada un archivo con los datos de la base de conocimiento, se almacena en una lista ligada para posteriormente realizar el cálculo de los TX's, una vez obtenido los TX's se determina si la base de conocimiento es satisfactible o no, es decir consistente.

Código para el cálculo del conjunto de TX`s

```
public int calculaTX()
{
        int n,lit,var,x,i;
        int tmp[]=new int[maxvar];

        LinkedList <Integer> cola=new LinkedList<Integer>();
        for(x=0;x<variables;x++)
        {
                for(i=0; i<variables; i++)tmp[i]=0;
                var=posicionInv(x);
                cola.addFirst(var);
                while (cola.size()!=0)
                {
                        lit=cola.removeLast();
                        n=posicion(lit);
                        if (tmp[n]!=1){
                                        TX[x].insertar(lit);
                        tmp[n]=1;
                         }
                        Nodo aux = L[n].raiz;
                while (aux!=null)
                {
                        n=posicion(aux.info);
                        if(tmp[n]!=1){
                             TX[x].insertar(aux.info);
                           cola.addFirst(aux.info);
                        tmp[n]=1;
                         }
                        aux=aux.liga;
                }
```

```
                }
            for(i=0;i<variables-1;i=i+2)
                if(tmp[i]==1 && tmp[i+1]==1)consistentes[x]=1;
        }

  for(i=0; i<variables; i=i+2)
  {
        if(consistentes[i]==1 && consistentes[i+1]==1) return 1;
  }
  return 0;
 }
```

El código ubicado en la parte superior representa código java también y a través de comparaciones primero identifica cuantos Tx´s tiene la lista ligada y posteriormente procede a localizarlos y almacenarlos también en otra lista ligadas. Una vez obtenidos calculados los Tx´s se determina si los Tx´s son satisfactibles o no satisfactibles.

Capítulo 3. Resultados Obtenidos

A continuación se muestran los resultados obtenidos de la aplicación desarrollada. Esta aplicación fue desarrollada utilizando el lenguaje de programación Java utilizando la técnica de programación orientada a objetos, donde la clase principal (ver figura 3.1) denominada sat utiliza las estructuras de datos cola, lista ligada y nodo.

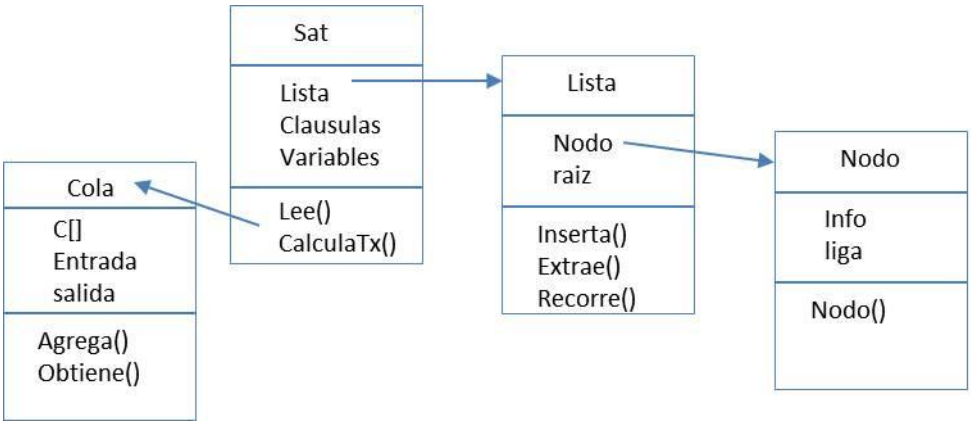


Figura 3.1. Esquema general de las clases desarrolladas en la aplicación

3.1 Ejemplo de la consistencia de una base de conocimiento

Considere la siguiente base de conocimiento

F = {(-x3,x1)(x1,-x2)(x1,x3)(-x2,x3)(-x1,-x3)}

A la formula F se le aplica el método de la cerradura transitiva (a,b) $\rightarrow$ (-a,b)(-b,a)

Obteniendo:

F={(x3,x1)(-x1,-x3)(-x1,-x2)(x2,x1)(-x1,x3)(-x3,x1)(x2,x3)(-x3,-x2)(x1,-x3)(x3,-x1)}

Ahora el calculamos los Tx´s

T[x1]={x1,-x3,-x2}

T[-x1]={-x1,-x3,-x2,x3,x1} inconsistente

T[x2]={x2,x1,x3,-x3,-x2,-x1} inconsistente

T[-x2]={-x2}

T[x3]={x3,x1,-x1,-x2} inconsistente

T[-x3]={-x3,x1,-x2}

Ahora se procede a evaluar la satisfactibilidad y observamos que T[-x1], T[x2] y T[x3] son inconsistentes, por lo tanto F es satisfactible.

Podemos observar el digrafo de implicaciones de F = {(-x3,x1)(x1,-x2)(x1,x3)

(-x2,x3)(-x1,-x3)} en la figura 3.2 y verificar que no hay trayectoria de x1 a –x1, x2 a –x2 y de x3 a –x3 para ningún nodo del grafo.

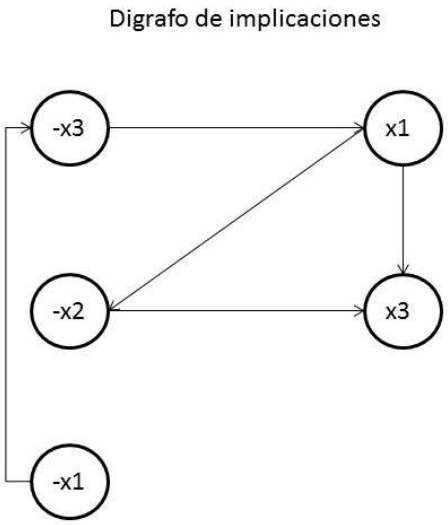


Figura 3.2 muestra el digrafo de implicaciones para F

3.2 Eliminación de clausulas

De acuerdo a la teoría AGM si se desea eliminar una clausula deberá privilegiarse la información más reciente y eliminar las clausulas más anteriores, consideremos el siguiente ejemplo

F = {(-x2,x3)(x2,x1)(x1,x2)(-x3,x3)(x3,-x1)(x2,-x2)(x2,x3)(-x1,x2)(x1,-x1)(-x2,-x3)}

Sobre esta función se deberá aplicar el algoritmo y verificar si es consistente o inconsistente.

A F se le aplica el método de la cerradura transitiva (a,b) $\rightarrow$ (-a,b)(-b,a)

FE={(x2,x3)(-x3,-x2)(-x2,x1)(-x1,x2)(-x1,x2)(-x2,x1)(x3,x3)(-x3,-x3)(-x3,-x1)(x1,x3)
(-x2,-x2)(x2,x2)(-x2,x3)(-x3,x2)(x1,x2)(-x2,-x1)(-x1,-x1)(x1,x1)(x2,-x3)(x3,-x2)}

Continuamos con el cálculo de los T[x]

T[x1] = {x1,x3,-x2,-x1,x2}

T[-x1] = {-x1,x2,x3,-x2,x1,}

T[x2] = {x2,x3,-x2,x1,x3,-x2,-x1}

T[-x2] = {-x2,x1,x3,-x1,x2,-x2}

T[x3] = {x3,-x2,x1,x2,-x3}

T[-x3] = {-x3,-x2,x1,x3,-x1,x2}

Podemos concluir que este ejemplo no es satisfactible, dado que en tx hay elementos del tipo –txi y txi, y esa expresión es contradictoria. De acuerdo a la revisión de creencias se debe privilegiar la información nueva y eliminar las clausulas más anteriores, por lo tanto en la formula anterior se eliminara la primera clausula

(-x2,x3) y F queda de la siguiente manera

F = {(x2,x1)(x1,x2)(-x3,x3)(x3,-x1)(x2,-x2)(x2,x3)(-x1,x2)(x1,-x1)(-x2,-x3)}

A F se le aplica el método de la cerradura transitiva (a,b) $\rightarrow$ (-a,b)(-b,a)

FE={(-x2,x1)(-x1,x2)(-x1,x2)(-x2,x1)(x3,x3)(-x3,-x3)(-x3,-x1)(x1,x3)(-x2,-x2)(x2,x2)

(-x2,x3)(-x3,x2)(x1,x2)(-x2,-x1)(x1,x1)(x1,x1)(x2,-x3)(x3,-x2)}

Continuamos con el cálculo de los T[x]

T[x1] ={x1,x3,-x2,-x1,x2,-x3}

T[-x1] = {-x1,x2,-x3}

T[x2] ={x2,-x3,-x1}

T[-x2] = {-x2,x1,x3,-x1,x2,-x3}

T[x3] ={x3,-x2}

T[-x3] = {-x3,-x1,x2,}

Como los Tx obtenidos no son contradictorios podemos decir que F es satisfactible, observamos el grafo de implicaciones en la figura 3.3 si analizamos el grafo podemos ver que existen trayectorias de los nodos xi a –xi , si se elimina esas trayectorias del grafo se eliminaran los caminos de xi a -xi y el ejemplo será satisfactible.

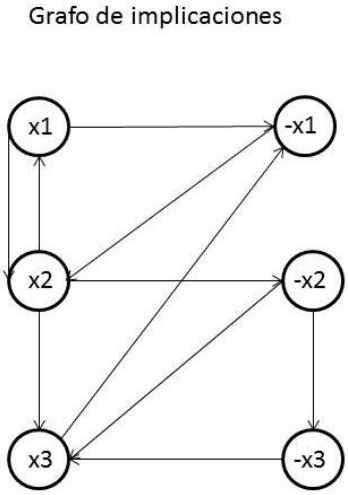


Figura 3.3 grafo de implicaciones para F

Ahora probamos este ejemplo en el programa, para lo cual capturamos las clausulas en WordPad como se muestra en la figura 3.4

Ejemplo2- WordPad

Archivo Edición Formato Ver Ayuda

-X2 X3
X2 X1
X1 X2
-X3 X3
X3 –X1
X2 –X2
X2 X3
-X1 X2
X1 -X1
-X2 -X3

Figura 3.4 muestra las clausulas en un archivo de texto

Ahora vamos a ejecutar el programa con el archivo de texto previamente editado en el block de notas y observamos el resultado del programa en la figura 3.5

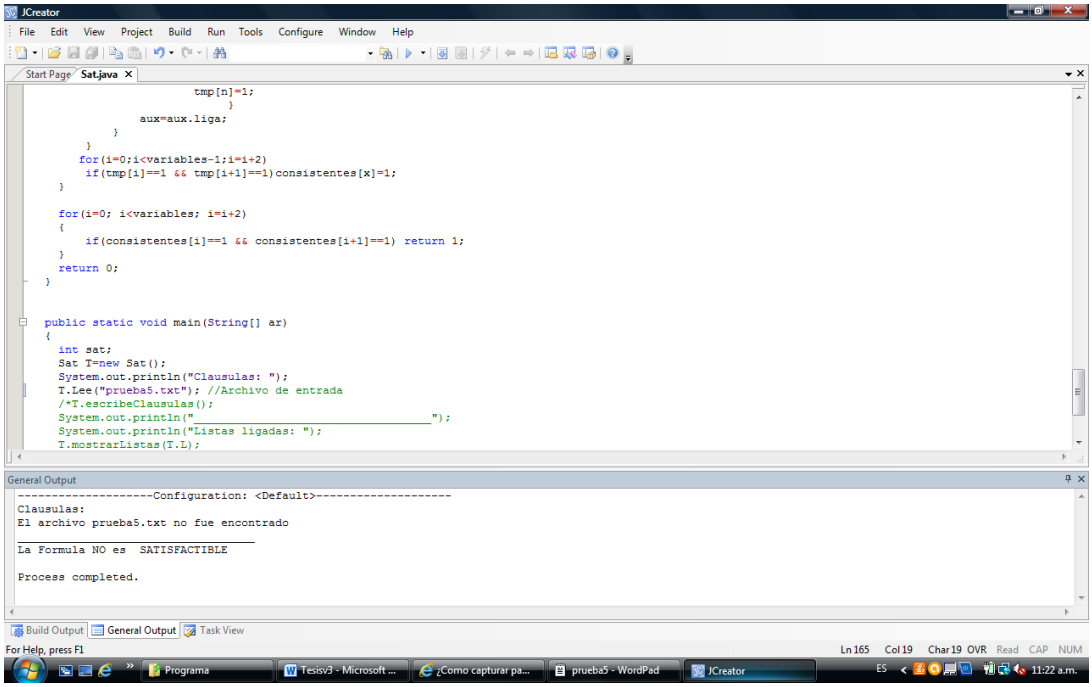


Figura 3.5 el programa muestra el resultado del archivo ejemplo5.

De acuerdo al ejercicio desarrollado previamente muestra el resultado y observamos que no es satisfactible. Ahora procedemos a eliminar la cláusula

(-2x,3x) del archivo ejemplo5.txt y ejecutamos nuevamente el programa para ver el resultado en la figura 3.6

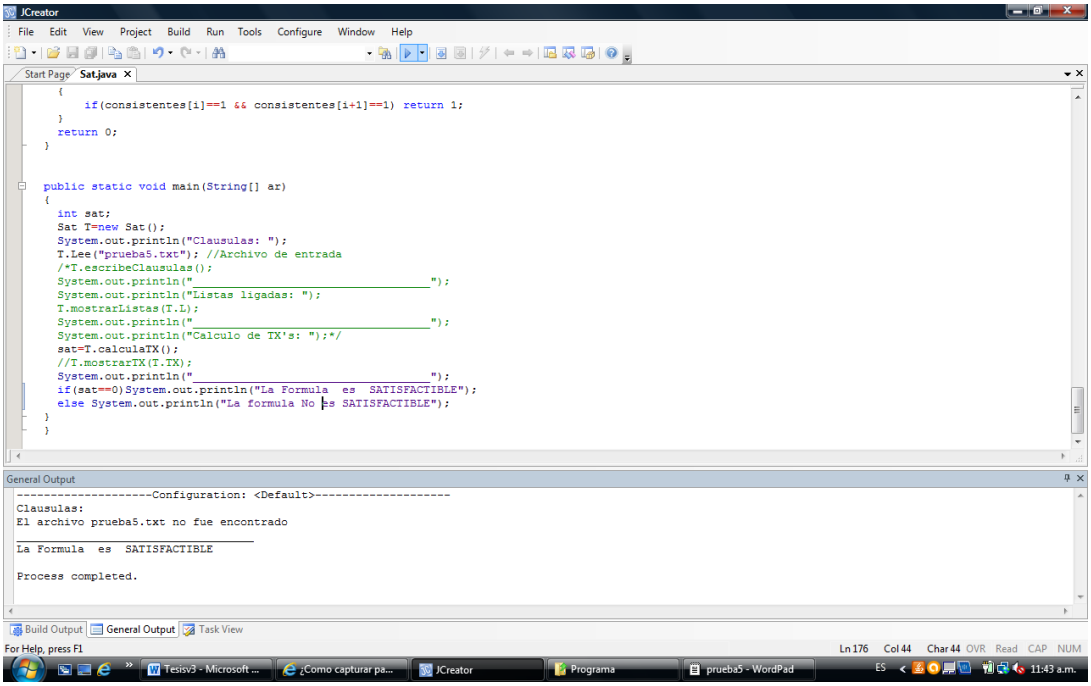


Figura 3.6 muestra el resultado de ejecución del programa, eliminando la cláusula (-2x,3x)

Y podemos observar que el programa funciona adecuadamente.

3.2Tablas comparativas

Se procedió a probar el programa ejecutando valores para 50 variables con 50 cláusulas, posteriormente aumentamos el número de cláusulas para observar el comportamiento del programa como se muestra en la tabla 3.6

Tabla 3.6 muestra los resultados del programa para diferentes números de variables y clausulas

Variables	Clausulas	Satisfactible	Solución
50	50	Si	
50	60	Si	
50	70	Si	
50	80	No	{(x37, -x2)}
50	90	No	{(x14, -x44)}
50	100	No	{(x24, -x35), (-x32, -x21)} {(x24, -x35), (x32, -x17)}
100	50	Si	
100	100	Si	
100	150	No	{(x97, -x50)} {(x99, x33)} {(x46, -x97)} {(-x46, x90)} {(-x43, -x5)} {(x5, x50)} {(-x90, -x33)}
100	160	No	{(-x84, x51)} {(x58, x97)} {(x84, x77)} {(-x58, -x77)} {(x84, -x97)}
100	170	No	{(-x48, -x8)} {(-x48, -x60)} {(x8, x83)} {(-x2, -x96)} {(x48, x96)} {(x60, -x83)}

100	180	No	{(-x12, x45) (-x31, x41) (x42, -x19)}
100	190	No	{(-x23, -x50) (-x62, x41) (x75, x57)} {(-x23, -x50)(-x62, x41) (-x24, x35)}
100	200	No	{(x97, -x50)} {(x99, x33)} {(x46, -x97)} {(-x46, x90)} {(-x43, -x5)} {(x5, x50)} {(-x90, -x33)}
200	100	Si	
150	150	Si	
150	200	No	{(-x72, -x57)} {(x120, x91)} {(-x90, -x91)} {(-x5, -x77)} {(x125, -x116)} {(x5, -x118)} {(x41, -x53)} {(-x120, x116)} {(x109, x57)} {(x118, x72)} {(-x125, x53)} {(-x41, -x109)}
200	200	Si	
500	500	Si	

En la tabla 3.6 se muestran los resultados obtenidos con diferentes archivos de texto que se introdujeron al programa con diferentes números de variable, números de cláusulas y nos muestra si la base del conocimiento es consistente o no consistente, para tal caso se propone eliminar o remover un conjunto de cláusulas a fin de que la base de conocimiento sea consistente, se puede observar que cuando el número de cláusulas es mayor que el número de variables es más probable que la base de conocimiento no sea satisfactible.

Conclusiones

En general el determinar si una formula 2SAT es satisfactible se puede procesar en tiempo lineal o polinomial y en el caso del algoritmo desarrollado procesa tres ciclos anidados de orden máximo n (número de variables) para el cálculo de los TX's asi que este algoritmo en forma general es acotado por un polinomio de grado 3. Respecto al manejo de memoria, dependiendo del número de variables de las clausulas, los almacenamientos de los arreglos arreglouno, arreglodos y arreglotres se mantiene en el orden lineal las localidades de memoria para el manejo de datos. De acuerdo a la clasificación de los algoritmos hecha por G. Brassard y Bratley Paul este algoritmo seria de tipo P.

Este trabajo integra el uso de estructuras de datos con la programación orientada a objetos para determinar si una base de conocimiento es consistente (satisfactible para la formula) o no.

El método utilizado en este trabajo de tesis para determinar si una fórmula es satisfactible o no puede ser extendido y aplicado a problemas de revisión de creencias agregándole el hecho de eliminar la cláusula más antigua para el caso de que la formula no sea satisfactible.

Perspectivas de mejora

Una de las principales mejoras es implementar un módulo que permita obtener que clausula o que clausulas se deben eliminar para lograr que la base de

conocimiento sea satisfactible, que si bien incrementara el tiempo de procesamiento, también garatizará que cualquier base de conocimiento pueda ser tratada como satisfactible.

Extender el uso de esta aplicación para que sea totalmente dinámica y pueda manipular bases de conocimiento de mayor tamaño que las utilizadas en este trabajo para modelar el problema.

Bibliografía

[1] Jaime Sisa Alberto. Estructura de datos y Algoritmos. Editorial Prentice Hall, México 2010.

[2] Gilles Brassard, Bratley Paul. Fundamentos de Algoritmia. Editorial Pearson educación, México 1997.

[3] Martínez Román, Quiroga Elda. Estructura de datos: referencia practica con orientación a objetos. Editorial Thomson Learning. México 2002

[4] García Osorio Cesar Ignacio. Formas Normales. Universidad de Burgos

[5] Luna Carlos Daniel. Una Generalización del Modelo AGM de Cambio de Creencias. Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial, año/vol. 5 número 013. Valencia España pp23-32 (2001).

[6] Fermé Eduardo. Lógicas no Monótonas para Modelos Alternativos de Revisión de Creencias. Universidad de Buenos Aires Argentina (1985).

[7] Eiter, T., Gottlob, G.: On the Complexity of Propositional Knowledge Base Revision, Updates, and Counterfactuals, Artificial Intelligence, 57(2-3). 227270(1992)

[8] LANMR2012, Ceur-ws.org/vol-911/Logic/Lenguages, Algorithms and new method of reasoning 2012. Meliza Contreras, Miguel Rodriguez, Pedro Bello, Barbara Gonzalez: An Introduction to belief revision and knowledge representation with 2CNF. Seveth Latin American Worshop Logic/Langauges, Algoritmos and NewMethods of Reasoning 2011. Toluca, Edo. de México.

Anexo 1: Código de la aplicación desarrollada

Clase Nodo:

```
class Nodo
{
  public int info;
  public Nodo liga;
  public Nodo(int x)
   {
     info = x;
     liga = null;
    }
}
```

Clase Lista:

```
class Lista
{
  Nodo raiz;
 public Lista()
            {
                        raiz = null;
            }

 public void insertar(int x)
  {
            Nodo aux;
    Nodo temp = new Nodo(x);
    if(raiz == null) raiz=temp;
    else{
            aux=raiz;
        while(aux.liga!=null)
          aux=aux.liga;
        aux.liga= temp;
       }
  }


 public int extraer()
  {
            Nodo aux=raiz;
```

```
            raiz=raiz.liga;
            aux.liga=null;
    return(aux.info);
  }


  public void mostrarLista()
  {
    Nodo aux = raiz;
    while (aux!=null)
    {
            System.out.print(aux.info+"->");
            aux=aux.liga;
    }
    System.out.println();
  }
```

Clase Sat:

```
class Sat
{
 int clausulas;
 int variables;
 static final int maxvar=1000;
 Lista L[] = new Lista[maxvar];
 Lista TX[] = new Lista[maxvar];
 int consistentes[]=new int[maxvar];

 public Sat()
 {
            clausulas=0;
            variables=0;
            for(int i=0;i<maxvar;i++)
            {
     L[i]=new Lista();
     TX[i]=new Lista();
            }
 }

 public void escribeClausulas()
 {
   System.out.println("Clausulas = "+clausulas);
   System.out.println("Variables = "+variables/2);
 }

 public static int posicion(int n)
{
```

```
 if (n<0) return((n*-2)-2);
   else  return(n*2-1);
}

public static int posicionInv(int n)
{
 if (n%2==0) return((n+2)/-2);
   else  return((n+1)/2);
}

 public void Lee(String arch) //Lee formula de archivo
  {
                FileInputStream fp;
                DataInputStream f;
                String linea = null;
                int token1,token2,x,y,x2,y2,n,max=1;
 try
    {
                fp = new FileInputStream(arch);
                f = new DataInputStream(fp);
                do {
                        linea = f.readLine();
                        if (linea!=null){
                                        StringTokenizer tokens=new
StringTokenizer(linea);
                                        token1 = Integer.parseInt(tokens.nextToken());
                                        token2 = Integer.parseInt(tokens.nextToken());
                                        // escribimos en pantalla los datos leidos
transformados en numeros
                                        System.out.println("("+token1+" "+token2+")");
                                        clausulas++;
                                        x=token1*-1;
                                        y=token2;
                                        x2=token2*-1;
                                        y2=token1;
                                        n=posicion(x);
                                        L[n].insertar(y);
                                        n=posicion(x2);
                                        L[n].insertar(y2);
                                        if (max<Math.abs(token1))
max=Math.abs(token1);
                                                        if (max<Math.abs(token2))
max=Math.abs(token2);
                                }
                        }while(linea != null);
                fp.close();
        }
```

```
catch (FileNotFoundException exc)
    {
       System.out.println ("El archivo " + arch + " no fue encontrado ");
    }

     catch (IOException exc)
     {
        System.out.println (exc);
     }
     variables=max*2;
 }


 public void mostrarListas(Lista X[])
 {
         int var;
         for(int i=0; i<variables;i++)
         {
                 var=posicionInv(i);
                 if(var<0)System.out.print("L[-x"+var*-1+"]=");
                 else System.out.print("L[x"+var+"]=");
                 X[i].mostrarLista();
         }
 }

 public void mostrarTX(Lista X[])
 {
         int var;
         for(int i=0; i<variables;i++)
         {
                 var=posicionInv(i);
                 if(var<0)System.out.print("TX[-x"+var*-1+"]=");
                 else System.out.print("TX[x"+var+"]=");
                 X[i].mostrarLista();
                 if(consistentes[i]==1) System.out.println("inconsistente");
         }
 }


 public int calculaTX()
 {
         int n,lit,var,x,i;
         int tmp[]=new int[maxvar];

         LinkedList <Integer> cola=new LinkedList<Integer>();
         for(x=0;x<variables;x++)
         {
                 for(i=0; i<variables; i++)tmp[i]=0;
```

```
                var=posicionInv(x);
                cola.addFirst(var);
                while (cola.size()!=0)
                {
                        lit=cola.removeLast();
                        n=posicion(lit);
                        if (tmp[n]!=1){
                                        TX[x].insertar(lit);
                        tmp[n]=1;
                         }
                        Nodo aux = L[n].raiz;
                while (aux!=null)
                {
                        n=posicion(aux.info);
                        if(tmp[n]!=1){
                             TX[x].insertar(aux.info);
                           cola.addFirst(aux.info);
                        tmp[n]=1;
                         }
                        aux=aux.liga;
                }
                }
          for(i=0;i<variables-1;i=i+2)
                if(tmp[i]==1 && tmp[i+1]==1)consistentes[x]=1;
        }

  for(i=0; i<variables; i=i+2)
  {
        if(consistentes[i]==1 && consistentes[i+1]==1) return 1;
  }
  return 0;
}
```

Código principal de la aplicación para determinar si la base de conocimiento es satisfactible

```
 public static void main(String[] ar)
 {
        int sat;
  Sat T=new Sat();
  System.out.println("Clausulas: ");
  T.Lee("exe.txt"); //Archivo de entrada
  T.escribeClausulas();
  System.out.println("_____");
  System.out.println("Listas ligadas: ");
  T.mostrarListas(T.L);
```

```
 System.out.println("_____");
 System.out.println("Calculo de TX's: ");*/
 sat=T.calculaTX();
/T.mostrarTX(T.TX);
 System.out.println("_____");
 if(sat==0)System.out.println("La Base de conocimiento es SATISFACTIBLE");
 else System.out.println("La Base de conocimiento NO es SATISFACTIBLE");
```