Técnicas de optimización de Árboles Binarios de Decisión para implementación de máquinas de estados

Trabajo fin de máster del
Máster en Investigación en Ingeniería del Software
de la Universidad de Educación a Distancia

Diego J. Romero López

Índice

Agradecimientos

Agradezco al Prof. Dr. José Luis Bernier el haberme dado la oportunidad de trabajar en este trabajo fin de máster.

Por último, también agradezco a mi familia el haberme apoyado de multitud de formas a lo largo de todo el máster y en especial, este trabajo que supone el colofón del mismo.

Introducción

En general, cuando se piensa en un sistema tolerante a fallos, se suele pensar en sistemas software que tienen vidas a su cargo, como software para aviónica o para centrales nucleares, pero, supongamos por un momento un sistema web que gestione ocupaciones de reserva de plazas para un lavadero de coches. Desde el momento que el cliente inicia su compra, está reservando un intervalo de tiempo. Este intervalo de tiempo se almacena en un estado que podríamos calificar como “pendiente”, puesto que todavía no se ha pagado, y bloquea la selección de esa misma ranura de tiempo para el resto de clientes. Obviamente, en este sistema, hay condiciones de carrera puesto que los clientes entran en la web de forma paralela.

Ahora, supongamos además que hay varios servicios (lavado, encerado, control de presión, limpieza interior del vehículo...) que son seleccionables y que cada uno de ellos se realiza en una ubicación distinta en distintos “box”. Así, internamente el sistema deberá llevar el control de la ocupación de cada “box”.

Cuando se realice un pago por internet, con todo el protocolo de comunicación que ha de ser implementado, deberá marcar la venta como pagada y reservar efectivamente la ocupación para el cliente, es obvio que éste se ha introducido en una máquina de estados implementada en el sitio web. Cuando el pago sea efectivo, deberá generar un ticket para que el encargado pueda leer de forma automática qué servicios ha comprado el cliente. Por supuesto, en este ticket deberá indicar quién ha realizado la compra.

Este tipo de problemas de restricciones de competencia entre instancias de un sistema pueden bien con bloqueos o pueden modelarse con árboles binarios de decisión (Binary Decision Diagram, BDD, en inglés) de forma que el sistema sea capaz de detectar inconsistencias y por ende, verificar que el el proceso se encuentra con unos datos incorrectos y por tanto, cancelar el procedimiento actual y volver a un estado seguro. En el caso de una detección de un estado inconsistente, el software podrá detener el proceso y volver al estado correcto más cercano. En nuestro ejemplo, cancelar la compra.

Otro ejemplo más lejano a un usuario normal puede ser el de desarrollo de software para sistemas de control. Se modelan los estados como proposiciones lógicas y se estructura un BDD con todas las restricciones. Este árbol se mantiene en memoria y refleja el estado real del sistema. De forma que cuando el sistema entre en un estado inconsistente, puede reiniciarlo o volver a un estado seguro. Un ejemplo de este tipo de sistemas puede ser cualquier software que compruebe que una máquina industrial está funcionando correctamente.

En definitiva, sea cual sea el uso que se le dé, usar un BDD como software de control adicional permite que se detecten errores en el software y que el sistema vuelva al estado estable más cercano. Además, el BDD reflejará en cada momento el estado real del sistema.

Una vez puesto en situación, podemos entonces entender que en este trabajo se ha implementado esa *piedra angular*. Una librería eficiente para BDDs que implementa todas las herramientas para que un equipo de ingenieros de software la puedan usar para construir sistemas tolerantes a fallos.

Además, esta implementación sigue las directrices más estrictas en cuanto a desarrollo, de manera que se pueda continuar su desarrollo de forma sencilla por cualquier estudiante de máster o doctorado, al contrario de como ocurre con algunas implementaciones oscuras basadas en lenguajes menos modernos que además tienen un nivel ofuscación y falta de diseño estructurado que hacen difícil el trabajo sobre ellas.

A lo largo de este trabajo se mostrará cómo se ha implementado esta solución, los problemas que se encontraron, las soluciones que se han aportado y los algoritmos que se han usado.

En especial, cabe destacar la parte de optimización de BDDs, en la que se han realizado varios experimentos con varios algoritmos que tratan, mediante el intercambio de variables de reducir el número de nodos de un árbol de este tipo.

Inicialmente daremos una pequeña introducción sobre qué son este tipo de estructuras, luego hablaremos sobre su implementación y por último mostraremos esos resultados de optimización de tamaño de BDDs sobre varios *benchmarks* basados en el formato DIMACS [DIMACS].

Objetivo

El objetivo de este trabajo fin de máster era realizar un estudio completo de algún método de verificación de software. Me decidí por los árboles binarios de decisión, en primer lugar porque ya había trabajado con ellos con anterioridad y el tema no me era del todo desconocido y en segundo lugar, porque era uno de los temas más explotado en cuanto a verificación de software y hardware, lo que me permitiría encontrar abundante información y ejemplos de uso en el mundo real.

Una vez fijado el tema, quería desarrollar una serie de experimentos que mostrasen cómo optimizar árboles binarios de decisión de forma sencilla. Esto no es fácil y en sucesivos trabajos he visto que los algoritmos exactos tienen órdenes de eficiencia muy grandes. Hasta que llegué al trabajo de [Rudell93], en el que, mediante una operación constante, monta dos algoritmos de optimización de BDDs.

Una vez que tuve claro en qué dirección iba a ir el trabajo, pensé en la implementación. Inicialmente no sabía si tomar una implementación referencia o realizar yo una propia. Es cierto que hay mucho software sobre este tema y todas está muy estudiada. Por ejemplo, las librerías BuDDy [BuDDy] y CUDD [CUDD] ofrecen dos de las librerías más usadas y eficientes. Ahora bien, la calidad del software de estos proyectos es escasa. La legibilidad del código es pequeña y la la documentación se refiere a su uso, pero no a su diseño. Estos paquetes tienen como objetivo el ser los más rápidos en cuanto a tiempo de ejecución, pero no logran tener ser un software de calidad ni muchos menos.

Estando en un máster de Ingeniería del Software, decidí hacer una implementación que respetase algo más los estándares y buenas prácticas y que, además, me permitiese realizar experimentos de forma sencilla y sabiendo exactamente qué estaba ocurriendo.

He llegado a leer determinados trabajos en los que se hacía uso de paquetes con BDDs en los que los autores no sabían como funcionaban éstos, y eso nos parece un grave error. Si no sabes exactamente como está hecho, ¿cómo puedes extraer conclusiones científicas?

De esta forma, decidí realizar una implementación desde cero de una librería de BDDs. Puesto que la eficiencia y el competir con otras implementaciones más asentadas no era mi objetivo, me decidí por usar un lenguaje de programación que ofreciese lo mejor de los dos mundos. Por un lado quería que estuviese cercano a la máquina en cuanto a eficiencia y por otro quería que me resolviera algunos temas como la gestión de memoria dinámica. Por todo eso, escogí el lenguaje estrella de la plataforma JVM, Java.

Comencé comprobando como efectivamente era muy costoso la creación de árboles simplemente mediante un generador recursivo y se necesitaba una funcionalidad más elaborada para alcanzar un orden de eficiencia que no fuera exponencial. Mi objetivo era tener un paquete no que pudiera competir en tiempo con BuDDy o el CUDD, sino que me permitiera trabajar con distintos algoritmos y con el que pudiera realizar experimentos.

Además, los paquetes de software existentes usaban aristas complementadas, de lo que yo no era muy partidario, ya que se pierde la unicidad del BDD ante una fórmula lógica determinada.

También comprobé como es complicada la paralelización de este tipo de algoritmos[1].

En definitiva, en este trabajo se incluye una librería de uso de BDD basada en Java que puede integrarse en sistemas de verificación formal o en sistemas de control. También se muestran una

1 Thomas van Dijk lo ha implementado en su tesis [Dijk2012], pero para ello ha tenido que modificar radicalmente el enfoque que se hacía hasta ese momento en la construcción de BDDs.

serie de resultados en los que se puede comprobar cómo se comportan los distintos algoritmos de reducción del número de vértices en árboles binarios de decisión.

Lógica proposicional

La lógica proposicional es un sistema formal en el que las fórmulas de un lenguaje formal se interpretan como proposiciones. Se usa un conjunto de reglas de inferencia y axiomas para derivar fórmulas. A estas fórmulas derivadas se las llama teoremas.

En la lógica proposicional, podemos interpretar las fórmulas de forma sintáctica, realizando derivaciones unas de otras y obteniendo resultados, o por el contrario podemos dar valores a cada proposición y aplicando los operadores lógicos obtener una interpretación de la fórmula. Ambas formas de proceder son equivalentes, así que para propósitos de este trabajos supondremos ambos tipos de demostración equivalentes.

Definición

La lógica proposicional (o lógica clásica) es aquella determinada por enunciados que pueden ser o bien ciertos, o bien falsos.

De forma formal, podemos decir que una proposición *p* es:

- *p* = TRUE
- *p* = FALSE
- ¬ *q*, donde *q* es una proposición y ¬ el operador de negación
- Dadas *p* y *q* proposiciones y OP un operador lógico, *p* OP *q* es otra proposición lógica.

A continuación explicamos cada uno de estos conceptos

Constantes

TRUE

Proposición cierta. Se representa como TRUE, true, 1 o T.

FALSE

Proposición falsa. Se representa con FALSE, false, 0 o F.

Operadores

Los operadores en la lógica proposicional se definen mediante tablas de verdad. Recordemos lo dicho anteriormente, que la demostración sintáctica y semántica son equivalente. Así, dejamos para el lector la demostración sintáctica, dado que está demostrado [LICS] que son equivalentes.

Operador de negación

p	¬ *p*
0	1
1	0

Operador de conjunción

Este operador proporciona una proposición cierta siempre y cuando al menos una de las proposiciones, que actúan como sus operandos, lo sea.

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

Muchas veces este operador se identifica con los símbolos ·, AND. También suele escribirse yuxtaponiendo directamente las variables, de forma que se escribirían una detrás de otra, por lo que si vemos $F = xy$, nos referiremos a $F = x$ AND y.

Operador de disyunción inclusiva

Este operador proporciona una proposición cierta siempre y cuando al menos una de las proposiciones, que actúan como sus operandos, lo sea.

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

Este operador se identifica con los símbolos OR, V, +.

Resto de operadores

El resto de operadores posibles puede definirse en función de los ya indicados. Por ejemplo, el operador de implicación obedece la siguiente tabla de verdad:

p	q	$p \Rightarrow q$	$\neg p \vee q$
0	0	1	1
0	1	1	1
1	0	0	0
1	1	1	1

Para el operador de equivalencia tendríamos algo parecido

p	q	$p \Leftrightarrow q$	$(\neg p \vee q) \wedge (p \vee \neg q)$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	1	1

No es nuestro objetivo mostrar todos los posibles operadores binarios lógicos, por lo que, ante un operador no incluido aquí, supondremos que la fórmula puede convertirse sin problemas a cualquier combinación de los operadores incluidos anteriormente.

En este trabajo no vamos a detallar todos los operadores posibles como XOR, NAND, NOR, etc. El lector puede encontrar más información en [LICS] y conociendo su comportamiento ante las entradas, si la salida es determinista podrá expresarlo como combinación de AND, OR y NOT.

Precedencia

De aquí en adelante, vamos a suponer que la precedencia de operadores sigue el siguiente orden:

- NOT
- AND
- OR
- $\rightarrow$

De todas formas, salvo en el caso de NOT, que tiene siempre la mayor precedencia, vamos a incluir paréntesis para evitar confusiones entre las distintas precedencias.

Propiedades

Propiedad asociativa

Los operadores OR, AND y $\leftrightarrow$ son asociativos, esto quiere decir que se cumplen estas equivalencias:

Asociatividad del operador OR

$$p \vee (q \vee r) \Leftrightarrow (p \vee q) \vee r$$

Asociatividad del operador AND

$$p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$$

Asociatividad del operador de equivalencia

$$(p \Leftrightarrow (q \Leftrightarrow r)) \Leftrightarrow ((p \Leftrightarrow q) \Leftrightarrow r)$$

Propiedad distributiva

Distribución de la conjunción con respecto a la disyunción

$$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$$

Distribución de la disyunción con respecto a la conjunción

$$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$$

Leyes de De Morgan

Las leyes de De Morgan relacionan los operadores AND y OR con el de negación según la siguiente expresión:

$$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$$

$$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$$

Estas propiedades son interesantes debido a que deberemos tener un algoritmo para negar algún subárbol, se deberá recorrer todos los vértices e ir intercambiando los descendientes de cada vértice. Esto de hecho es lo que se hace cuando se construye una expresión en la que la negación afecta a una subexpresión compleja, como por ejemplo: $\neg((a \wedge b) \vee c)$.

Árboles Binarios de Decisión

Introducción

Un árbol binario de decisión [BRYANT1986] es una forma de representar tablas de verdad basada en la expansión de Boole[2], en la que se enuncia que cualquier fórmula lógica puede descomponerse en la expresión:

$$F = xF_{(x=1)} + \neg xF_{(x=0)}$$

Donde $F_{(x=0)}$ es la expresión F asignando a la variable x el valor *false* y $F_{(x=1)}$ es la expresión F sustituyendo la variable x por el valor *true*.

Si representamos esta expresión de forma gráfica tendremos un nodo con la expresión F (que depende de la variable *x*), de forma que si se le da a la x el valor 0 se obtiene F(x=0) y se se le da el valor 1 se obtiene F(x=1). La asignación de x=0 la marcamos con una arista discontinua, y la asignación de x=1 la marcamos con una arista continua.

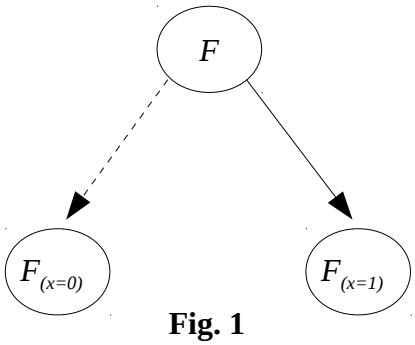


Fig. 1

Si F depende de otra variable, llamada y, podremos repetir este mismo proceso de bifurcación de asignaciones con $F_{(x=0)}$ y con $F_{(x=1)}$. Es decir, si suponemos que la siguiente variables es y, tendríamos el siguiente árbol de decisión:

2 También se le suele llamar expansión de Shannon, aunque su creador fue George Boole, matemático, lógico y filósofo inglés del siglo XIX. Nosotros usaremos distintamente cualquiera de estos dos nombres.

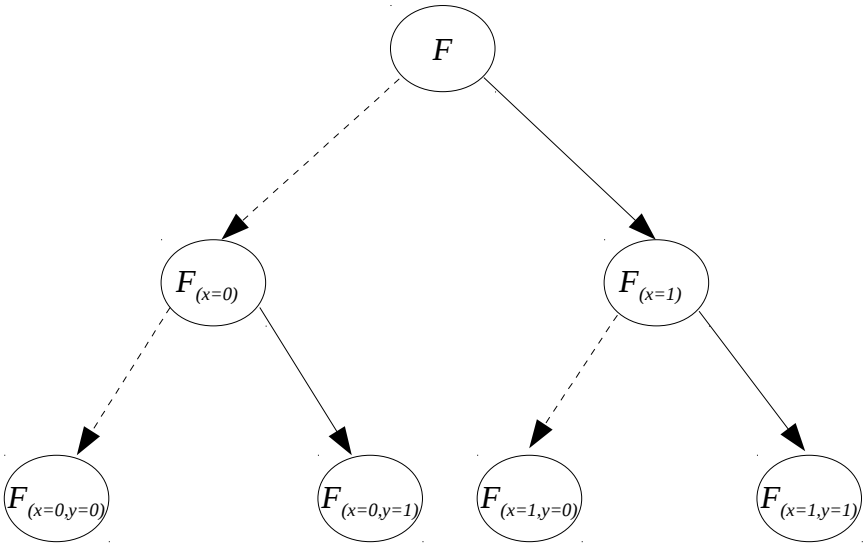


Fig. 2

Como podemos ver, hablamos de árboles binarios de decisión porque cada nodo tiene exactamente dos nodos descendientes.

Puede ocurrir que alguno de los nodos no dependa de ninguna variable y tenga un valor constante (**0** ó **1**). En ese caso, ese nodo será un nodo hoja, dado que no tendrá descendientes, y sólo podrá tomar el valor *true* o *false*.

Vamos a ver un ejemplo de fórmula lógica real $F = xy$. Y tomando el orden de las variables como x, y luego, y. También, para facilitar la visibilidad del diagrama, suponemos que $F_{(a,b)} = F_{(x=a,y=b)}$.

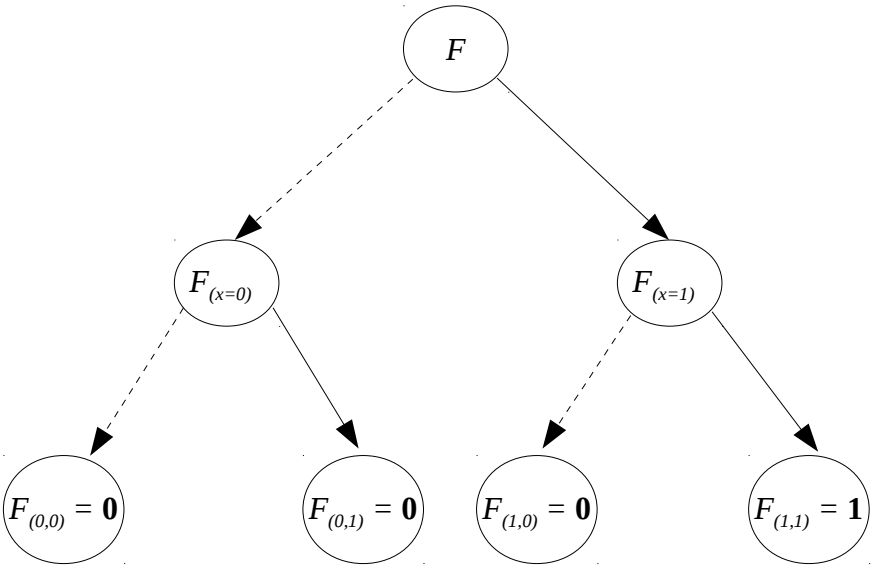


Fig. 3

Notemos que al nodo que viene de una arista punteada se le suele llamar, *low*, y al de la línea llena *high*, debido a que son producto de la asignación de **0** y **1** a su vértice padre, respectivamente.

Otro ejemplo en el que podemos ver varios niveles es el siguiente: $F = xz + y$, con el orden x, y, z.

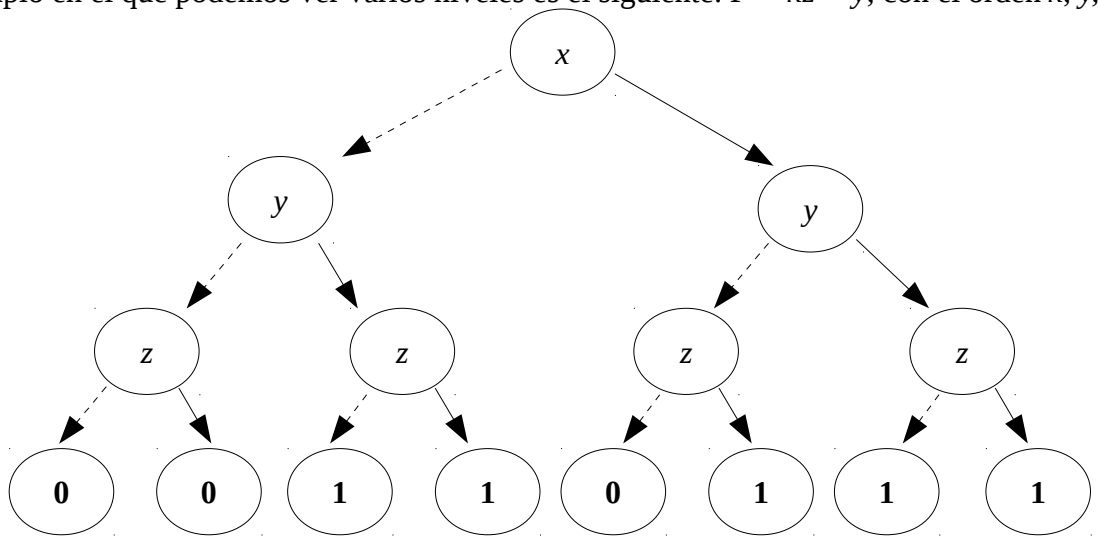


Fig. 4

Como se puede ver en este ejemplo, hay ramas completas repetidas. En la siguiente sección vamos a mostrar cómo se pueden reducir los BDD haciendo que pasen de ser árboles binarios a dígrafos acíclicos.

Árboles reducidos

Dado el árbol de la **Fig. 3** (función $F = xy$), un detalle importante que podemos apreciar en este diagrama es la repetición de nodos. Es decir, podemos ver cómo se repiten los nodos hoja y tenemos tres nodos con el valor **0** y sólo uno con el valor **1**. Si eliminamos los nodos con el mismo valor, obtenemos el siguiente árbol binario reducido:

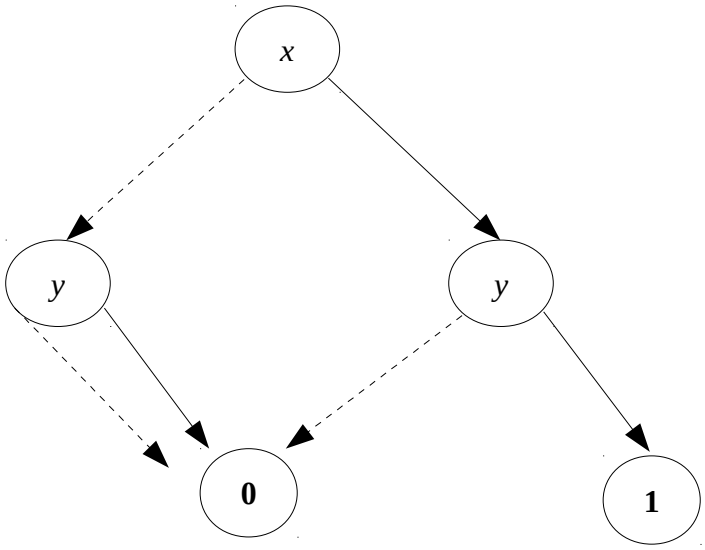


Fig. 5

Como podemos ver, hemos reducido el árbol en 2 nodos sobre un total de 7 nodos. Esto no puede parecer mucho, pero pensemos a gran escala, en fórmulas de miles de variables, conseguir una reducción de un tercio de su tamaño es muy deseable.

También vemos como, en el nodo y de la izquierda, independientemente del valor que tome y, se obtiene el valor **0.** ¿No se podría tener una normas para reducir aún más el árbol evitando repeticiones innecesarias?

Randal Bryant en la Universidad Carnegie Mellon inventó el concepto de Diagrama de Decisión Binario Ordenado Reducido Compartido en [BRYANT1986], indicando las restricciones que debía tener un BDD para que fuera mínimo y único dado el mismo orden de variables. Para ello, el diagrama ha de cumplir dos propiedades.

No-redundancia

En primer lugar, no debe haber vértices redundantes, esto es, no puede haber un vértice cuyos descendientes sean otro. En el caso de que en el proceso de construcción esta casuística se dé, se tomará el nodo descendiente y se eliminará el nodo y se le pasará el

Es decir, dado un vértice *v*, si *low(v) = high(v)*, entonces se produce la siguiente sustitución:

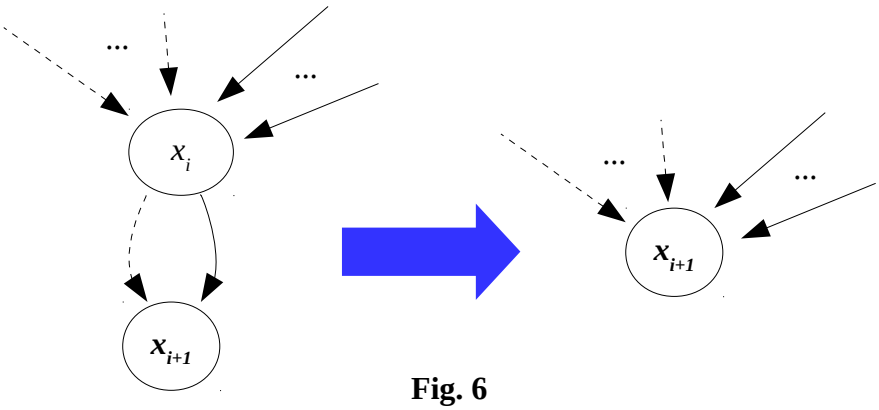


Fig. 6

Unicidad de cada vértice

Si un vértice tiene los mismos descendientes que otro y la misma variable, entonces ha de ser sustituido por éste. Es decir, supongamos que tenemos un árbol que tiene dos vértices distintos *u* y *v* que cumplen que ambos tienen la misma variable y que tienen los mismos descendientes *low* y *high*. Entonces, ambos son el mismo vértice.

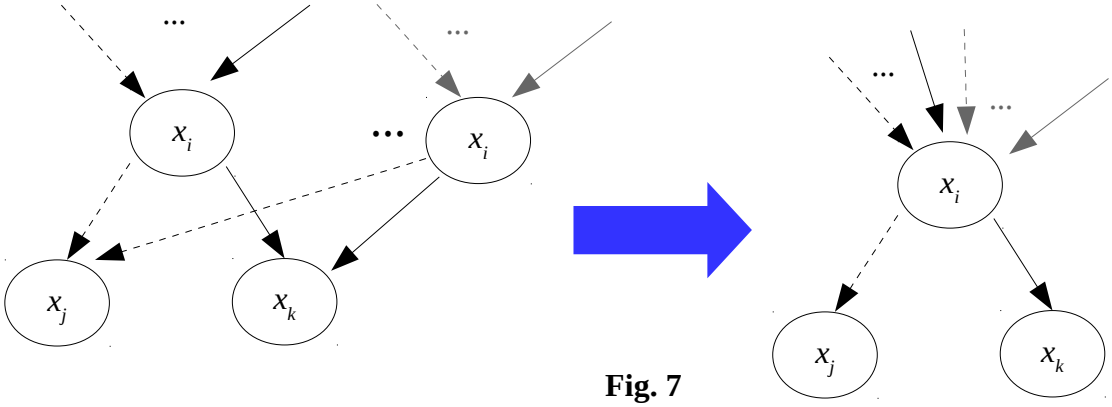


Fig. 7

Por ejemplo, si tenemos el árbol de la figura 4 ($F = xz + y$), tendremos varios vértices repetidos y redundantes. Si los eliminamos, obtendremos el siguiente árbol reducido:

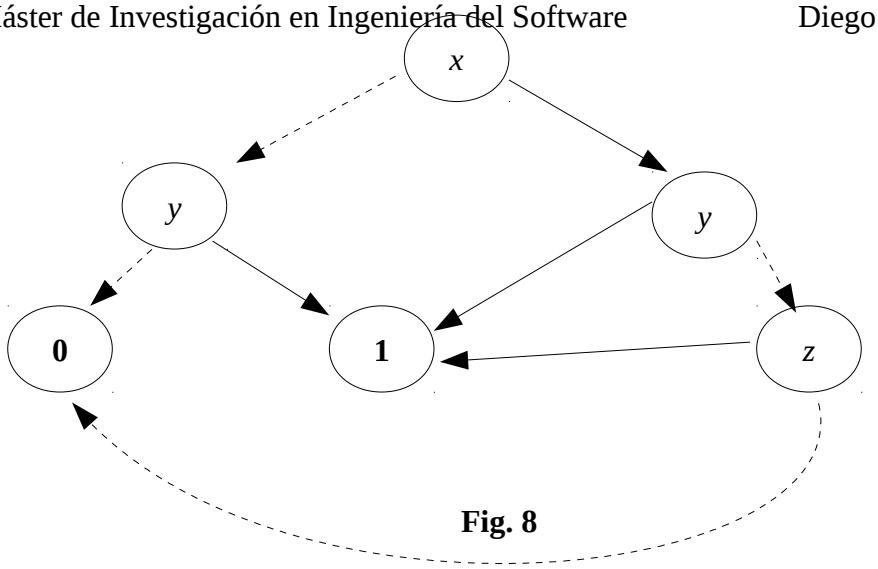


Fig. 8

Orden de las variables

Como podemos ver, hablamos de que las variables tienen un orden entre ellas. Esto nos sirve para evitar tener niveles repetidos, tengamos en cuenta que cada camino que va desde la rama raíz representa una asignación de valores de verdad a variables, lo que sería equivalente a una fila de una tabla de verdad, y por tanto, no se puede repetir asignaciones.

De igual forma, hemos de respetar este orden en todas las ramas del árbol, o si no, podríamos encontrarnos con problemas a la hora de evaluar funciones o de reducir el árbol.

Por ejemplo, supongamos la función $F=xz+y$, mostramos a continuación dos órdenes de los posibles, *(x, y, z)* y en la imagen de la derecha *(x, z, y)*:

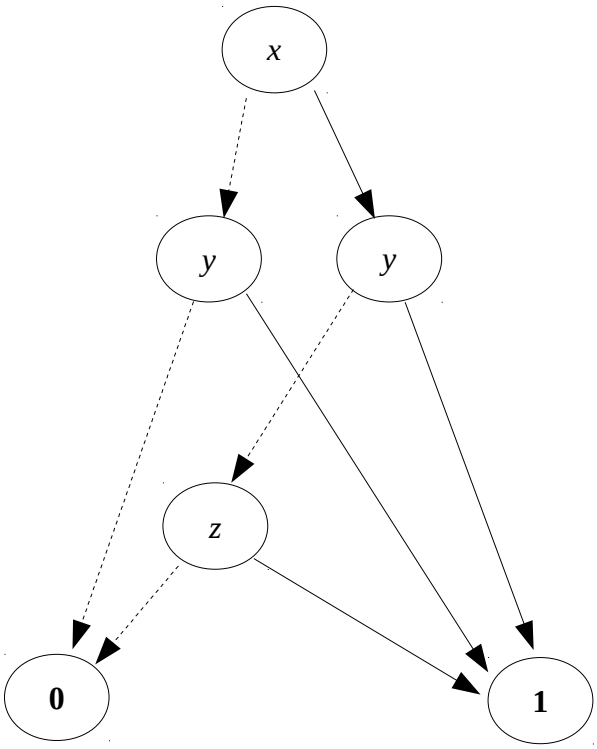


Fig. 9

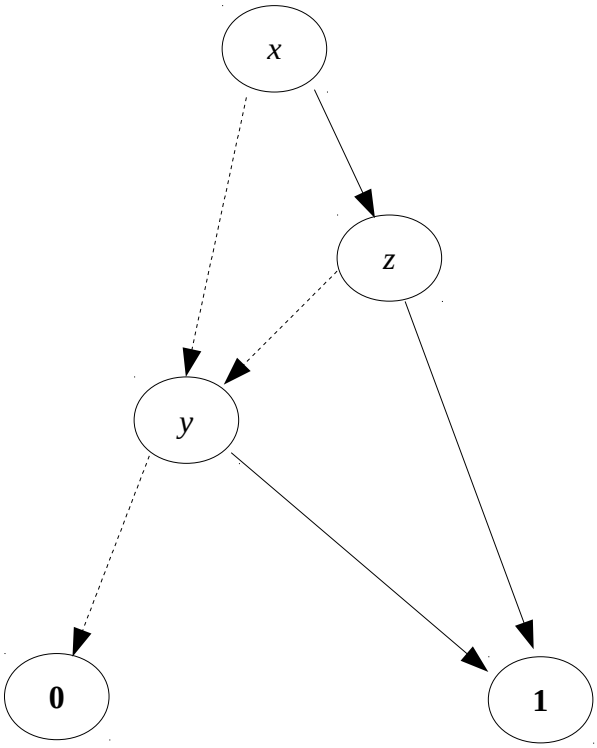


Fig. 10

El BDD de la figura 9, que es el que tiene el orden (x, y, z) tiene un nodo más que el árbol de la figura 10, que es el que tiene el orden (x, z, y). Como podemos ver, el orden es determinante en cuanto al tamaño del árbol.

Por supuesto, en este sencillo ejemplo sólo se incrementa de tamaño el BDD un vértice, lo que no es mucho. Ahora bien, pensemos en el árbol para la fórmula lógica $F=x_1x_3+x_2x_4$. Si suponemos que el subíndice de cada variable indica su orden en el árbol, obtendríamos un BDD de 8 vértices. En cambio, si le dotamos de un orden algo más inteligente, siendo éste (x_1, x_3, x_2, x_4), el BDD es de un tamaño de 6 vértices.

A continuación se muestran los árboles gráficamente:

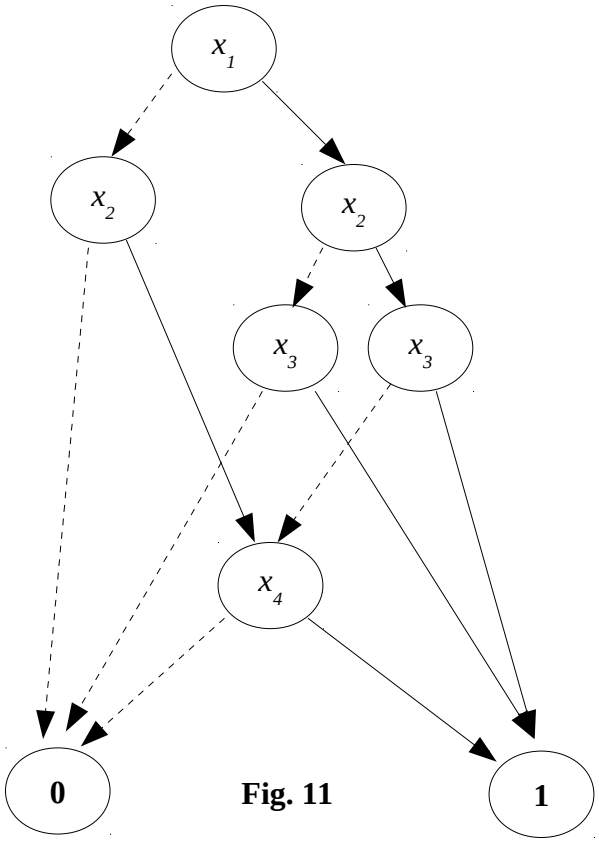


Fig. 11

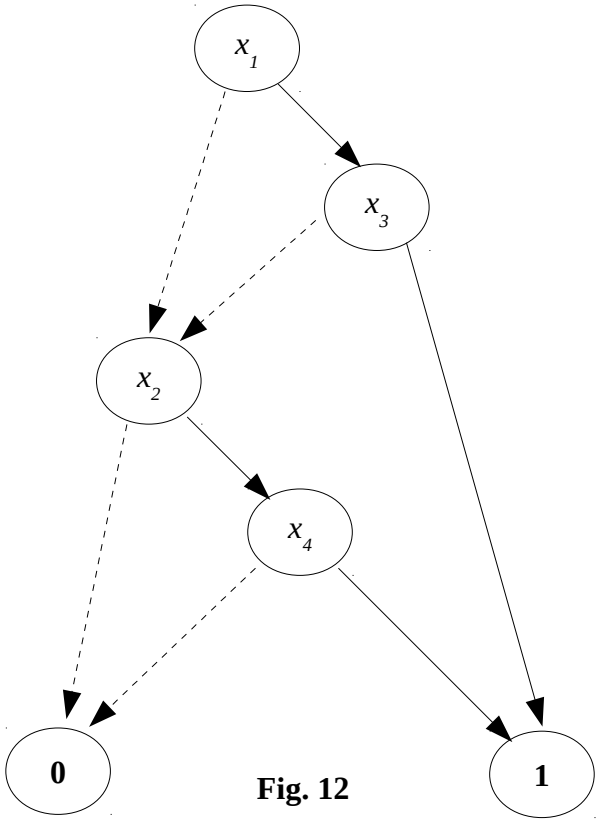


Fig. 12

Notemos cómo se puede pasar de un número de nodos exponencial a uno lineal simplemente cambiando el orden a las variables.

De hecho, la mayoría de las estrategias de optimización tratan de encontrar órdenes que reduzcan el tamaño del árbol.

Definición

Ahora que tenemos ciertos conceptos, vamos a dar una definición de lo que es un ROBDD.

Un árbol binario de decisión es un grafo dirigido acíclico con un vértice como raíz.

Cada vértice tiene los siguientes atributos *v = {id:Int, var:Variable, low:Vertex, high:Vertex}*.

- *id*: identificador del vértice. Número entero mayor que cero. Nos servirá para identificar cada vértice y distinguirlo del resto.
- *var*: variable que está asociada con el vértice. Como hemos visto antes, cada nivel de vértices tiene asociada una variable. Las variables se numeran como enteros mayores que cero indicando de esta forma el orden que tienen en el árbol, esto es var_i precede a var_j si y solamente si $var_i < var_j$
- *low*: referencia al vértice que se toma cuando se la ruta toma el valor 0 para el vértice actual.
- *high*: referencia al vértice que se toma cuando se la ruta toma el valor 1 para el vértice actual.

No se permiten ciclos en el árbol, esto es,

$$Para\ todo\ vértice\ v,\ v.low = u \leftrightarrow v.var < u.var$$ [3]

A *low* y *high* se le llama los descendientes de cada vértice y cumplen las siguientes propiedades:

Sean *v* y *u* vértices del árbol.

- *v.low != v.high*
- *v.low = u.low* AND *v.high = u.high* AND *v.var = u.var* $\leftrightarrow$ *v = u*

Existen unos vértices especiales constantes:

- **1**: {*id=1*, *low=null*, *high=null*}
- **0**: {id=0, *low=null*, *high=null*}

Estos vértices constantes no tienen descendientes, de ahí que sus atributos *low* y *high* tomen el valor *null*.

Operaciones

Los árboles de decisión tienen una serie de operaciones propias [Bryant1992], [Somenzi]. Nosotros hemos implementado las que nos parecían más importantes. Obviamente la creación, *Apply* y la restricción. Si se desea otra operación, dada la naturaleza abierta de la librería animo al lector a estudiar la implementación de otras operaciones.

3 En este documento usaremos la notación orientada a objetos para indicar que la propiedad pertenece al objeto, de manera que *v.var* indica la variable del vértice *v*.

Creación

Normalmente los autores tal y como ocurre en [Bryant1992][Andersen1999] hacen uso de dos estructuras de datos:

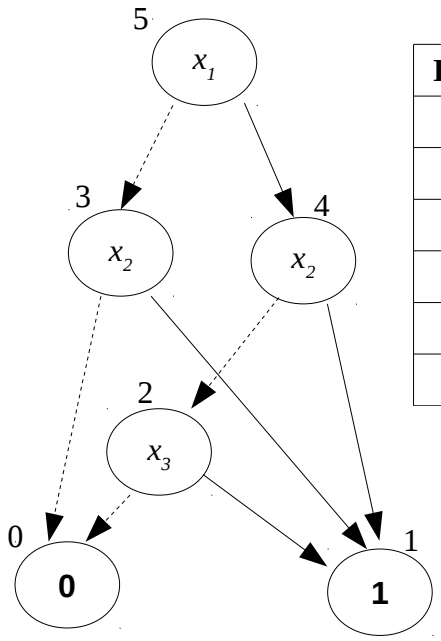
La tabla de vértices en la que un índice nos da la tripleta: *variable*, *low* y *high*. Normalmente a esta tabla se la suele llamar *T*.

Por ejemplo, para el árbol $F = x_1x_3 + x_2$

Tendríamos las variables en orden:

Variable	x_1	x_2	x_3
Índice	1	2	3

Y el siguiente árbol su tabla T:



Id vértice	**Variable**	**Id vértice *low***	**Id vértice *high***
0			
1			
2	3	0	1
3	2	0	1
4	2	2	1
5	1	3	4

Fig. 13

En la tabla *T*, podemos ver como a los nodos hoja *true* y *false* le asignamos de forma arbitraria los identificadores 0 y 1 y obviamente sin variable ni descendientes. El resto de nodos se va asignando de abajo a arriba (veremos más adelante cómo se genera un árbol con la operación *Apply*), aunque en realidad, el identificador da igual, lo que nos tiene que interesar es que esa tabla contiene la estructura arbórea y la asociación entre vértices y variables.

De esta forma, gracias a esta tabla *T* podemos extraer de forma eficiente qué variable tiene el nodo y sus descendientes. Ahora bien, tal y como hemos visto antes, hablamos mucho de unicidad de nodo en cuanto a que sólo existe un nodo con la misma variable, *low* y *high*. Por tanto usaremos una tabla inversa para poder obtener y comprobar que efectivamente sólo hay un único vértice con una variable y unos valores de *low* y *high* determinados. A esta tabla inversa la podemos llamar *H*.

La tabla *H* es una tabla que dada la variable, y valores *low* y *high* obtiene el vértice o nos informa de que no existe en tiempo constante.

A partir de este momento, la construcción sigue un proceso recursivo basado en la expansión de Shannon [Andersen1999]. Mostramos su pseudocódigo a continuación:

```
# Obtiene un nuevo vértice
mk(variable, low, high):
        # Si ambos descendientes son iguales, no creamos un nuevo vértices
        # devolvemos uno de ellos
        if low == high:
                return low
        # Si existe el vértice con la misma variable y descendientes
        # Lo devolvemos
        if H[variable, low, high] != null:
                return H[variable, low, high]
        # Es un nuevo vértice
        v = Vertex(variable, low, high)
        H[variable, low, high] = v
        # Añadimos el vértice v a la tabla T con un id único
        T[T.getNewKey()] = v
        return v

# Construye cada uno de los niveles del árbol
# Nótese que cada nivel es en realidad una variable
build(formula, level):
        if level > n:
                if t == False
                        return 0
                return 1
        else:
                # Obtiene el vértice low del nuevo vértice
                low = build(formula(x[level]=0), level+1)
                # Obtiene el vértice high del nuevo vértice
                high = build(formula(x[level]=1), level+1)
                # Nuevo vértice
                return mk(level, low, high)

# Llamada inicial
```

```
buildBDD(formula):
                return build(formula, 1)
```

Nótese que en la construcción en el paquete software este es uno de los métodos de construcción de árboles, siendo el otro el uso del algoritmo *Apply*, que se detalla a continuación.

Apply

El algoritmo *Apply* que permite ejecutar una operación binaria lógica entre dos árboles BDD fue detallada en [Bryant1986].

En este algoritmo se va construyendo de forma recursiva gracias a la siguiente propiedad de la expansión de Shannon:

Si definimos un operador que podemos llamar *if-then-else* con una forma similar al operador ternario de cualquier lenguaje de programación:

$$x \rightarrow y_{0,} y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$$

Entonces la expansión de Shanon puede definirse de la siguiente forma:

$$t = x \rightarrow t_{(x=1)}, t_{(x=0)}$$

Pues bien, la siguiente propiedad se cumple para todos los operadores booleanos binarios:

$$(x \rightarrow t_{1,} t_2) op (x \rightarrow t'_{1,} t'_2) = x \rightarrow (t_1 op t'_1), (t_2 op t'2)$$

Una vez dicho esto, podemos mostrar el pseudocódigo del algoritmo definido por Bryant:

```
# Construye una nueva clave para cada vértice
makeUniqueKey(var_index, low, high):
        if(low==null && high==null):
            return var_index+"-"+"NULL"+"-"+"NULL";
        return var_index+"-"+low.index+"-"+high.index

# Añade un vértice único a la tabla hash
addNewVertex(var_index, low, high):
        # Obtiene la siguiente clave e incrementa el contador global
        index = getNextKey()
        v = new Vertex(index, var_index, low, high)
        T[index] = v
        U[makeUniqueKey(var_index, low, high)] = v
        V[var_index].add(v)
        return v

# Añade un vértice no-redundante a la tabla hash
addVertex(var, low, high):
```

```
        vertexUniqueKey = makeUniqueKey(var_index, low, high)
        if(uniqueKey  in U):
            return U[uniqueKey]
        return addNewVertex(var, low, high)

# Aplica el algoritmo Apply a dos vértices
applyVertices(Vertex v1, Vertex v2):
        # Clave hash del subárbol de estos dos vértices
        String key = "1-" + v1.index + "+2-" + v2.index
        if( key in G ):
            return G[key]

        if(v1.isLeaf() && v2.isLeaf()):
                # op es la operación boolean entre dos vértices hoja
                # (que pueden ser true o false)
            if(op(v1,v2)):
                return True # Vértice con el valor true
            return False  # Vértice con el valor false

        var = -1
        low = null
        high = null
        # v1.index < v2.index
        if (!v1.isLeaf() and (v2.isLeaf() or v1.variable < v2.variable)):
            var = v1.variable;
            low = applyVertex(v1.low, v2)
            high = applyVertex(v1.high, v2)
        else if (v1.isLeaf() or v1.variable > v2.variable):
            var = v2.variable
            low = applyVertex(v1, v2.low)
            high = applyVertex(v1, v2.high)
        else:
            var = v1.variable
            low = applyVertex(v1.low, v2.low)
            high = applyVertex(v1.high, v2.high)

        # Respeta la propiedad de no-redundancia:
        # "Ningún vértice ha de tener como low y high a un único vértice."
```

```
        if(low.index == high.index):
            return low

        # Respeta la propiedad de unicidad:
        # "Ningún vértice ha de tener la misma variable y vértices
        # low y high que otro."
        Vertex u = addVertex(var, low, high)
        G[key] = u
        return u;

# Llamada al algoritmo de apply
apply(operation, bdd1, bdd2):
      # Caché para evitar cálculos repetidos
      G = {}
      String function = bdd1.function + " "+operation+" "+bdd2.function
      # Llenar la tabla hash T que contiene el nuevo árbol
      # con vértices de bdd1 y bdd2
      root = appyVertex(bdd1.root, bdd2.root)
      # Construcción del nuevo BDD
      return new BDD(function, root)
```

Restrict

En [Bryant1992], Randal Bryant define esta operación de manipulación de árboles binarios de decisión que asigna valores booleanos a algunas variables, de esta forma restringiendo la ruta desde la raíz del árbol y obteniendo un nuevo árbol de menor tamaño.

De forma gráfica podríamos verlo más fácilmente:

Supongamos el árbol $F = x+yz$

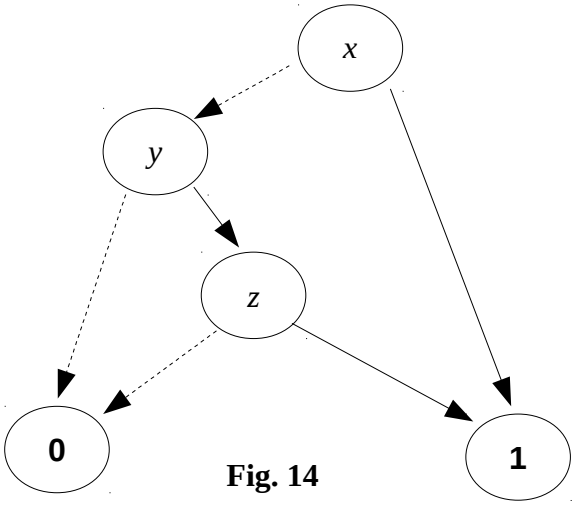


Fig. 14

Si asignamos a la variable *y* el valor **1**, el BDD perderá todos los vértices con esa variable y se sustituirán por los de su rama *high*. En nuestro caso, sólo hay un vértice, de forma que el *low* del vértice *x* llegaría directamente al vértice z, es decir:

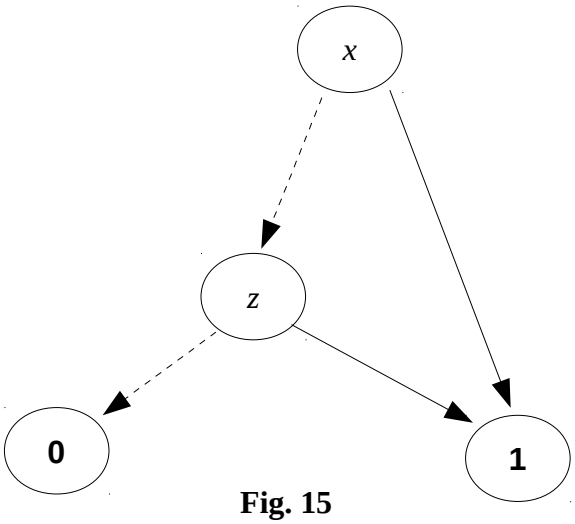


Fig. 15

Como se puede ver a simple vista, este árbol restringido, obedece a la expresión lógica $F_{y=1}=x+z$.

Además de obtener un árbol restringido, esta operación puede usarse para realizar una evaluación lógica de la expresión contenido en el BDD. De esta forma, si se asigna un valor lógico a cada variable, al final se obtendrá un árbol de sólo un vértice, el vértice **1** o **0**. Obviamente el vértice resultante será el que indique el valor de verdad de la expresión lógica contenida en el árbol, dada esa asignación de valores de verdad para las variables.

```
# Obtiene una nueva raíz de un BDD basándose en el BDD con raíz v
# y con una asignación de valores booleanos en algunas variables
restrictFromVertex(v, assignement):
        if(v.isLeaf()):
            # Hay un único vértice 1 ó 0
            return T[v.index]
        if(v.variable in assignement):
```

```
            boolean value = assignement[v.variable]
            if(value):
                return restrictFromVertex(v.high, assignement)
            else:
                return restrictFromVertex(v.low, assignement)
        else:
            low = restrictFromVertex(v.low, assignement)
            high = restrictFromVertex(v.high, assignement)
            if(low.index == high.index)
                return low
            return addVertex(v.variable, low, high)

# Obtiene un nuevo BDD basándose en este BDD junto con una
# asignación booleana a algunas variables.
restrict(bdd, assignement):
        restrictedBDD = restrictFromVertex(bdd.root, assignement);
        rfunction = bdd.function
        for(pair : assignement.pairs()):
            variable = VARIABLES[pair.key]
            value = pair.value
            rfunction = rfunction.replace(variable, value)
        return new BDD(rfunction, restrictedBDD)
```

Swap

Esta operación intercambia dos niveles del árbol. El objetivo es tener una operación que permita obtener algunos vértices huérfanos, de forma que se puedan eliminar y por tanto reduciendo el tamaño del árbol.

La implementación es ignorada en la mayoría de los trabajos de la bibliografía y en ellos se indica [Rud93] como referencia. Para comprender mejor esta operación se pueden ver las notas del mismos autor en el taller [Rud93W].

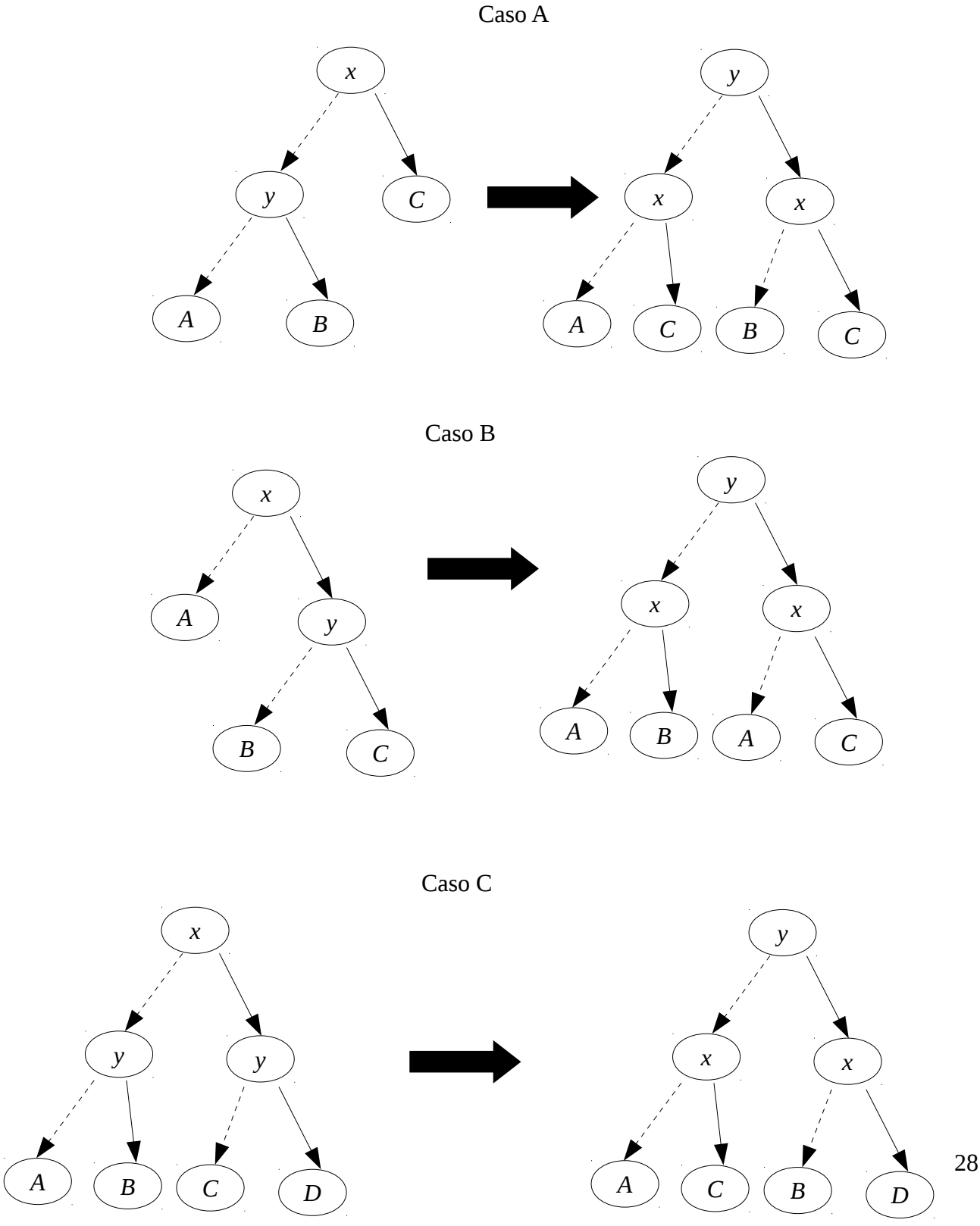


Fig. 16

En el resto de escenarios de intercambios de variables, el vértice *x* no tiene vértices descendientes con la variable *y*, por lo tanto, no se produce ningún intercambio.

A continuación incluimos el pseudocódigo de los distintos casos de intercambio, indicando en cada uno de ellos a qué caso se refiere.

El método `addWithoutRedundant(varI, A, C)` añade un vértice con esos valores de variable y con los descendientes low y high, comprobando que no existe ya un vértice con esos atributos. Si no existe, devuelve un nuevo vértice con esos valores, si existe, lo obtiene y de igual forma lo devuelve.

```
# Reemplaza el vértice v (si tiene la variable varJ)
# con el vértice del siguiente nivel
swapVertex(Vertex v, int varJ):
        swapWasMade = false
        varI = v.variable

        low = v.low
        high = v.high

        A = null
        B = null
        if (!low.isLeaf()):
            A = low.low
            B = low.high
        else:
            A = low
            B = low

        C = null
        D = null
        if (!high.isLeaf()) :
            C = high.low()
            D = high.high()
        else:
            C = high
            D = high

        newLow = null
```

```
        newHigh = null

        # Caso a:
        if (low != null && low.variable == varJ &&
                  (high == null || high.variable != varJ)):
            newLow = addWithoutRedundant(varI, A, C)
            newHigh = addWithoutRedundant(varI, B, C)
            setVertex(v, varJ, newLow, newHigh)
            swapWasMade = true
        # Caso b:
        else if ((low == null || low.variable != varJ) &&
                  (high != null && high.variable == varJ)):
            newLow = addWithoutRedundant(varI, A, B)
            newHigh = addWithoutRedundant(varI, A, C)
            setVertex(v, varJ, newLow, newHigh)
            swapWasMade = true
        # Caso c:
        else if ((low != null && low.variable == varJ) &&
                  (high != null && high.variable == varJ)):
            newLow = addWithoutRedundant(varI, A, C)
            newHigh = addWithoutRedundant(varI, B, D)
            this.setVertex(v, varJ, newLow, newHigh)
            swapWasMade = true
        # Caso d:
        else if ((low == null || low.variable != varJ) &&
                  (high == null || high.variable != varJ)):
            swapWasMade = false
        # Caso e:
        else if ((low == null || low.variable != varJ) && high == null):
            swapWasMade = false

        return swapWasMade

# Intercambia un nivel con el siguiente
swap(int level):

        # Si es el último nivel, ignorar
        if(level == variables.size()-1)
```

```
        return false

    variableI = variables.getVariableInPosition(level)
    variableJ = variables.getVariableInPosition(level+1)

    boolean swapWasMade = false

    verticesOfThisLevel = V[variableI]
    for(Vertex v : verticesOfThisLevel):
        swapWasMade = swapWasMade || swapVertex(v, variableJ)

    variables.swap(variableI, variableJ);
    return swapWasMade
```

Algoritmos

Con el objetivo de reducir el número de vértices, hemos desarrollado una serie de algoritmos que hacen uso del intercambio de variables.

Window Permutation

Este algoritmo está basado en los trabajos [Fujita91] y [Ishiura91]. Consiste en definir una ventana móvil de un determinado tamaño sobre las variables ordenadas y obtener el mejor orden posible en cada posición. En cada iteración en la que se mueve la ventana se almacena la mejor solución y al final, se devuelve ésta.

Es decir, se obtiene para cada posición de la ventana la permutación que genera un orden cuyo tamaño de árbol es mínimo.

Rudell, en [Rud93] indica que este algoritmo es eficiente para tamaños de variables 4 ó 5, pero sus resultados indican que su algoritmo es mejor.

Sifting de Rudell

Richard Rudell en el año 1993 desarrolló un algoritmo de reducción del número de vértices en árboles BDD, [Rud93]. El algoritmo consiste en encontrar la posición óptima de una variable en el árbol, suponiendo que las demás variables están fijas.

Así, comenzamos con una variable determinada y movemos sus vértices hacia el fondo del árbol y luego de vuelta hacia la primera posición. En cada paso vamos calculando el número de vértices que tiene el árbol, de forma, que cuando se termine el proceso se sepa exactamente cuál es la posición que genera un tamaño de árbol menor.

Como ya sabemos la mejor posición, sólo tendremos que subir o bajar los vértices de esa variable hasta encontrar la posición antes encontrada.

Como se puede ver, este algoritmo heurístico es sencillo y da unos resultados muy buenos para problemas con pocas variables.

Un aspecto importante a tener en cuenta es el de cómo ir escogiendo las variables para ir realizando los intercambios. Nosotros hemos optado por tener varias estrategias: ir escogiendo las variables según su número de vértices, tal y como se recomendaba en [??] o tomar la variable que más ocurrencias tenga en la fórmula o tomar una variable al azar.

Algoritmo genético

El algoritmo genético fue desarrollado después de leer el trabajo [Lenders&Baier04] pensé que sería una buena idea implementarlo y compararlo a las heurísticas ya comentadas.

Estos algoritmos se basan en la metáfora biológica de que una población irá mejorando poco a poco si vamos realizando cruces entre ellos y mutaciones esporádicas. De esta forma, después de un número de iteraciones

En nuestra implementación, hemos optado por un algoritmo genético clásico, que no contuviera hibridación. De forma que contiene las operaciones clásicas de este tipo de algoritmos.

Antes de comenzar a describir sus partes, hemos de destacar un detalle sobre el motor de números aleatorios. Este generador usa la misma semilla para todas las operaciones aleatorias, de forma que sean reproducibles los experimentos. Es decir, que si ejecutamos con las mismas semillas los experimentos, devolverán siempre los mismos resultados.

Estructuras de datos

Hemos optado porque cada cromosoma sea un orden de variables. Cada cromosoma tendrá el orden y el número de vértices que se obtienen aplicando ese orden al BDD. Así podremos usar este número de vértices como medida de la bondad del cromosoma, evidentemente, cuanto menor sea este valor, mejor orden será.

Población inicial

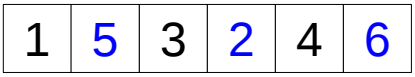
De esta forma, cuando generamos inicialmente una población de cromosomas, lo que hacemos es generar órdenes aleatorios de variables y reordenar nuestro árbol para calcular el tamaño de éste. Es obvio que este proceso es costoso, y su tiempo dependerá en gran medida del número de cromosomas que vayamos a usar, dado que para aplicar al árbol el orden de cada uno de ellos, habrá que realizar como mucho para cada variable *n-1* intercambios, por lo que tendremos un orden de eficiencia cuadrático.

Selección

Después, en la selección, se toman aleatoriamente un número determinado de padres. No se optimizan usando el algoritmo de Rudell como en [Lenders&Baier04], porque pensamos que aplicar una heurística aquí podía hacer que se llegase a un máximo local.

Cruce

Estos cromosomas seleccionados se emparejan dos a dos para generar, mediante cruces, los cromosomas hijos. Este cruce sí ha sido tomado de [Lenders&Baier04] y se basa en intercalar las posiciones de cada una de las variables de los padres, sin repetir variables, claro. Por ejemplo, si tenemos los siguientes órdenes, para generar su descendiente iríamos intercalando cada una de las variables:



Nótese que cuando se va a escoger la variable 5 del cromosoma de la izquierda, ya se le ha asignado una posición a esta variable, debido a que se encontraba en el frente del cromosoma de la derecha. Ante este problema, hemos de continuar buscando una variable del mismo padre que no se encuentre ya en el cromosoma hijo. En ese caso concreto, le asignamos la variable 4, dado que es la primera del cromosoma padre que todavía no tiene el hijo.

Mutación

La operación de mutación se ejecuta sólo con una determinada probabilidad y consiste en intercambiar las posiciones de dos variables. Obviamente, puede que en un caso concreto no se cambie ninguna variable de sitio, pero si se produce algún cambio habrá que recalcular el tamaño de grafo que genera.

Actualización de la población

Para terminar la iteración, se actualiza la población. Primero se añaden a la población los hijos anteriormente generados. Luego eliminamos de la población los peores cromosomas hasta que al final, la población permanezca con el mismo número de cromosomas que tenía inicialmente.

Software

Herramientas usadas

Para ejecutar el software proporcionado se necesita una máquina virtual de Java con la versión 7. Concretamente este paquete se ha desarrollado usando OpenJDK7 [OpenJDK].

Dada la naturaleza multiplataforma de Java, no se requiere de ningún sistema operativo específico, ni enlazado con librería externa para su uso, dado que están integradas en el paquete jar *store/DJBDD.jar*.

A continuación, vamos a detallar las librerías usadas, dando un fundamento al motivo que nos ha llevado a hacerlas parte integral de nuestro proyecto:

antlr3

Antlr proporciona la posibilidad de crear reconocedores para gramáticas. En nuestro caso lo hemos usado para proporcionar a nuestra librería de la capacidad de construir árboles a partir de una expresión lógica con la misma sintaxis que una expresión lógica en el lenguaje de programación Java.

Recordemos que la construcción del BDD se realizaba de forma recursiva realizando *Apply* entre sucesivos árboles generados a partir de la expresión lógica. Pues bien, el proceso es el siguiente, tomamos la expresión lógica, la convertimos en un árbol sintáctivo abstracto usando antlr3 y vamos recorriendo ese árbol, generando BDDs y vamos operándolos entre ellos hasta obtener uno que contenga toda la expresión completa.

Este paquete contiene herramientas muy sencillas que permiten la creación de un reconocedor sintáctico a partir de una gramática como la que hemos usado:

```
grammar Logic;


options {
  output=AST;
}
parse
  :  expression EOF!    // omitir token EOF
  ;
expression
  :  dimplication
  ;
dimplication
  :  isdifferent ('<->'^ isdifferent)*    // `<->` es equivalencia o doble imp.
  ;
isdifferent
  :  implication ('!='^ implication)*    // `!=` es lo contrario de  <->
```

```
  ;

implication
  :  notimplication ('->'^ notimplication)*    // `->` es la implicación
  ;
notimplication
  :  or ('!->'^ or)*    // `!->` es la implicación negada
  ;
or
  :  and ('||'^ and)*    // `||` es el O lógico
  ;
and
  :  not ('&&'^ not)*      // `&&` es el Y lógico
  ;
not
  :  '!'^ atom    // `!` es el operador de negación
  |  atom
  ;
atom
  :  ID
  |  TRUE
  |  FALSE
  |  '('! expression ')'!    // los paréntesis no tienen valor sintáctico
  ;
TRUE  : 'true';   // True y False los tomamos como palabras reservadas
FALSE  : 'false'; //
ID    : ('a'..'z' | 'A'..'Z' | '0'..'9' | '_' | '{' | '}')+; // Variable lógica
Space : (' ' | '\t' | '\r' | '\n')+ {$channel=HIDDEN;}; // Caracteres vacíos
```

Para facilitarnos las cosas, se puede ver como hemos incluido cada operador y su negación. Esto permitirá poder negar un árbol completo de forma sencilla sin usar aristas etiquetadas. En siguientes secciones se profundizará en este detalle.

Con respecto a la versión, conocemos que existe una versión nueva de *antlr* (la versión 4), pero dado que nuestra gramática es sencilla y generativa por la derecha, no veíamos la necesidad de usar la nueva versión porque entre otras cosas, éramos familiares a la 3.

Javaluator

Tal y como hemos visto con el apartado de construcción y el anterior, la generación se realiza de forma recursiva generando un árbol sintáctico y generando BDDs y operando sobre ellos hasta obtener el BDD de la raíz, que es el que representa el árbol binario de toda la expresión.

Ahora bien, para dotar de mayor flexibilidad al desarrollador, permitimos que el proceso recursivo de creación del árbol no requiera de un análisis sintáctico, ni del uso de la operación *Apply*, sino simplemente de la creación implícita de la tabla de verdad, evaluando la expresión lógica con todas las posibles combinaciones de asignaciones de valores de verdad para las variables.

Obviamente esta opción no es deseable y es ineficiente, por lo que está desactivada por defecto y sólo debería usarse para expresiones lógicas triviales o cuando tengan un número lo suficientemente pequeño de variables como para ser eficiente, dado que probar todas las combinaciones de n variables implica realizar 2^n evaluaciones.

Diseño

A nivel de diseño software, a continuación se muestran los paquetes ordenados de forma alfabética y desarrollamos en cada uno de ellos la funcionalidad que contiene.

Paquete raíz: djbdd

Paquete raíz que contiene a todos los de la librería djbdd.

core

Este es el módulo principal, contiene todas las clases básicas de la librería. Sin estas clases, no habría sistema. Inicialmente se introdujeron estas clases en el paquete predeterminado, pero debido a que se vio que era una mala práctica, se creó un paquete por defecto en el que estuvieran todas. Este es ese paquete.

A continuación, describimos cada una de ellas de la forma más detallada posible en el orden lógico de uso:

VariableList

Lista de variables. Inicialmente se pensó en hacer que la lista de variables no fuera una clase sino simplemente una colección de objetos `String`. Eso se vio que no era propicio debido a que queríamos tener por un lado las variables, cada una con su índice y por otro el orden que se le había dotado en función a los intercambios realizados.

Por tanto, esta clase tiene dos estructuras principales (y una para facilitarnos el desarrollo, pero que no es vital):

- `variables`: lista de variables donde el índice indica el identificador de la variable. Este atributo no cambia nunca en el transcurso de la vida del sistema.
- `orderedVariables`: lista de variables de acuerdo al orden. Se usa para depuración, ya permite tener las variables en el orden natural. Cuando se intercambien dos variables, esta lista tomará el orden natural de las variables.
- `order`: indica el orden de cada variable, de forma que el valor en la posición i indica el orden de la variable i-ésima. Inicialmente, cada valor contiene la posición en la que se encuentra. En cuanto se empiecen a realizar intercambios de variables, los valores cambiarán a la posición donde se encuentre realmente.

Vertex

Esta clase contiene la estructura de cada uno de los vértices del

Tiene los siguientes atributos:

- `index`: clave única en la tabla *T*.
- `variable`: identificador de la variable que contiene en la lista de variables. Los vértices hoja la tienen asignada a un valor fijo que no representa a ninguna variable.
- `low`: referencia al descendiente *low*. Para los vértices hoja es *null*.
- `high`: referencia al descendiente *high*. Para los vértices hoja es *null*.
- `num_parents`: contador de vértices que apuntan a este vértice. Dado que usamos una misma red de vértices para todos los BDDs, tenemos que llevar la cuenta de cuándo un vértice no es apuntado por ningún otro.
- `num_rooted_bdds`: indica para cuántos BDDs es este vértice la raíz. De forma similar a `num_parents`, este atributo guarda la cuenta de cuántos BDDs lo usan como raíz, debido a que se comparte toda la malla de vértices.

Además de estos atributos, el vértice tiene las siguientes operaciones destacadas:

- `isLeaf`: informa si el vértice es un vértice hoja.
- `isRedundant`: informa si sus atributos *low* y *high* son iguales.
- `isDuplicate`: informa si dos vértices son duplicados, es decir, si comparten los mismos atributos variable, *low* y *high*.
- `isOrphan`: informa si el vértice es huérfano, es decir, si no hay ningún BDD que lo tenga como raíz y si no hay ningún vértice que apunte a él. En ese caso, este vértice puede eliminarse.
- `isChildOf/isParentOf`: comprobación de descendencia/paternidad (respectivamente). Estos métodos permiten saber si un vértice que se pasa como parámetro es hijo/padre (respectivamente) del vértice actual.

Por supuesto hay operaciones de conversión a cadena y de establecimiento de variable, *low* y *high* del vértice. Con respecto a esta última operación, hemos de notar que no es pública (al contrario del resto) sino que es propia del paquete puesto que se hace uso en la operación de intercambio de variables, implementada en la clase `TableT`.

TableT

Esta clase contiene las estructuras que definen el grafo global en el que están basados todos los BDDs.

En primer lugar, hay una tabla *T*, que contiene cada vértice dado su índice. Es decir, es una tabla hash en la que para cada vértice, se guarda una entrada en la que la clave es su atributo *index* y el valor es el propio vértice.

En segundo lugar, hay una tabla *hash* que almacena de forma única todos los vértices, de manera que nunca haya dos vértices con el mismo trío de valores variable, *low* y *high* que sea igual. Así, la clave es la concatenación sigue estas reglas:

- Si es el vértice hoja **1**: `"-1-N-N"`
- Si es el vértice hoja **0**: `"-2-N-N"`
- Si es otro tipo de vértice: `variable+"-"+low.index+"-"+high.index`

A esta tabla *hash* la llamamos *U* (de tabla de unicidad de vértices).

En tercer lugar, hay una estructura que agrupa todos los vértices en conjuntos según la variable que contiene cada uno. Implementamos esta estructura de grupos de vértices por variables como una tabla *hash* en la que la clave es la variable y el valor es una lista con todas las variables.

Como veremos en cuando expliquemos las distintas alternativas del algoritmo de intercambio de variables de Rudell, necesitamos tener un acceso rápido a los vértices de cada nivel. Obviamente, en cada intercambio, tendremos que cambiar a los vértices del conjunto que hayamos adelantado un nivel, y por tanto, cambiarles también su variable.

Esta estructura la llamaremos *V*, porque agrupa a los vértices por variables.

Ahora, vamos a mostrar un ejemplo de la estructura que se tendría en memoria para el árbol de la función $F=x_0x_2+x_1x_3+x_4$ siendo el orden de las variables el indicado por el subíndice de cada una de ellas. Así, tenemos el BDD representado gráficamente a continuación:

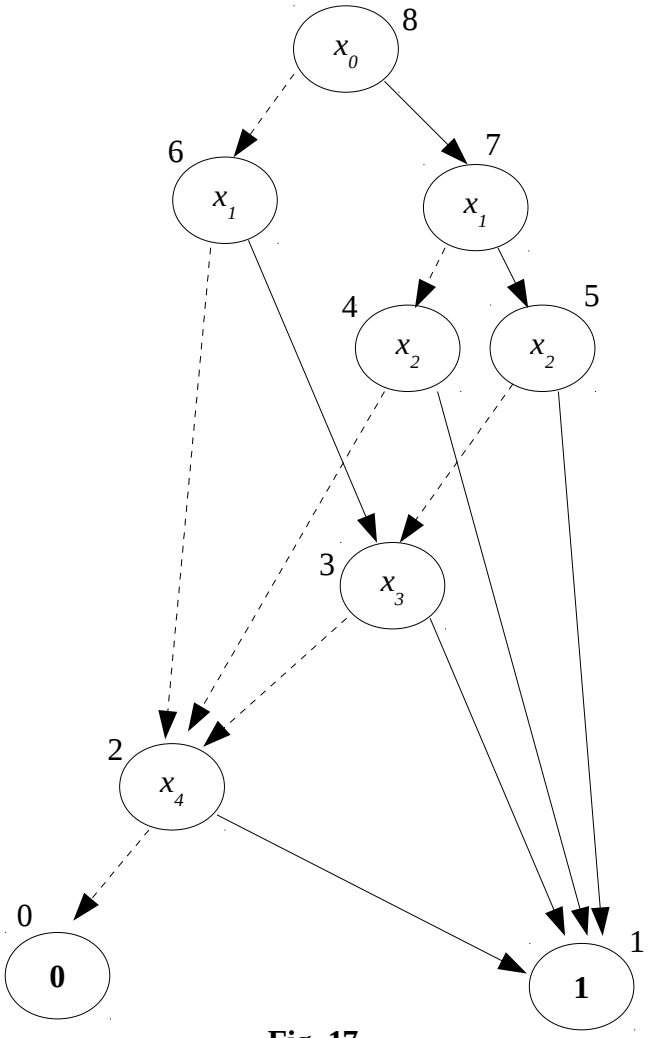


Fig. 17

Nótese que las variables ahora comienzan en `0`, dado que en nuestra implementación, las variables son el índice de posición en la `VariableList`, comenzando en `0` y terminando en `n-1`, siendo `n` el número de variables.

Mostramos a continuación los valores de cada una de las tablas para el BDD con la expresión lógica $F=x_0x_2+x_1x_3+x_4$:

Tabla T			
Índice	**Variable**	**Low**	**High**
0	-1	*null*	*null*
1	-2	*null*	*null*
2	4	0	1
3	3	2	1
4	2	2	1
5	2	3	1
6	1	2	3
7	1	4	5
8	0	6	7

Tabla U	
Clave	**Vértice**
-2-N-N	0
-1-N-N	1
4-0-1	2
3-2-1	3
2-2-1	4
2-3-1	5
1-2-3	6
1-4-5	7
0-6-7	8

Tabla V	
Variable	**Vértices**
0	{8}
1	{6, 7}
2	{4, 5}
3	{3}
4	{2}

Una vez mostrado el ejemplo, queda un aspecto importante que tratar sobre la implementación de

estas tablas. Dado que en las operaciones sobre los BDDs, hay que liberar vértices, estas tablas deberían reflejar esas eliminaciones. Para hacerlo más sencillo, estas tablas no tienen referencias a vértices, sino referencias débiles a objetos de la clase `Vertex`, vértices en definitiva, [JavaWeakReferences] [UnderstandingWeakReferences]. Esto es, si el vértice no es referenciado por nadie más que por estas tablas, el recolector de basura puede eliminarlo. O dicho de otra forma, estas referencias no tienen poder como para indicar al recolector de basura que no han de eliminar a los vértices a los que apuntan.

Así, en cuanto un vértice deje de estar apuntado por otro, el recolector de basura de la JVM tendrá capacidad para eliminar el objeto, y por tanto, liberando la memoria usada.

En cambio, ahora, lo único que hace falta es liberar de forma periódica las referencias débiles que no apuntan a nada. El problema que nos encontramos con este enfoque fue debido a la implementación del algoritmo de intercambio de niveles. Así que eliminamos de forma explícita los vértices cuando llamamos al método de recolección de basura de esta clase, y de ahí que usemos en `Vertex` un contador de padres y de BDDs que lo usan como raíz. El método de recolección de basura implementado para solventar este problema, se llama `gc`.

Como pudimos comprobar, no se puede confiar en que al dejar un objeto sin referencias vaya a ser eliminado por el recolector de basura, dado que éstos están configurados de manera que si no se necesita memoria en el *heap*, no se realiza la acción de eliminar datos sin referenciar.

En cuanto a métodos destacados de este clase, está el método de recolección de basura antes comentado, métodos de comprobación de existencia dado un vértice, métodos de adición de vértices a las tres tablas y por supuesto, métodos de eliminación de vértices.

Tenemos que destacar el método de intercambio de un nivel con el siguiente, dado que este método se usa en los algoritmos de reducción por intercambio de variables que hemos implementado. Además, esta clase, llevará la cuenta de cuántas veces se ha realizado la operación de intercambio, dado que usaremos ese dato para comprobar la efectividad de cada método.

Por último, notamos que hay métodos que imprimen en un archivo de texto en formato DJBDD y en PNG el grafo completo.

BooleanEvaluator

Esta clase encapsula la llamada a Javaluator y permite que se pueda evaluar una expresión lógica de forma sencilla. Esta clase hereda de `AbstractEvaluator` que es una clase abstracta de Javaluator. Por supuesto, define todos los operadores lógicos posibles, su precedencia y su asociatividad. Por último, sobrescribe el método `evaluate`, para que devuelva la semántica correcta de cada operación lógica.

Cabe destacar, que el método de evaluación es un método estático llamado `evaluateFunction` que prepara la expresión sustituyendo cada variable por su valor de verdad y realizando la llamada a `evaluate`. Así, ocultaremos todo lo relacionado con Javaluator al resto de componentes de este software, logrando una buena separación entre niveles de abstracción.

BDD

Esta clase representa a un BDD concreto. Es decir, a un expresión lógica en forma de árbol binario de decisión.

Como atributos, cada árbol está determinado por un vértice raíz, un subconjunto de variables de las

definidas como posibles variables de la expresión y su tamaño.

Estas variables permiten optimizar las operaciones lógicas con otros BDDs, además del propio proceso de construcción. El motivo de esto son las expresiones lógicas que no tienen todas las variables. Por ejemplo, en determinadas fórmulas, se usaban sólo una decena de variables, pero como habíamos definido más de 4000, el proceso de construcción era muy lento sin razón de serlo. Cuando aplicamos esta optimización, el proceso de construcción fue mucho más rápido.

El tamaño de un BDD ha de consultarse visitando todos los vértices que cuelgan del vértice raíz. Este proceso es costoso, de manera que una vez calculado, se guarda para futuras consultas.

Por último, la última optimización que le introduje fue la de almacenar si era un árbol tautológico o contradictorio, es decir, si el árbol devolvía siempre verdad o siempre falso, respectivamente.

BDDApply

Esta clase contiene la implementación del algoritmo *Apply* para ejecutar operaciones entre dos BDDs.

Cabe destacar que se usa una tabla de caché para no volver a computar las operaciones varias veces sobre los mismos vértices, pero podemos decir que es una implementación muy fiel al algoritmo antes mostrado.

GCThread

Esta clase contiene la implementación de una hebra que permite la ejecución de forma concurrente de un recolector de basura.

Actualmente, la implementación de BDDs se basa en referencias débiles, tal y como hemos visto anteriormente. Pero esas referencias débiles han de ser eliminadas puesto que a fin de cuentas, ocupan espacio. Pues bien, esta clase permite al desarrollador olvidarse de la tarea de ir llamando al recolector de basura del sistema de BDDs, pudiendo configurar además la frecuencia de llamada de éste de forma dinámica.

Además, se ha desarrollado esta funcionalidad con una estructura orientada a objetos, para poder tratar con ella de la forma más correcta desde el punto de vista de la Ingeniería del Software.

io

Siguiendo la convención de Java, este módulo contiene utilidades de entrada y salida. Este módulo contiene clases para tratar con cada uno de los tipos de fichero desde los que se puede cargar un BDD.

Así, tenemos los siguientes tipos de cargadores:

- `CstyleFormulaFileLoader`: carga un BDD a partir de una expresión en estilo C.
- `DimacsFileLoader`: carga un BDD a partir de un archivo de cláusulas en formato DIMACS.
- `SheFileLoader`: carga un archivo a partir de un archivo en formato de Stephen She.

Adem á de estos cargadores de BDDs desde archivos, existe una clase que permite generar un archivo PNG usando la librería GraphVizJava. Esta clase es `Printer`. Por ejemplo, el siguiente gráfico ha sido generado usando esta funcionalidad:

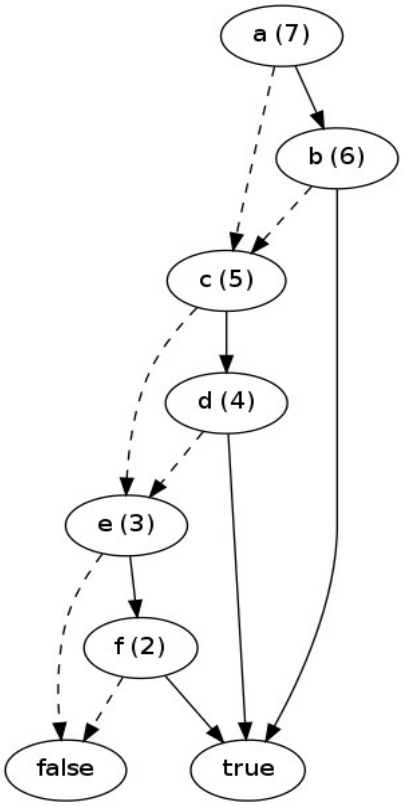


Fig. 18

Nótese que en el diagrama generado a los vértices hoja se les llama *false* y *true* para representar el vértice **0** y el **1** (respectivamente). Y que el resto de vértices contiene la variable y el identificador único de cada vértice entre paréntesis.

El resto de clases son ayudantes internos que permiten descargar de funcionalidad a las clases principales y que no tienen mucha más relevancia.

logic

Paquete que contiene métodos con el analizador sintáctico de fórmulas basado en antlr3. Contiene las clases generadas por antlr3, `LogicLexer` y `LogicParser` y un analizador desarrollado por mí que hace uso de éstas, `FormulaParser`. También hay una clase con ejemplos de uso que puede ignorarser completamente.

`FormulaParser` devuelve un árbol de sintaxis con los símbolos terminales y operadores de la fórmula. Este árbol lo usaremos en la construcción del BDD que hace uso de *Apply*.

main

Módulo principal, contiene el controlador que procesa los argumentos pasados desde línea de órdenes y realiza las llamadas a cada uno de los módulos según las órdenes que le pase el usuario.

Si se ejecuta sin argumentos, se muestra un pequeño manual de uso con todas las opciones que puede usar.

reductors

Este paquete contiene cada uno de los algoritmos de reducción del tamaño del BDD.

Cada método de reducción ha de heredar de la clase `ReductionAlgorithm`, que tiene el método run que ha de ser sobrescrito por el desarrollador que quiera implementar un método de reducción.

Además esta clase contiene varias referencias a objetos globales para hacer la vida más sencilla al desarrollador, guarda una referencia de la tabla global T y de las variables. Por ende, la reducción se aplica al grafo total con todos los vértices. Esto es muy útil en el sentido de que podemos generar varios árboles BDD y mientras estén en memoria, si no los destruimos, si aplicamos algún método de reducción podremos comprobar cómo se reduce el tamaño de todos.

Los métodos de reducción implementados son los siguientes:

- `WindowPermutationReductor`: reduce el tamaño de los BDDs usando el algoritmo [Fujita91]. Lo llamamos con el nombre que le dio R. Rudell: Algoritmo de Permutación en una ventana.
- `SiftingReductor`: aplica el método de reducción de Rudell [Rudell93] sobre todos los BDDs que existan en el sistema.
- `GeneticReductor`: esta clase contiene la implementación del algoritmo genético de Lenders y Baier [Lenders&Baier04].
- `TotalSearch`: realiza una búsqueda total. Para cada posición de variable, prueba todas las demás, así, al finalizar el algoritmo, obtendremos la mejor solución.

test

Pruebas de funcionamiento del paquete de BDD. Consta de 18 pruebas que pueden ejecutarse con la opción `–-runtest` seguida del número de test que deseemos.

Cada test es estático e independiente. No produce resultados especiales, sino que el desarrollador ha de saber de antemano qué test está ejecutando para poder interpretar correctamente la salida que éste genera. Normalmente se tratan de impresiones de un BDD o de mensajes que indican por sí mismos si el tests es correcto o ha fallado.

Usamos este paquete como método rápido de probar fórmulas lógicas convenientes durante el proceso de desarrollo del software.

Nos habría gustado que el mismo test indicase si es válido o no e implementar tests unitarios, pero nuestro conocimiento en ese particular es limitado y el tiempo era limitado, por lo que no se ha hecho.

timemeasurer

En Java no existe un método canónico de medición de tiempos y queríamos encapsularlo en algún tipo de unidad funcional. Por tanto, hemos creado una clase, `TimeMeasurer`. Esta clase empieza a contar el tiempo desde que se construye una instancia de ella y termina, al ejecutar el método `end`. Como se puede ver, el nivel de encapsulación llega hasta las mediciones de tiempo.

graphvizjava

Este paquete contiene una clase realizada por Laszlo Szathamary, que permite tratar con GraphViz

[GraphViz] de forma sencilla desde Java. Usamos esta utilidad, porque permite de forma sencilla convertir un archivo PNG a partir de un archivo `dot` [DOT]. Usamos el formato `dot` como formato intermedio para mostrar los BDDs.

Para ver más información sobre este software, vea [GraphvizJava], donde se incluye todo el paquete de software original, los créditos completos y más información adicional.

logger

Contiene un sistema de registro de mensajes centralizado que permite activar y desactivarlo en tiempo de compilación. Si se desactiva antes de compilar este módulo, no se ejecutarán las condiciones de comprobación puesto que Java comprobará que la condición para mostrar los mensajes es falsa de forma constante, por lo que no se producirá una pérdida de rendimiento. Eso sí, este módulo no se puede activar y desactivar en tiempo de ejecución, por lo que sugiero realizar todas las pruebas primero, desactivarlo y finalmente generar el `.jar`.

Sólo contiene una clase, la clase Log que contiene varios métodos de impresión de mensajes con nombres exactamente iguales a los de `System.out`, como son `print`, `println`, etc.

Implementación

La mayoría de las implementaciones actuales están realizadas en código C o C++ de baja calidad. De hecho, en la mayoría de los trabajos encontrados, no se preocupan de detalles de cómo está implementada la operación swap, de manera que a la hora de mostrar resultados, simplemente se desarrolla un algoritmo basado en esta operación y se toman tiempos.

He optado por realizar una implementación libre y gratuita que pudiera usarse como laboratorio por parte de investigadores.

La implementación se encuentra disponible para descargar bajo licencia libre GPL3 con excepción de uso de clases en https://github.com/diegojromerolopez/djbdd.

Manual de uso de los binarios

Imprimir el BDD como imagen:

```
java -jar DJBDD.jar --image --<format> <file>
```

Imprimir el BDD a un archivo:

```
java -jar DJBDD.jar --print --<format> <file>
```

Reducir de tamaño un BDD definido en un fichero (sólo se admiten formato DIMACS y cstyel):

```
java -jar DJBDD.jar —-reduce --method=<algorithm> --<format> <file> <output_file>
```

Donde `<algorithm>` es uno de los siguientes algoritmos:

- rudell-order-by-occurence
- rudell-random-order
- simulated-anealing
- window

- genetic
- memetic

Y `<output_file>` es el fichero de salida.

Formatos soportados (parámetro `<format>`)

- **dimacs**: formato de cláusulas en forma normal conjuntiva. Por ejemplo:

```
c  simple_v3_c2.cnf
c
p cnf 3 2
1 3 0
-2 3 -1 0
```

  Equivale a la fórmula `(x1 OR x3) AND (NOT x2 OR x3 OR NOT x1)`.

- **cstyle**: expresiones booleanas al estilo de C, usando como operadores lógicos el &&, `||`, `->`, `<->` y el `!`, para el AND, OR, implicación y doble implicación, respectivamente. Deben estar precedidas por una línea con todas las variables separadas por comas, dadas en el orden el que han de tomarse. Ejemplos de expresiones válidas son los siguientes:
  - `a && !b`
  - `a -> (!b && (c || d))`
  - `a <-> (b && !c)`
  - `a != (b && !c)`
  - `x0 && x1 && !x2`
- **djbdd**: archivo DJBDD. Formato de texto en el que se pueden guardar los árboles BDD en este paquete. Este formato no se soporta para la llamada al procedimiento de reducción.

Ejemplos de uso de la API

```
// Variables booleanas
String[] variables={"a", "b", "c", "d", "e", "f"};

// Siempre ha de iniciar el sistema BDD antes de crear
// cualquier objeto BDD. Si no, el sistema dará error
BDD.init(variables);

// Las funciones se especifican con la sintaxis de operadores lógicos de Java,
// añadiendo además los operadores -> (operador de implicación)
// y <-> (operador de doble implicación)
String function = "(a && b) || (c && d)";

// Con la implementación actual, el orden de las variables
// es global a todos los BDDs, por lo tanto, si queremos cambiarlo
```

```
// hemos de usar la operación de intercambio de niveles,
// que intercambiará dos niveles para todos los BDDs

// Construcción de un nuevo BDD
BDD bdd = new BDD(function);

// Mostrar el BDD bdd en la salida estándar
bdd.print();

// Puede imprimir el BDD como una imagen PNG
Printer.printBDD(bdd1, "bdd_"+bdd1.size());

// Otro BDD
String function2 = "(a && e) || f"
BDD bdd2 = new BDD(function2);

// Muestra el BDD en el terminal
bdd2.print();

// Operaciones entre BDDs

// Este BDD es el AND lógico entre bdd y bdd2
BDD bdd3 = bdd.apply("and", bdd2);

// Este BDD es el OR lógico entre bdd y bdd2
BDD bdd4 = bdd.apply("or", bdd2);

// Destruye explícitamente bdd2
bdd2.delete();

// Se puede liberar memoria de nodos “muertos” llamando
// a esta función de recolección de basura
BDD.gc();
```

Experimentos y resultados

Hemos realizado diversos experimentos con los algoritmos de reducción.

En [SATLIBBMK] hay una serie de benchmarks que se usan para comprobar la eficiencia de determinados sistemas de comprobación de satisfacibilidad.

Problemas no satisfacibles

La librería se comporta muy bien, y es capaz de comprobar la satisfacibilidad de un problema concreto. Evidentemente, no buscábamos eso cuando la desarrollábamos, pero es una muestra más de que el trabajo es correcto.

Una fórmula no satisfacible es una contradicción, por lo que al construir el árbol, dado que nos ha de devolver una representación de los caminos posibles según los valores de las variables, ha de darnos el único camino que hay: el vértice *false*. Este vértice indica que independientemente de los valores que se asignen a las variables de esa fórmula, se va a obtener un resultado negativo siempre.

Problemas satisfacibles

Todas las fórmulas que construimos y optimizamos son satisfacibles por los motivos anteriormente comentados.

Hemos tomado las 50 primeras cláusulas lógicas de los distintos archivos DIMACS y hemos trabajado sobre ellas, debido a que no buscábamos un software eficiente a nivel de poder competir con los paquetes que hay en la red, sino proporcionar un entorno con las más avanzadas características de software de ingeniería y que además, nos permitiera estudiar el proceso de construcción y optimización de un árbol BDD.

Problemas abiertos

Bibliografía

[Andersen1999] An Introduction to Binary Decision Diagramas. H. R. Andersen. Lecture notes for Efficient Algorightms an Programs. IT Universidad de Copenhague. 1999.

[antlr3] Versión 3 del analizador de lenguajes. http://www.antlr3.org/

[Antlr3Book] The definitive ANTLR reference. Terence Parr.

[Bryant1986] Graph-Based Algorithms for Boolean Function Manipulation, Randal E. Bryant, 1986

[Bryant1992] Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams, Randal E. Bryant. Acm Computing Surveys. 1992.

[DIMACS] DIMACS CNF formula format. http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html

[Somenzi] Binary Decision Diagrams. Fabio Somenzi. Department of Electrical and Computer Engineering. University of Colorado at Boulder.

[ImpBDD] Efficient implementation of a BDD package, Karl S. Brace, Richard L. Rudell, Randal E. Bryant.

[ImpBDDStornetta] Implementation of an Efficient Parallel BDD Package. Tony Stornetta, Forrest Brewer.

[Rud93] *Dynamic variable* ordering for ordered binary decision diagrams. Richard L. Rudell 1993.

[Rud93W] BDDs: Implementation Issues & Variable Ordering. Richard Rudell 1993.

[JacobiEtAl] Incremental Reduction of Binary Decision Diagrams. R. Jacobi, N. Calazans, C. Trullemans.

[OpenJDK7] Plataforma de referencia de desarrollo Java de código abierto. http://openjdk.java.net/

[Javaluator] Librería de evaluación Java. http://javaluator.sourceforge.net/en/home/

[LICS] Logic in Computer Science. M. Huth, M. Ryan. Cambridge University Press.

[GraphViz] Graphviz, Graph Visualization Software. http://www.graphviz.org/

[GraphVizJava] Paquete de software de uso de GraphViz desde Java, por Laszlo Szathmary. http://www.loria.fr/~szathmar/off/projects/java/GraphVizAPI/index.php

[DOT] Formato del lenguaje DOT. http://www.graphviz.org/doc/info/lang.html

[UnderstandingWeakReferences] Understanding Weak References, por Ethan Nicholas. https://weblogs.java.net/blog/enicholas/archive/2006/05/understanding_w.html

[Java7API] Documentación de la API de Java 7. http://docs.oracle.com/javase/7/docs/api/

[JavaWeakReferences] Documentación de la API de Java 7 sobre referencias débiles (clase WeakReference). http://docs.oracle.com/javase/7/docs/api/java/lang/ref/WeakReference.html

[ABDDOpt] Advancesd BDD Optimization. Rüdiger Ebendt, Görschwin Fey & Rolf Drechsler.

[BP&BDD] Branching Programs and Binary Decision Diagrams: Theory and Applications. Ingo Wegener.

[BDDT&I] Binary Decision Diagrams: Theory and Implementation. Rolf Drechsler.

[ALGOVLSI] Algorithms and Data Structures in VLSI Design. C. Meinel & T. Theobald.

[Sasao96] Representations of Discrete Functions. Tsutomu Sasao.

[Hassoun&Sasao] Logic Synthesis and Verification. Soha Hassoun, Tsutomu Sasao.

[SATLIBBMK] SATLIB Benchmark problems. http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

[Fujita91] Fujita, M., Matsunaga, Y.,Kakuda, T., On Variable Orderings of Binary Decision Diagrams for the Application of Multi-Level Logic Synthesis, Proceedings of the European Conference on Design Automation, 1991.

[Ishiura91] N. Ishiura, H. Sawada, S. Yajima. Minimization of Binary Decision Diagrams Based on Exchanges of Variables. En Proceedings International Conference on Computer-Aided Design. 1991.

[Lenders&Baier04] Genetic Algorithms for the Variable Ordering Problem of Binary Decision Diagrams. Wolfgang Lenders, Christel Baier.

[Dijk2012] The Parallelization of Binary Decision Diagram operations for model checking. Tom van Dijk. Tesis. 2012.