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

En este trabajo se muestra una librería de uso de BDD basada en Java.

Lógica proposicional

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
- $\neg q$, donde q es una proposición y $\neg$ 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. En este trabajo nos centraremos en la interpretación semántica de las fórmulas dejando para el lector la demostración sintáctica, dado que está demostrado (REF) que son equivalentes.

Operador de negación

p	$\neg p$
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 AND 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 excribirse yuxtaponiendo directramente las variables, de forma que se escribirían una destrás 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 OR 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	$p \rightarrow q$	$\neg p$ OR 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	$p \leftrightarrow q$	*(p* AND *q)* OR *($\neg p$* AND *$\neg q$)*
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 [REF] 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
- →

De todas formas, salvo en el caso de NOT, vamos a incluir paréntesis para evitar confusiones entre las distintas precedencias.

Propiedades

Propiedad asociativa

Propiedad distributiva

Leyes de De Morgan

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[1], 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.

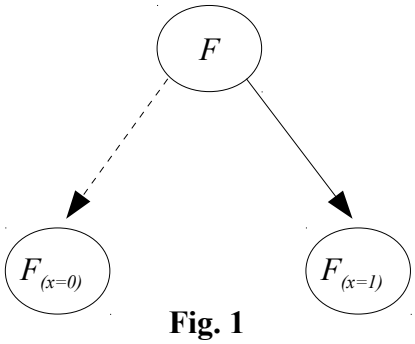


Fig. 1

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:

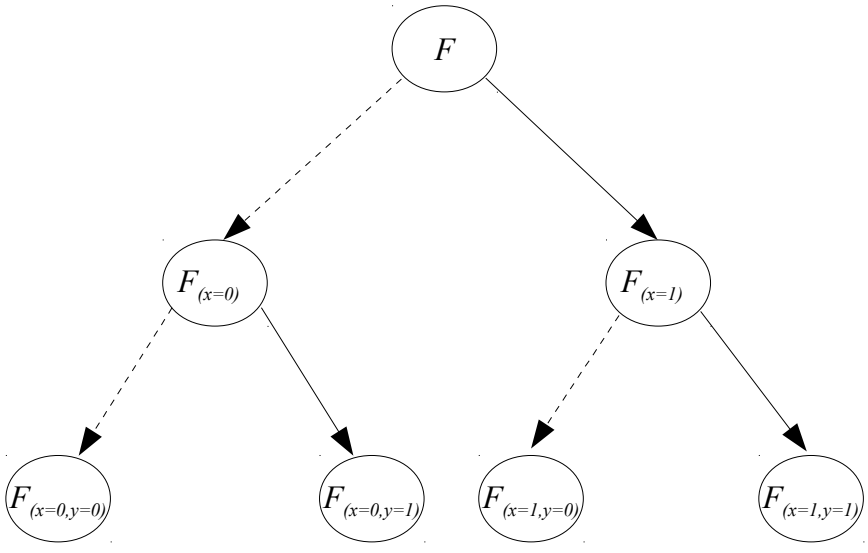


Fig. 2

1 También se le suele llamar expansión de Shannon, aunque su creador fue George Boole, matemático, lógico y filósofo inglés del siglo XIX.

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 $F_{(a,b)} = F_{(x=a,y=b)}$.

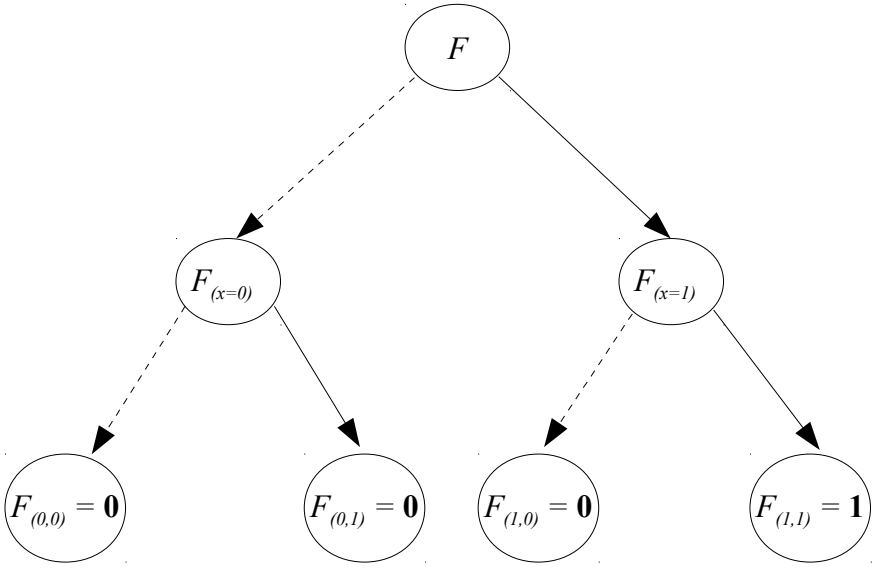


Fig. 3

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.

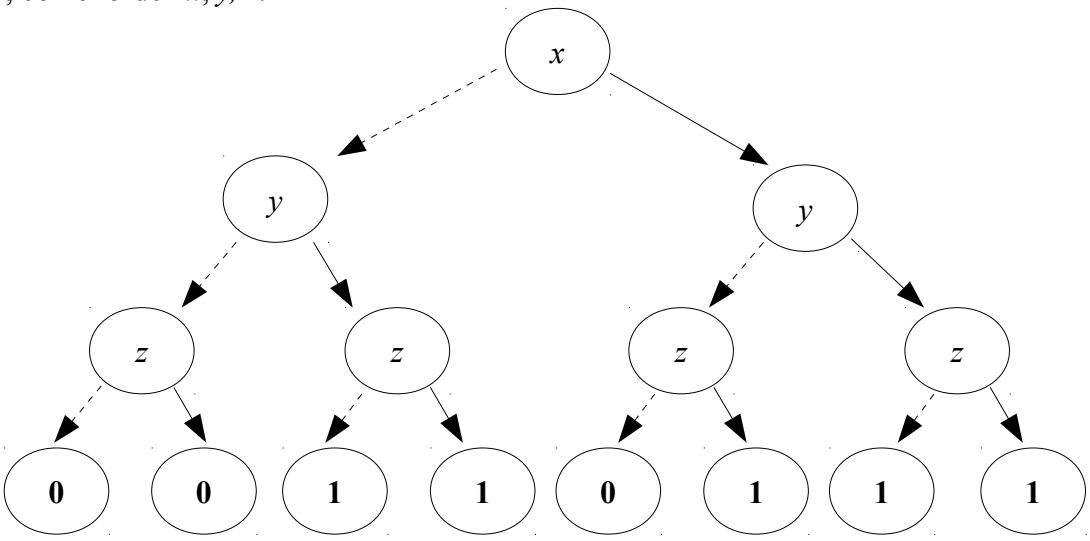


Fig. 4

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 digrafos acíclicos.

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.

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:

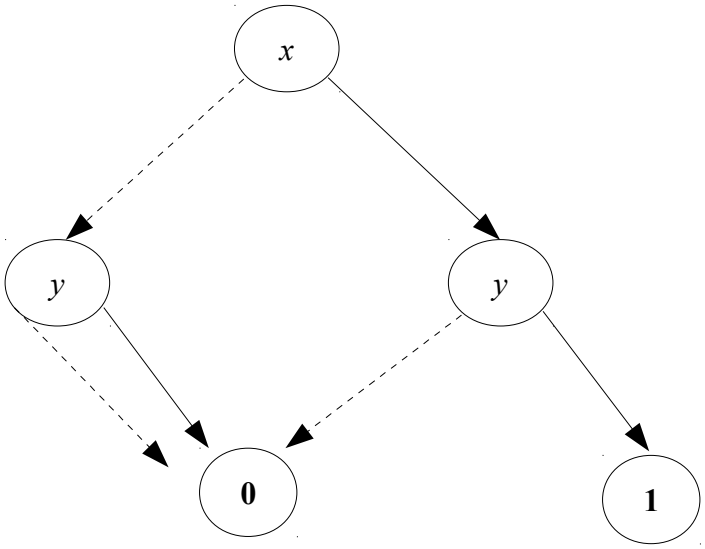


Fig. 5

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

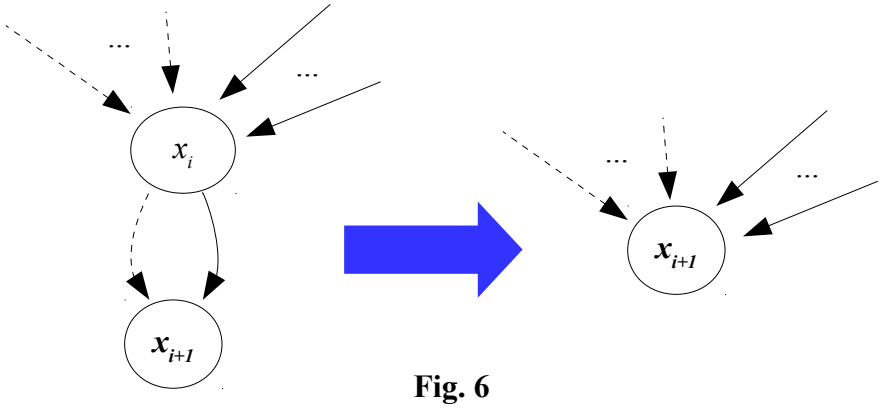


Fig. 6

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:

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.

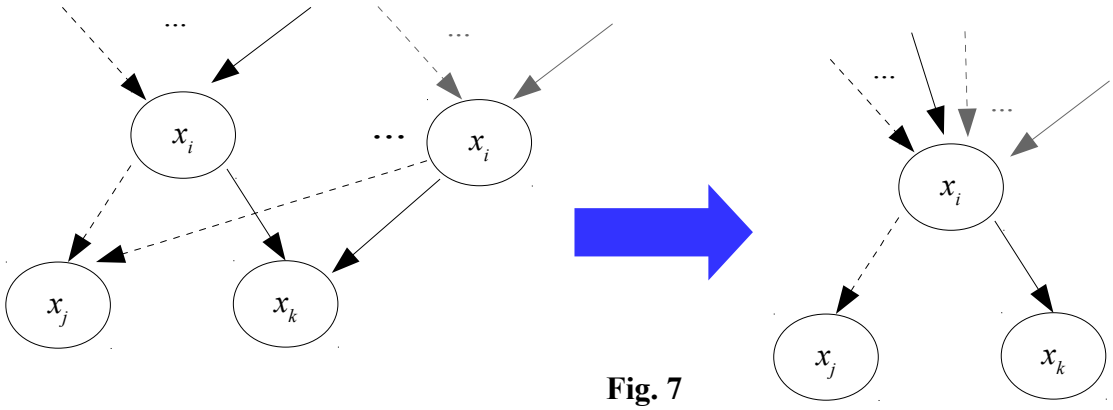


Fig. 7

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:

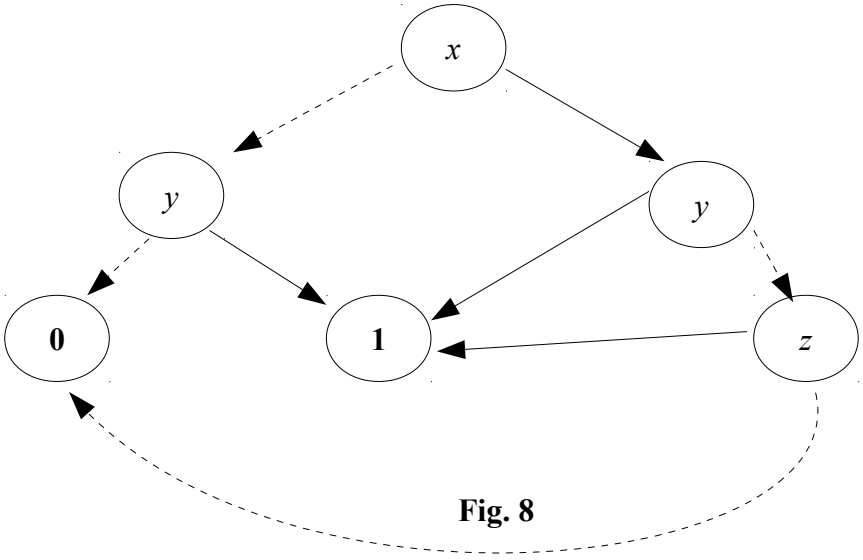


Fig. 8

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,

$$Para\ todo\ vértice\ v,\ v.low = u \leftrightarrow v.var < u.var^2$$

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

Sean *v* y *u* vértices del árbol.

- $v.low \mathrel{!}= v.high$

2 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*.

- *v.low* = *u.low* AND *v.high* = *u.high* AND *v.var* = *u.var* ↔ *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].

Apply

El algoritmo APPLY que permite ejecutar una operación binaria lógica entre dos árboles BDD fue detallada en [BRYANT1986].

Nuestro pseudocódigo es el siguiente:

```
# 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 $F = x+yz$

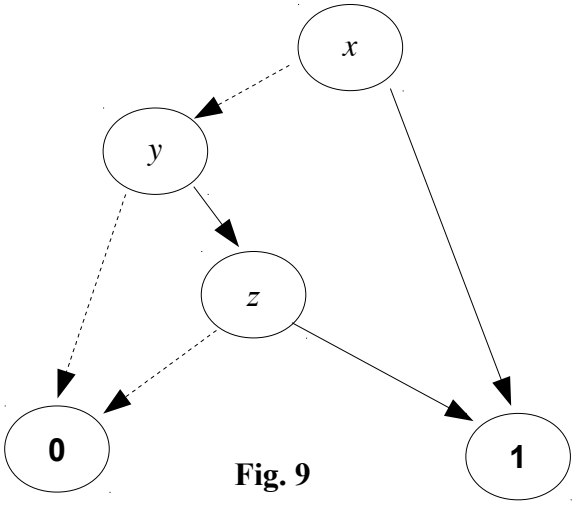


Fig. 9

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:

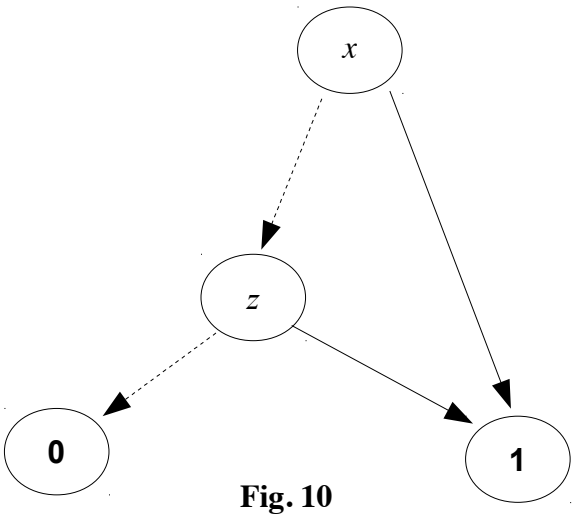


Fig. 10

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)
```

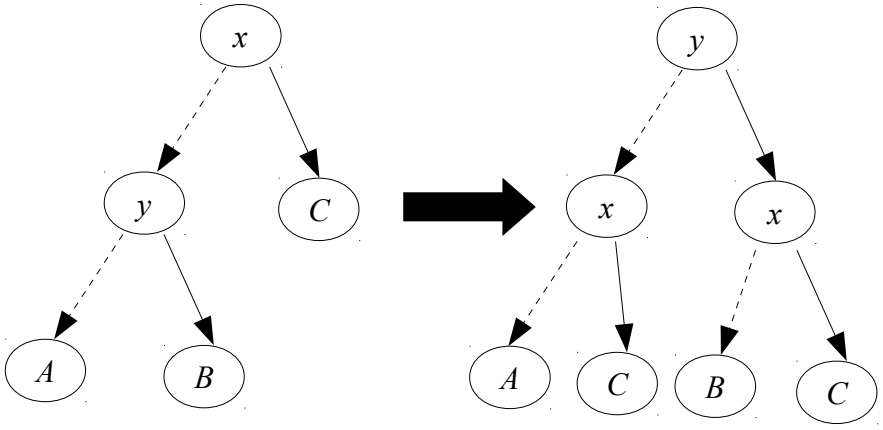
Swap

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.

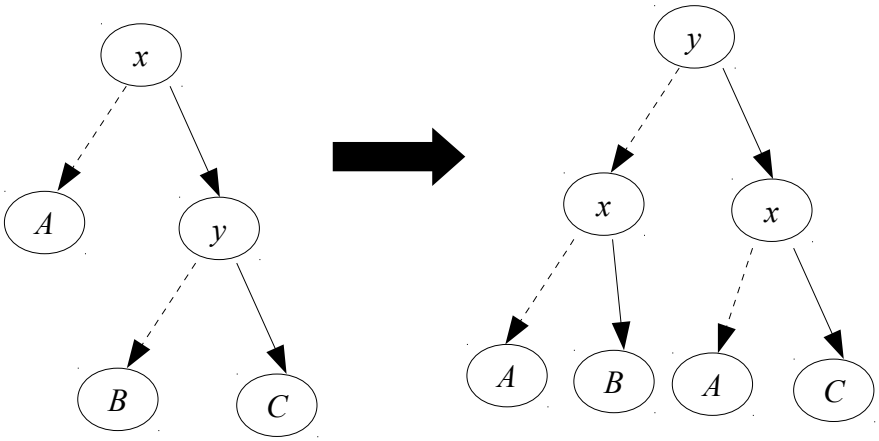
La implementación es ignorada en la mayoría de los trabajos de la bibliografía y en ellos se indica [Rud93] como referencia. Para comprender mejor esta operación se pueden ver las notas del mismos autor en el taller [Rud93W].

En [JacobiEtAl], los autores indican de forma gráfica cada uno de los posibles cambios escenarios en los que se puede encontrar un vértice que ha de intercambiarse con el del siguiente nivel. Los reproducimos a continuación:

Caso A



Caso B



Caso C

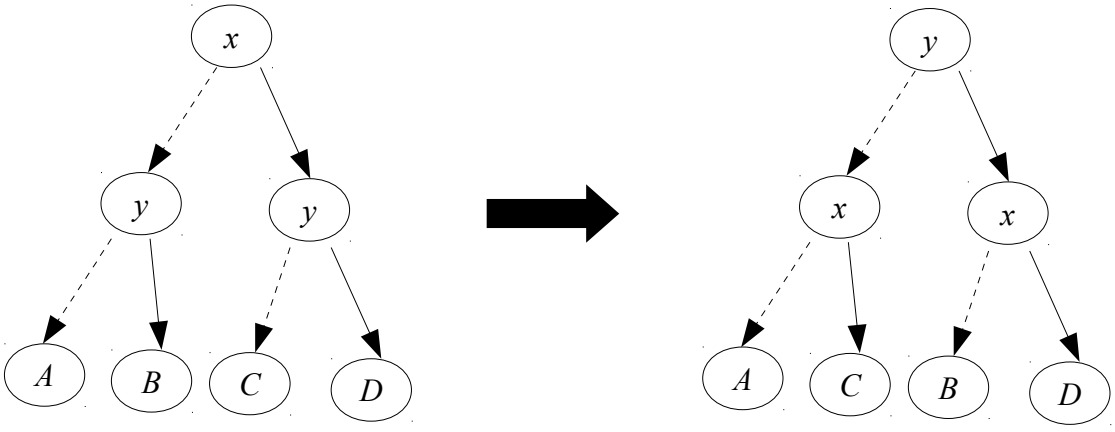


Fig. 11

```
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.

Sifting de Rudell

Richard Rudell en el año 1993 desarrolló un algoritmo de reducción del número de vértices en árboles BDD. 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 a tener en cuenta es el de cómo ir escogiendo las variables para ir realizando los intercambios. Nosotros hemos optado por ir escogiendo las variables

según su número de vértices, tal y como se recomendaba en [??].

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

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

Experimentos y resultados

Problemas abiertos

Bibliografía

[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.

[ImpBDDBrace] 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/

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