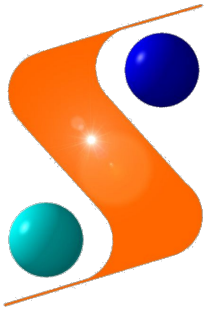
Desarrollo de un sistema dinámico de verificación de software basado en un comprobador de satisfacibilidad lógica



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

Ponente: Diego J. Romero López
Tutor: Elena Ruiz-Larrocha

Índice

1. Introducción

La complejidad del software llega a niveles en los que es imposible abarcar los estados del sistema.

Hay sistemas críticos que no deben fallar.

Si fallan, es una catástrofe.

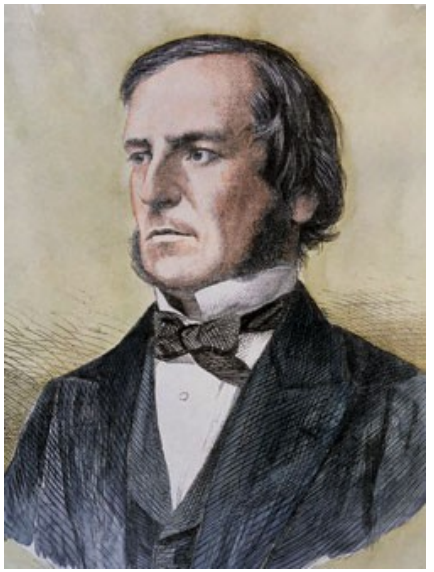
El coste de comprobar manualmente todos los estados de un sistema software es demasiado elevado.

¿Puede el sistema recuperarse si detecta un fallo?

Ejemplos de sistemas críticos:

- Aviónica.
- Transportes: vehículos, trenes...
- Sistemas médicos.
- Sistemas militares
- Banca
- Sistemas de posicionamiento geográfico

2. Lógica booleana



Definida por George Boole en el S. XIX.

Axiomatización del pensamiento filosófico.

Es la base de toda la lógica en computación.

Permite almacenar conocimiento con una expresión lógica.

Ej:

$$RESERVA_{estado = PENDIENTE} \wedge PAGA \Rightarrow RESERVA_{estado = PAGADO}$$

Proposición lógica

Una proposición lógica es:

- TRUE
- FALSE
- ¬ q, donde q es una proposición y ¬ el operador de negación.
- p OP q, con p y q proposiciones y OP un operador lógico.
- Operadores lógicos:
  - NOT
  - AND
  - OR
  - IMP

Leyes de la lógica proposicional

- Asociativas

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

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

- Distributivas

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

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

- Leyes de De Morgan.

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

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

Satisfacibilidad

Una expresión lógica es *satisfacible* cuando se puede obtener el valor cierto (TRUE) para al menos una asignación de valores de verdad a sus proposiciones.

Satisfacibles

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

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

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

No satisfacibles

$$(p \wedge not\ r) \wedge (not\ \ p \wedge r)$$

$$p \wedge (\neg p \wedge r)$$

Nuestro objetivo es comprobar la satisfacibilidad de forma eficiente.

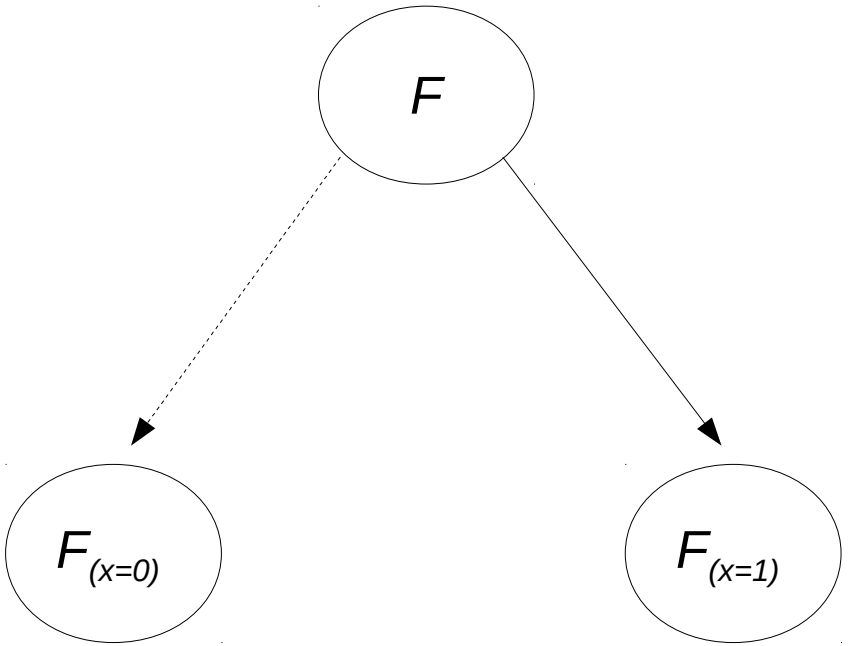
¿Cómo podemos hacerlo (de forma eficiente)?

- Analizador sintáctico.
- Procesamiento de texto.
- Resolutores SAT.
- Árboles Binarios de Decisión

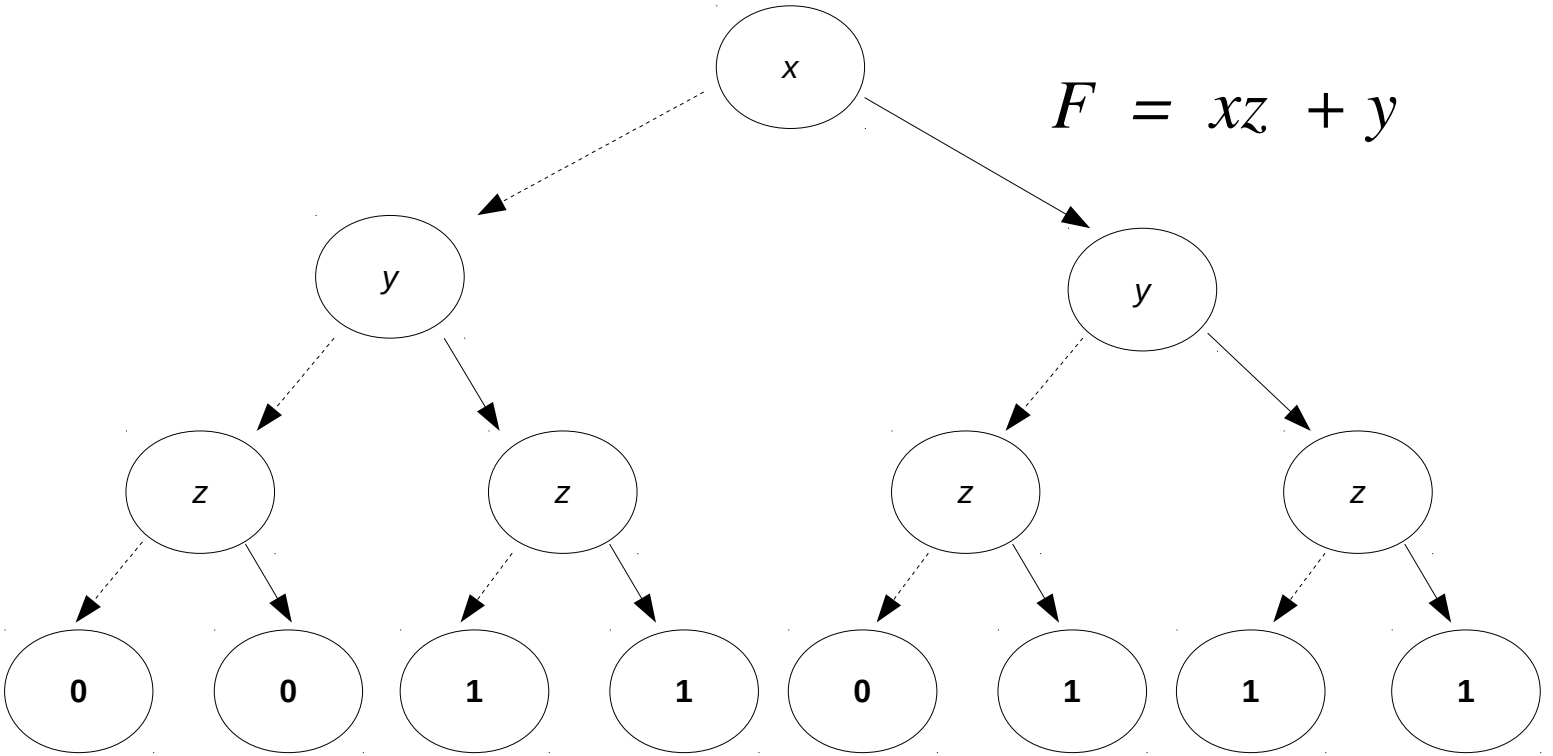
3. Árboles Binarios de Decisión

Estructura de datos basada en la expansión de Boole-Shannon:

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



Los Árboles Binarios de Decisión (BDD) son una forma de expresar una Tabla de Verdad:

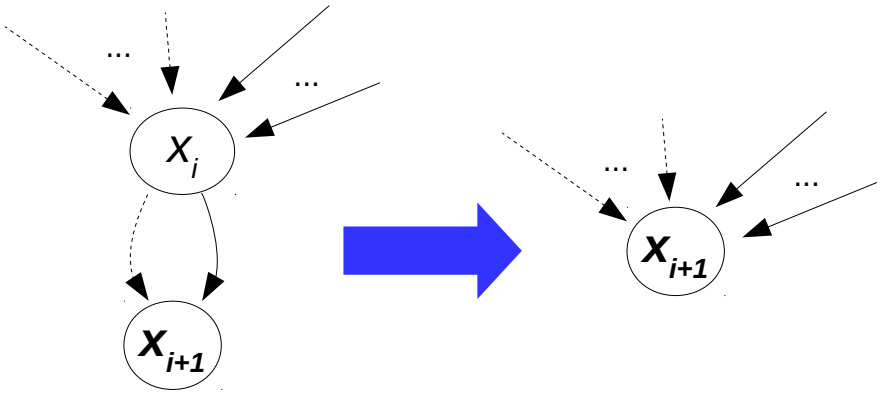


Problema: crecimiento de 2^n donde n es el número de variables.

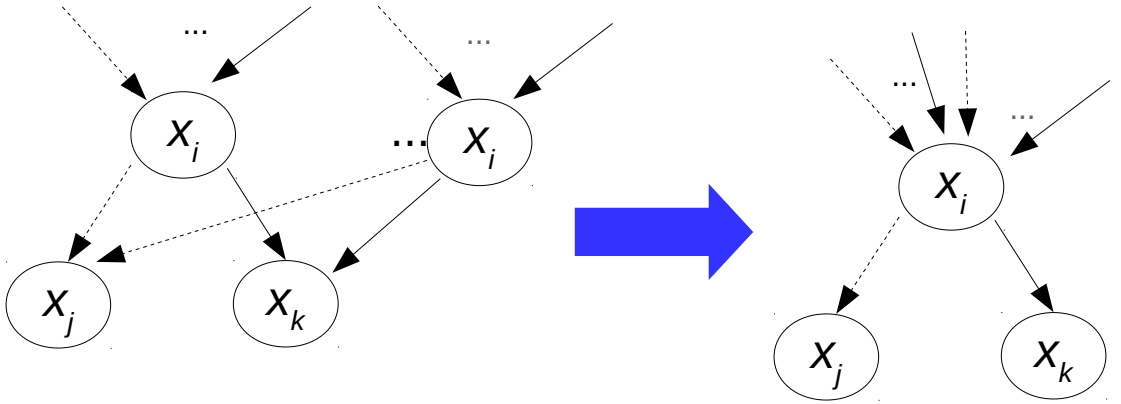
Solución: convertir el árbol en un grafo dirigido acíclico.

Randall E. Bryant define una serie de restricciones para construir el grafo a partir de un árbol binario de decisión:

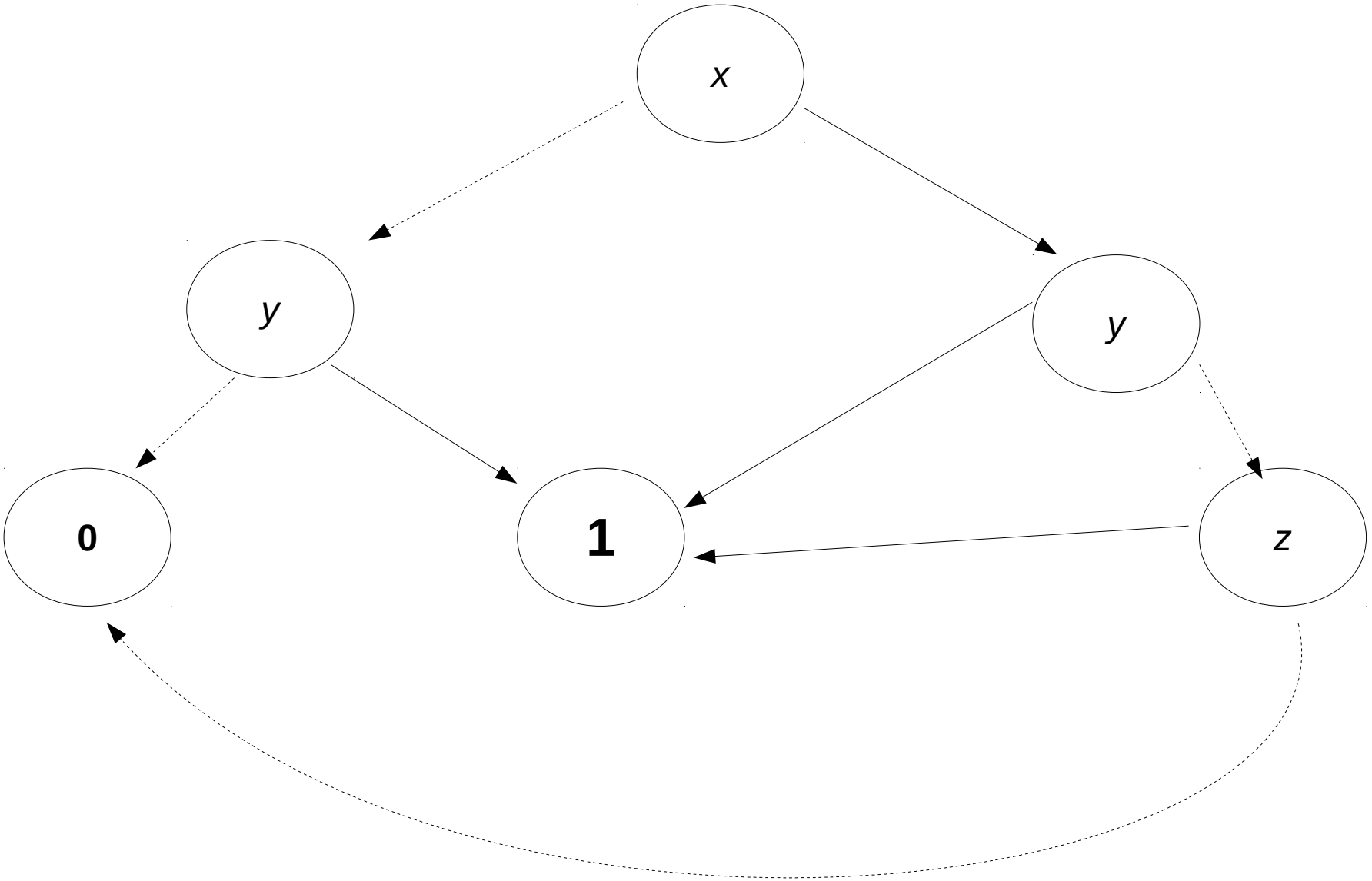
Unicidad:



No-redundancia:



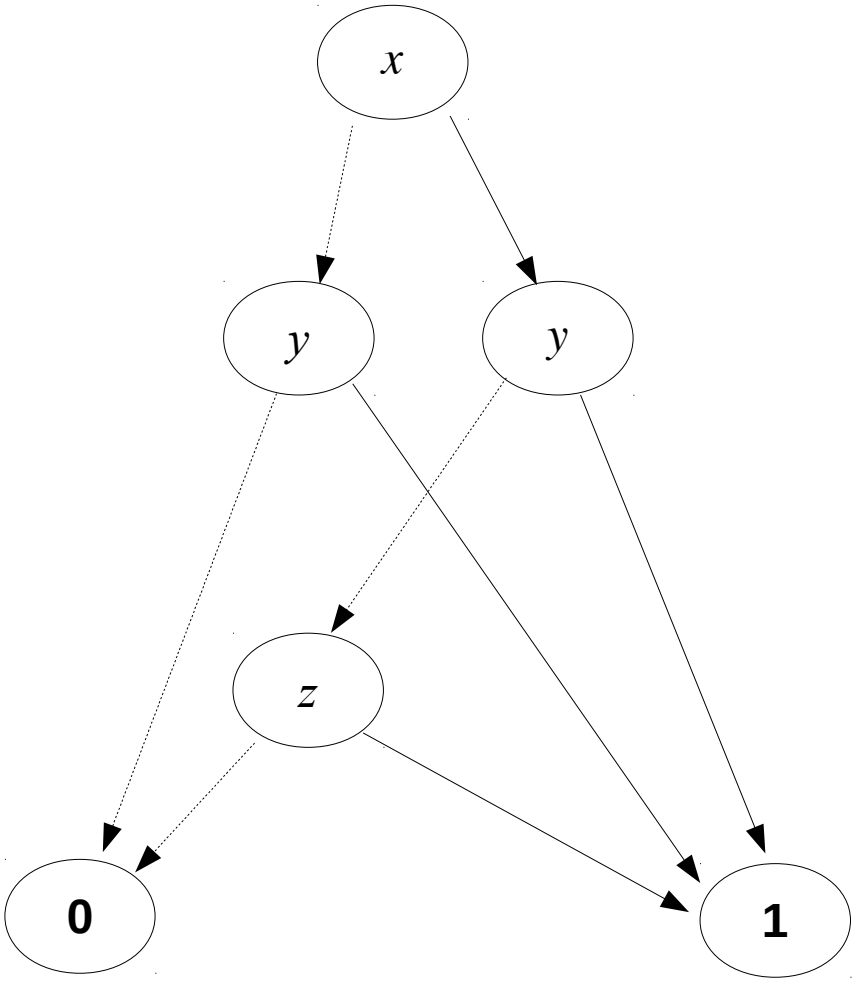
Aplicando esas restricciones a nuestro ejemplo:

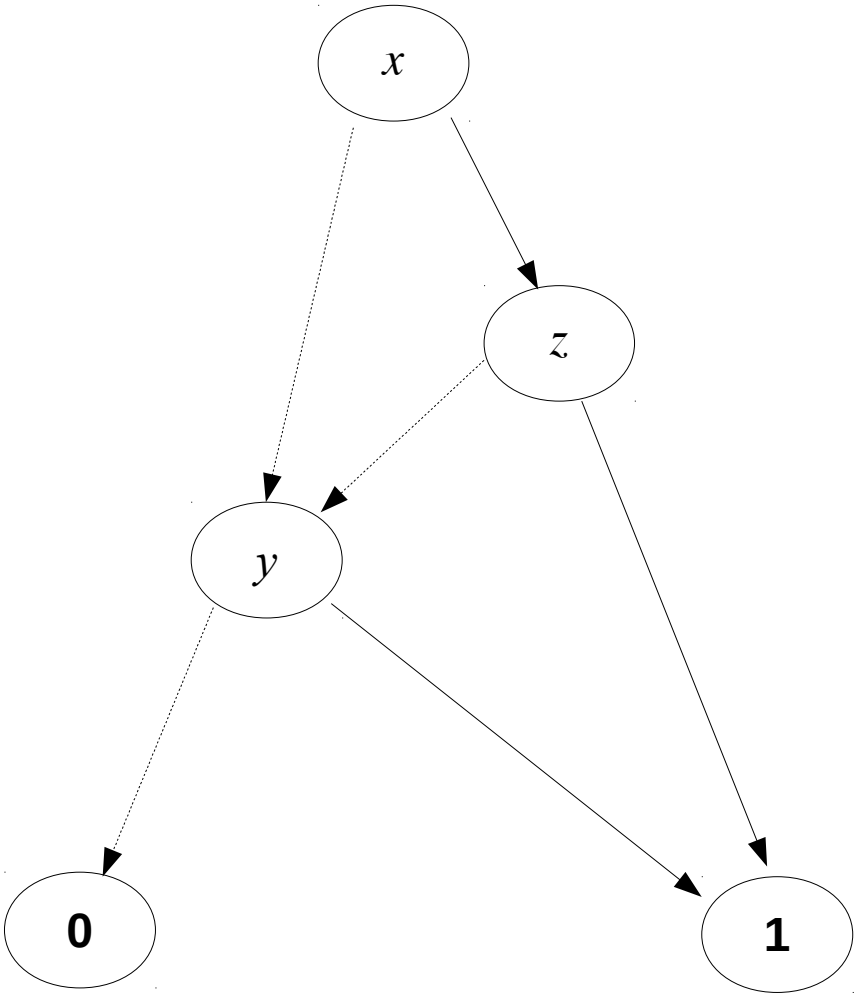


Propiedades

- Se pueden aplicar operaciones lógicas sobre árboles:
  - Binarias:
    - AND
    - OR
  - Unarias:
    - NOT
  - Restrict: evalúa la función lógica que contiene.
- Los BDDs proporcionan un mecanismo de **minimización** de expresiones lógicas.
- Los árboles son **únicos** para un orden de variables.
- La **implantación en hardware** de un BDD es *sencilla*.

El tamaño de los BDDs depende del orden de variables:





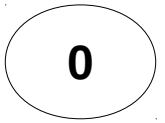
Encontrar el orden que hace el BDD óptimo en tamaño es un problema NP-duro.

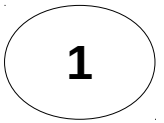
4. Verificación de software

Los BDDs pueden usarse como evaluadores de expresiones lógicas, de las restricciones del sistema.

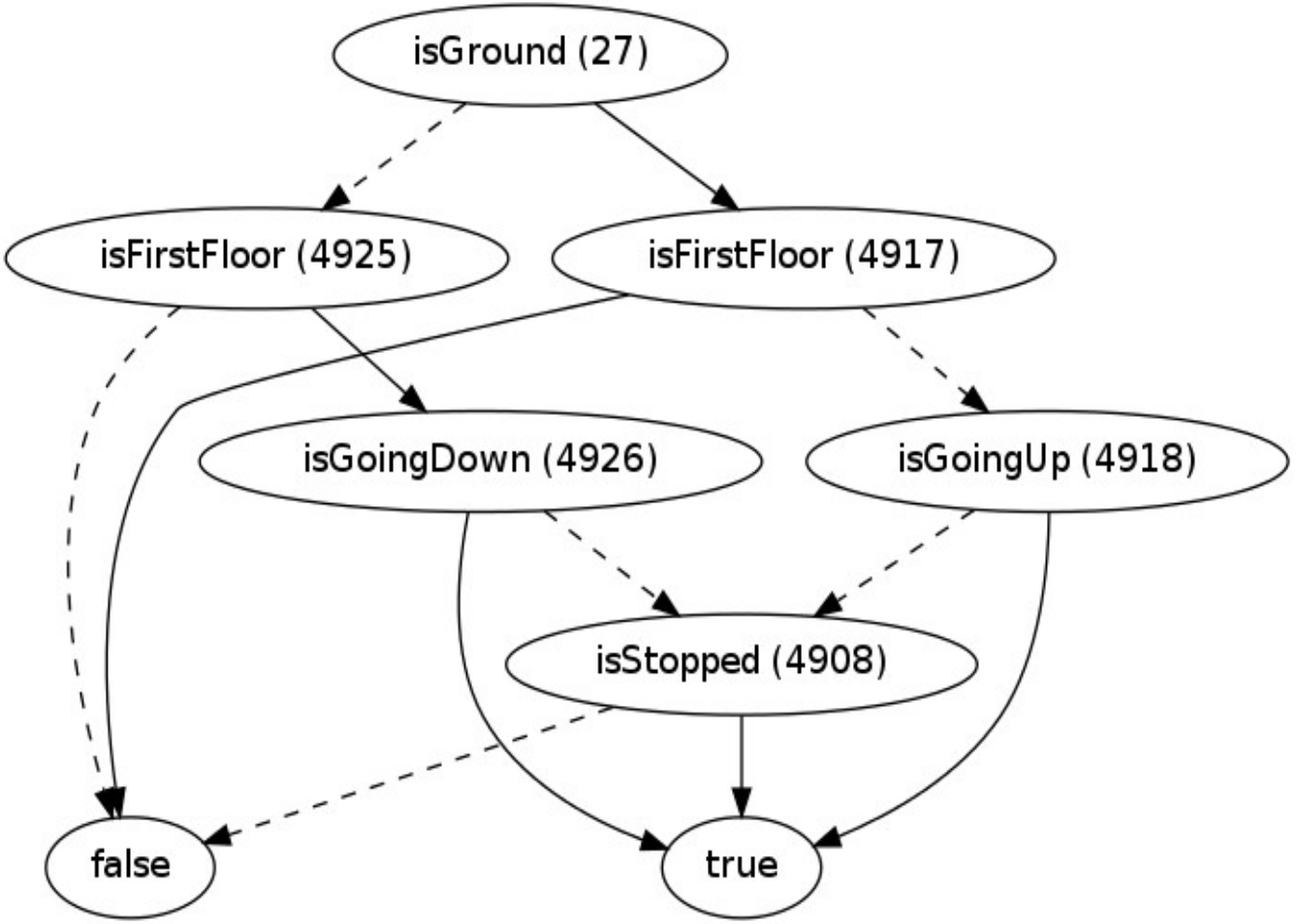
Operación **restrict**: asigna a una serie de variables unos valores de verdad y por tanto *restringe* el árbol.

Si se asignan valores de verdad a todas las variables, el BDD se comporta como un evaluador completo de la expresión, pudiendo devolver:

Árbol *false*: 

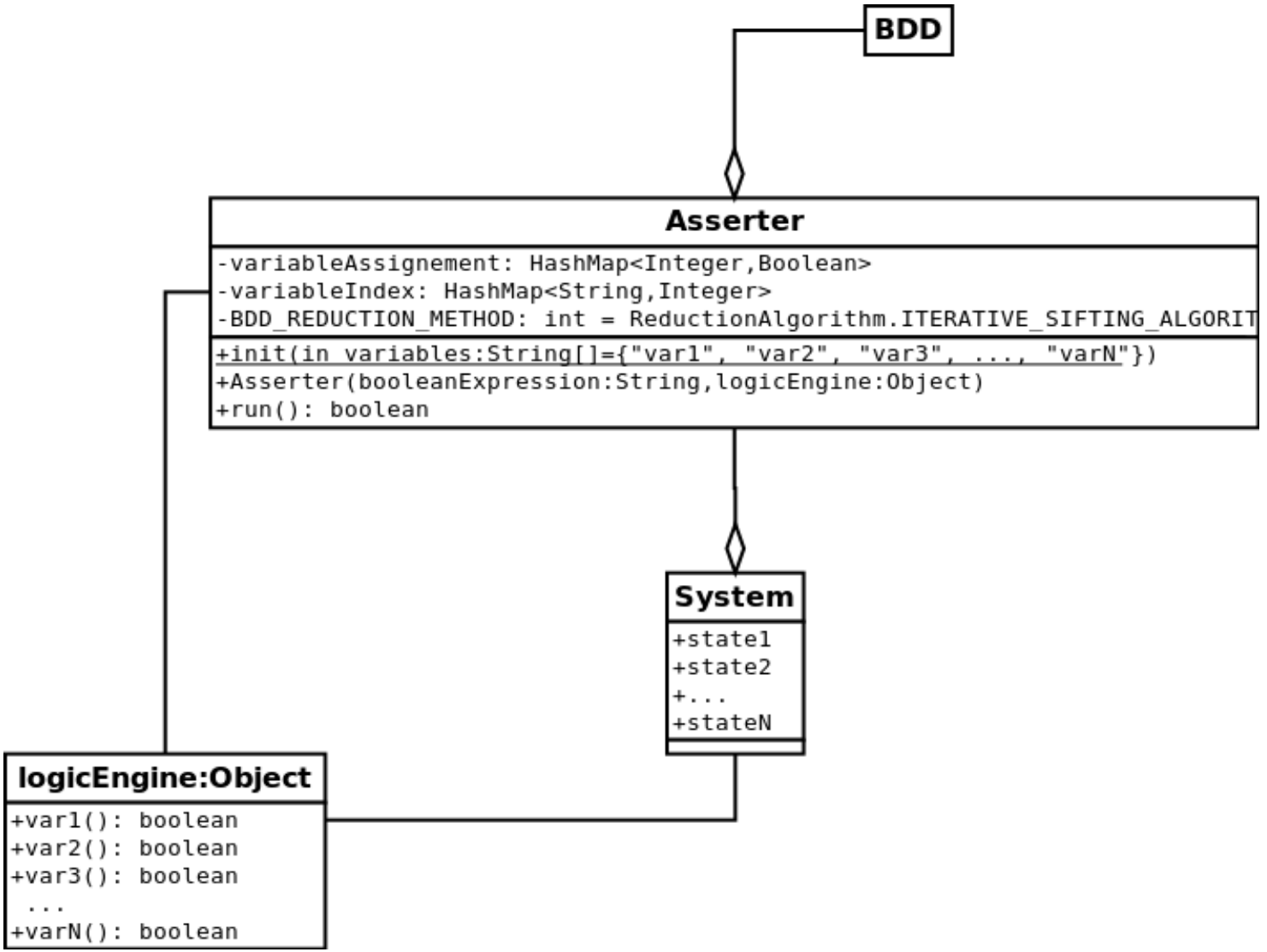
Árbol *true*: 

Un BDD puede mantener las reglas de un sistema:



Permitiendo evaluar si su estado es *consistente* o *inconsistente*.

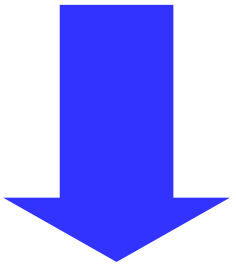
Arquitectura software del comprobador:



¿Cuántos proposiciones lógicas (variables) hacen falta para poder expresar las restricciones sobre un sistema software?

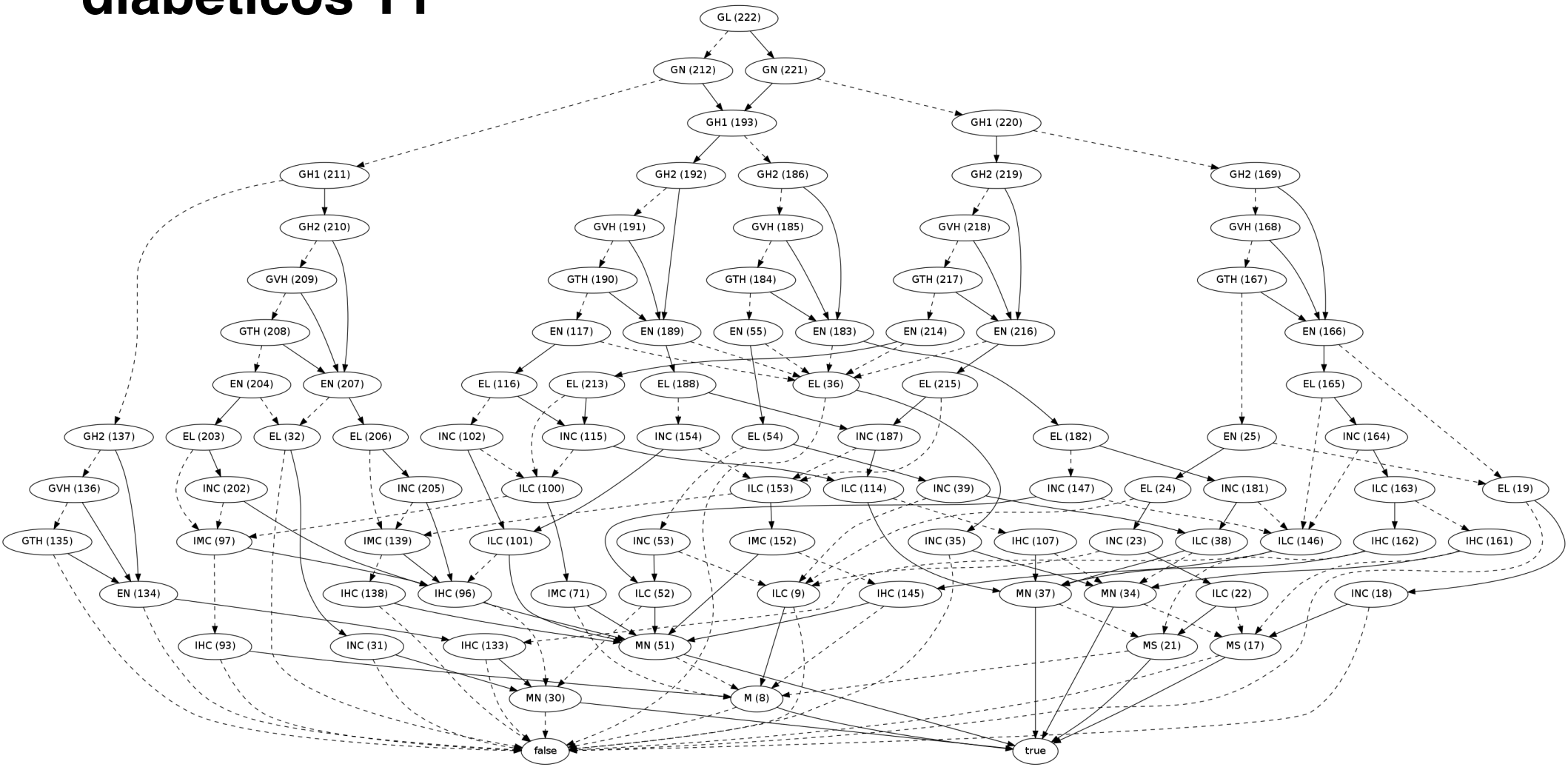
¿Cuál es la estructura de la fórmula?

¿Cuál es el mejor orden para construir el BDD?



¿Cuántos nodos harán falta para construir el BDD asociado?

Ejemplo: Asesor de control glucémico para diabéticos T1



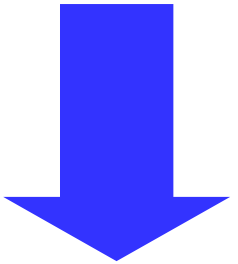
- 17 variables.
- 10 reglas.
- 94 vértices.

5. Reducción de Árboles Binarios de Decisión

El BDD depende fuertemente del orden de variables.

Podemos reordenar las variables del BDD hasta obtener un orden de variables *suficientemente pequeño* como para que sea *fácil* trabajar con él.

Nos centraremos en la reordenación dinámica.



Es más versátil y se puede usar una vez construido el BDD (o durante su construcción).

Hay diversas estrategias:

- Algoritmos de Búsqueda Local:
  - Sifting.
  - Widow Permutation

- Algoritmos Evolutivos
  - Algoritmos Genéticos.
  - Algoritmos Meméticos.

- Algoritmos Exactos:
  - Búsqueda Exhaustiva.

Mi solución (Iterative Sifting) **vence en la mayoría de las pruebas** realizadas debido a su forma de explotar la búsqueda de soluciones.

Iterative Sifting

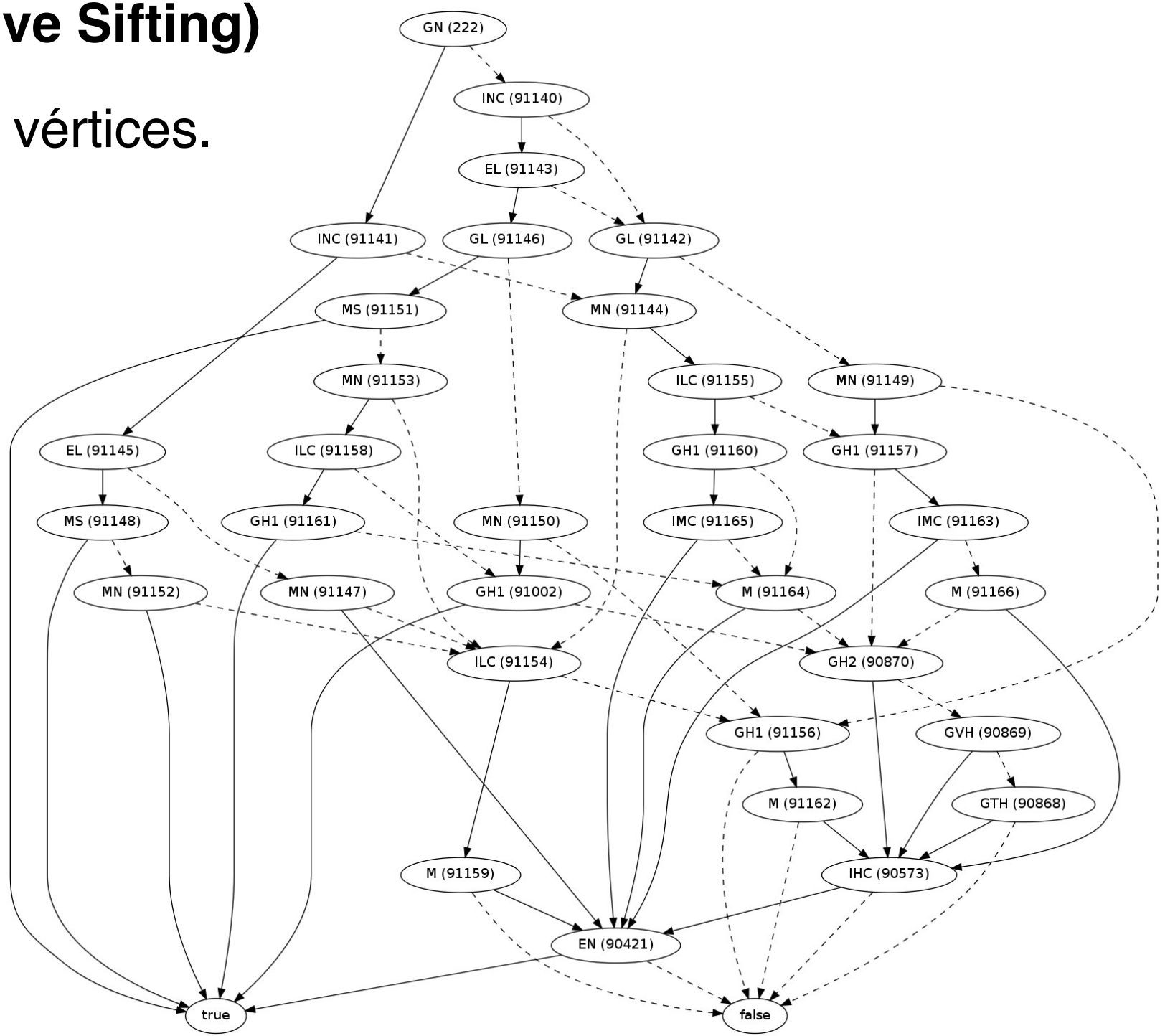
Algorithm 1 Iterative Sifting

Require: *numIterations*: Number of iterations.
Require: *BDD* A Binary Decision Diagram.
Ensure: A BDD with a new variable order that reduces its previous size.

```
1:  thereWasAnImprovement = False
2:  i = 0
3:  bestVariableOrder = BDD.variables()
4:  bestBDDSize = BDD.size()
5:  repeat
6:      thereWasAnImprovement = False
7:      i = 0
8:      while i < numIterations do
9:          # Order variables in descending order according to
10:         # the number of vertices of each one in the BDD.
11:         variableOrder = orderVariables(BDD.variables())
12:         for all var_j ∈ variableOrder do
13:             # Move the variable var_j to the best position
14:             # That is, the position that makes the BDD minimal.
15:             # Described in [7]
16:             moveVaribleBestPosition(var_j, BDD)
17:             bddSize = BDD.size()
18:             if bddSize < bestBDDSize then
19:                 bestVariableOrder = BDD.variables()
20:                 bestBDDSize = bddSize
21:                 thereWasAnImprovement = True
22:             end if
23:         end for
24:         i = i + 1
25:     end while
26: until not thereWasAnImprovement
27: applyOrderToBDD(BDD, bestVariableOrder)
28: return BDD
```

Ejemplo: Asesor de Diabéticos T1 (reducido usando Iterative Sifting)

- 36 vértices.



6. Problemas abiertos

- ¿Hasta qué punto es necesaria la paralelización de estos árboles?
- ¿Es la descomposición de fórmulas lógicas de Boole-Shannon la mejor?
- ¿Se puede desarrollar un Algoritmo Genético que no desborde la memoria?

7. Conclusiones

- DJBDD es una biblioteca nueva para trabajar con Árboles Binarios de Decisión.
- He desarrollado una herramienta para el uso de BDDs como comprobadores de consistencia.
- Hay dos ejemplos de uso de comprobación de consistencia, uno de ellos real.
- He desarrollado un nuevo algoritmo de reducción de BDDs: Iterative Sifting, cuya publicación está pendiente.