Documentación del proyecto de resolución de Tableaux en lógica proposicional

Pablo Martínez Sánchez

4 de octubre de 2017

Índice

1. Introducción

La finalidad de este documento es explicar de forma concisa cómo funciona la parte del proyecto encargada de resolver el *tableaux* a partir de una fórmula dada. Todo el código, y más información de interés(como el manual de usuario) se encuentra en github.El proyecto está implementado en C, y a gran escala, su funcionamiento es el siguiente:

1. Mediante las herramientas *Bison* y *Flex* , se hace un análisis de la entrada y, si es correcta, se crea la fórmula(como una estructura de datos)
2. Se resuelve el *tableaux* a partir de dicha fórmula
3. Se imprime el resultado por la salida estándar y en un fichero *SVG*

Dicho esto, este documento se centra en el funcionamiento del segundo punto, cuyo código fuente está en los ficheros *Formula.c* y *Formula.h*

2. Estructuras de datos

Las estructuras de datos se organizan de forma jerárquica, de manera que la más alta en la jerarquía contiene estructuras del siguiente nivel, y así sucesivamente; Se detallan todas ellas, empezando por la más alta.

2.1. Tableaux

```
struct TableauxRep {
  Formula f;
  Tableaux ti;
  Tableaux td;
  int etiqueta;
};
```

El *Tableaux* se genera en forma de árbol, donde, inicialmente, sólo existe la raíz del árbol(que es creada de forma automática una vez se ha leído la entrada). La estructura de datos que representa dicho árbol es *TableauxRep*, que contiene:

- Formula f: Se trata de la fórmula de dicho nivel en el árbol(en la raíz, es la fórmula que ha introducido el usuario.
- Tableaux ti: Es un puntero al *tableaux* izquierdo.
- Tableaux td: Es un puntero al *tableaux* derecho.
- int etiqueta: Define la etiquetada de dicho nivel del árbol. Un nodo cualquiera en nivel determinado nodo puede estar abierto o cerrado(en *Formula.h* se encuentran los *define* necesarios) Este campo solo tiene una finalidad visual, pues se utiliza a la hora de dibujar el árbol, una vez resuelto. Realmente solo se usa la etiqueta cuando es un nodo hoja.

Es importante destacar los casos que pueden darse de cara a un nodo cualquiera del árbol, y como dicha estructura respondería:

- Un nodo hoja: Tanto *ti* como *td* son nulos.
- Un nodo con un hijo(alfa fórmula): Siempre sucede que *ti* es no nulo y *td* es nulo(ti apunta a la siguiente fórmula)
- Un nodo con dos hijos(beta fórmula): Ambos son no nulos.

2.2. Formula

```
struct FormulaRep {
  Atomo a1;
  Atomo a2;
  Formula f1;
  Formula f2;
  int COD_OP;
  int not;
  struct FormulaRep *sig;
};
```

Se define una fórmula como uno o dos fórmulas, relacionadas con un operador(solo si hay dos). Si una fórmula en realidad solo contiene un átomo, entonces no se almacena como fórmula, sino como átomo directamente. Por ejemplo;

$$a \vee b$$

Consta de un átomo, un operador y otro átomo. En este caso, *COD_OP* sería donde se almacena el operador, que es un *or*. Tanto *a1* como *a2* serían no nulos(contendrían a *a* y *b*,respectivamente), mientras que tanto *f1* como *f2*, serían nulos. No obstante, es posible encontrar algo como;

$$a \vee (b \wedge c)$$

En cuyo caso, *a1* contendría el átomo *a*, pero *a2* no podría contener un átomo, sino que $(b \wedge c)$ se almacena en una fórmula(en este caso, *f2*). Entonces, *a1* contendría al átomo *a*, *COD_OP* sería *or* y *f2* sería un puntero a una fórmula, en el que a su vez; *a1* sería *b*, *COD_OP* sería *and* y *a2* sería *c*. A continuación se exponen otros casos:

$$(a \implies d) \vee (b \wedge c)$$

- *COD_OP* sería *or*
- *f1* y *f2* apuntarían a dos fórmulas;
- En *f1*, *COD_OP* sería la *implicación*, su *a1* sería *a* y su *a2*, *d*
- En *f2*, *COD_OP* sería la *and*, su *a1* sería *b* y su *a2*, *c*

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

- *f1* sería nulo(*a1* contendría el átomo *p*)mientras que *f2* apuntaría a una formula:
- *a2* contendría el átomo *q* mientras que *f1* apuntaría a la siguiente fórmula:
- En *a1* se guarda *d*, en *a2*, *r* y en *COD_OP*, *and*.

No obstante, también es posible negar una fórmula completa, como por ejemplo:

$$\neg(a \wedge r)$$

La única diferencia respecto a todo lo anterior es que en este caso, se estaría utilizando el entero *not*(que estaría como *NEGADO*, tal y como está especificado en los *defines* de *Formula.h*). Por último queda comentar como se representaría una lista de fórmulas, por ejemplo:

$$a \wedge b, r, p \vee m$$

Si bien es cierto que lo anterior es equivalente a sustituir las comas por *and*s lógicos, este proyecto soporta la representación mediante comas. Para ello, se utiliza el último campo que queda por comentar, el de *sig*. De esta forma, la fórmula contiene información para $a \wedge b$, y un puntero a la siguiente fórmula, que sería r, que tendría un puntero a la siguiente fórmula, $p \vee m$. A modo de resumen;

- Formula f: Se trata de la fórmula de dicho nivel en el árbol(en la raíz, es la fórmula que ha introducido el usuario.
- Atomo a1: Representa el átomo izquierdo(si existe)
- Atomo a2: Representa el átomo derecho(si existe)
- Formula f1: Representa un puntero a la fórmula izquierda(si existe)
- Formula f2: Representa un puntero a la fórmula derecha(si existe)
- int COD_OP: Código del operador(si existe)
- int not: Representa si la fórmula global va o no negada
- struct FormulaRep *sig: Siguiente fórmula(si existe)

2.3. Átomo

Se trata de la última estructura de datos.

```
struct AtomoRep {
  char id;
  int not;
};
```

- char id: Almacena el carácter que define a un átomo(esto está restringiendo la longitud de caracteres de los átomos a uno)
- int not: Representa la negación o no de dicho átomo, permitiendo negar cualquier átomo por separado)

3. Procedimientos y funciones

3.1. Funcionamiento general

En esta sección se explica detalladamente cuál es el proceso de transformación del nodo hoja del *tableaux* , hasta tener el *tableaux* completamente desarrollado.
Es importante recordar que todo lo mencionado se encuentra en *Formula.c* y *Formula.h*.
Cuando *Flex* y *Bison* terminan el análisis, pasan una fórmula a la función

```
void ResolverTableaux(Formula oracion, ...)
```

Que a su vez, crea el *tableaux* a partir de la fórmula(que consiste tan solo del nodo raíz) y llama a

```
void Resolver(Tableaux t)
```

El objetivo de dicha función es resolver el *tableaux* t. Esto significa dos cosas; Por un lado, la función es recursiva, y se llamará para los *tableaux* izquierdo y/o derecho. Por otra parte, es la función que lleva a cabo todo el trabajo. En oración

```
Formula oracion = t->f;
```

guardamos el puntero a la primera fórmula de la lista. De esta forma, podemos avanzar por la lista con este puntero(con el que recorremos la lista), pero también conservamos la dirección de la primera fórmula de la lista. Lo primero que se hace es buscar, en la lista de fórmulas, una que sea alfa fórmula(puesto que es preferible resolver estas primero, ya que dará lugar a árboles de menor tamaño):

```
while(oracion->sig != NULL && (SoloTieneUnAtomo(oracion) ||
    !EsAlfaFormula(oracion)))
```

El bucle continua mientras haya un siguiente y la fórmula actual solo tenga un átomo o mientras dicha fórmula no sea alfa fórmula. Es decir, si una fórmula solo tiene un átomo, no nos interesa intentar resolverla(porque no hay nada que resolver) Si, una vez terminado el bucle, la fórmula donde nos hemos quedado no es alfa fórmula

```
if(!EsAlfaFormula(oracion))
```

Entonces se reinicia la búsqueda, esta vez, buscando una beta fórmula. Tanto si era alfa o no, se comprueba

```
if(oracion->sig == NULL && SoloTieneUnAtomo(oracion))
```

Si esto fuera cierto, quiere decir que nos encontramos en un nodo hoja porque todas las fórmulas en la lista son átomos, en cuyo caso, comprobamos si hay una contradicción en el nodo.

```
if (ContieneContradiccion(t->f))t->etiqueta = CERRADO;
    else t->etiqueta = ABIERTO;
```

Esto se comprueba solo para una mejor presentación una vez se muestre el resultado. Si, por el contrario, la comprobación fuera falsa $(if(oracion- > sig == NULL...))$ querría decir que el puntero oración esta apuntando a una fórmula(dentro de la lista) que sí es posible resolver. De hecho, es la primera(por la izquierda) que hay que resolver. Antes de nada, es necesario comprobar si existe una contradicción en la fórmula o no. Si lo hubiera, sería inútil intentar resolver dicho nodo, porque el resultado de esto sería siempre el mismo(todo insatisfacible). De esta forma, solo continuaría si no hubiera contradicción, en cuyo caso, se analiza el operador(puesto que no se resuelven fórmulas con un sólo átomo, siempre hay un operador). Dependiendo del operador, se llamará a una función u otra

```
switch(oracion->COD_OP) {
      case COD_DIMP:
        if(oracion->not == NEGADO) dobleImpNegado(t,
   busqueda);
        else dobleImp(t,busqueda);
        break;

      case COD_AND:
        if(oracion->not == NEGADO) andNegado(t,busqueda);
        else and(t,busqueda);
        break;

      case COD_IMP:
        if(oracion->not == NEGADO) impNegado(t,busqueda);
        else imp(t,busqueda);
        break;

      case COD_OR:
        if(oracion->not == NEGADO) orNegado(t,busqueda);
        else or(t,busqueda);
        break;
    }
```

Como se aprecia, una vez sabemos el operador, debemos consultar si la fórmula entre paréntesis va negada(esto viene dado por el campo *not*). En función de esta información, se llama a la función correspondiente. Puesto que la filosofía de todas las funciones es

igual, vamos a ver una en concreto para así comprender el funcionamiento de todas ellas. En el caso del *and*:

```
void and(Tableaux t,int busqueda) {
  Formula aux;
  Formula izquierda;
  Formula derecha;

  t->ti = CrearTableaux(CopiarFormula(t->f));
  if(busqueda == 0) {
    izquierda = ExtraerIzquierda(t->ti->f);
    derecha = ExtraerDerecha(t->ti->f);
    derecha->sig = t->ti->f->sig;
    izquierda->sig = derecha;
    LiberarFormula(t->ti->f);
    t->ti->f = izquierda;
  }
  else {
    aux = BuscarOracion(busqueda,aux,t->ti);
    izquierda = ExtraerIzquierda(aux->sig);
    derecha = ExtraerDerecha(aux->sig);
    derecha->sig = aux->sig->sig;
    izquierda->sig = derecha;
    LiberarFormula(aux->sig);
    aux->sig = izquierda;
  }
  Resolver(t->ti);
}
```

Por parámetros recibimos *t*, que es el puntero a la primera fórmula del *tableaux* padre. Como el *and* es una alfa fórmula, solo interesa que el padre tenga un hijo, y es por esto que a $t->ti$ se le asigna una copia del padre. De esta forma, es posible operar en $t->ti$ y hacer las modificaciones que sean necesarias de forma independiente al padre. La variable búsqueda hace referencia a la enésima fórmula(suponiendo que se empieza en la primera, en la del parámetro *t*), esto es, el número de veces que hay que avanzar en la lista para llegar hasta la fórmula concreta donde esta el *and*, y por lo tanto, donde hay que operar. La única diferencia entre si es 0 o no, es que si fuera 0, el puntero a modificar es $t->ti$, mientras que si no lo es, sería una fórmula que se encontraría a partir de dicha lista.
En el *and*, lo único que hay que hacer es romperlo y, donde antes había una fórmula, ahora hay dos:

$$a \wedge (b \vee s)$$

pasa a ser

$$a, b \vee s$$

Esto se traduce en; extraer lo que haya a la izquierda del operador:

```
izquierda = ExtraerIzquierda(aux->sig);
```

y lo mismo con la derecha. Entonces, se conecta la fórmula derecha con la fórmula original, la izquierda con la derecha y por último, se elimina la fórmula original. Una vez hecho esto, el nodo actual(el izquierdo del padre) ya está resuelto, por lo que lo único que queda es resolver dicho nodo, para lo cual se vuelve a llamar a

```
void Resolver(Tableaux t)
```

pasándole el nodo actual. De esta forma, se resuelve todo el *tableaux* de forma recursiva.
Aunque no se comente detalladamente el funcionamiento de todas las funciones, a continuación se citan todas ellas con una breve descripción.

3.2. Funciones concretas

- `Formula CrearFormula(Atomo a)`

  Crea una fórmula a partir del átomo *a*

- `Atomo CrearAtomo(char _id,int _not)`

  Crea un átomo a partir de los parámetros

- `Atomo CopiarAtomo(Atomo original)`

  Devuelve una copia del átomo *original*

- `Atomo ExtraerAtomo(Formula f)`

  Devuelve el único átomo que contiene(si tiene mas de uno, o *f* tiene fórmulas anidadas, devuelve NULL)

- `int SoloTieneUnAtomo(Formula f)`

  Devuelve *TRUE* si la fórmula *f* sólo tiene un átomo, *FALSE* en caso contrario

- `int EsAlfaFormula(Formula f)`

  Devuelve *TRUE* si la fórmula *f* es una alfa fórmula, *FALSE* en caso contrario

- `Formula NegarFormula(Formula f)`

  Devuelve la fórmula *f*, negada

- `Formula CopiarFormula(Formula original)`

  Devuelve una copia de la fórmula *f*

- `Formula Unir(Formula formula1,int operador,Formula formula2)`

Devuelve la unión de la fórmula *formula1* con la fórmula *formula2* mediante el operador *operador*

- ```
  Formula Concatenar(Formula f1,Formula f2)
  ```

  Devuelve la concatenación de la fórmula *f1* con la fórmula *f2*. Es decir, pone *f2* a continuación de *f1*

- ```
  Formula ExtraerIzquierda(Formula f)
  ```

  Devuelve la formula o atomo que haya en la izquierda de *f*

- ```
  Formula ExtraerDerecha(Formula f)
  ```

  Devuelve la formula o atomo que haya en la derecha de *f*

- ```
  Formula BuscarOracion(int busqueda, Formula aux,
     Tableaux t)
  ```

  Busca la oración *aux* en el *tableaux t* que esta a *busqueda* iteraciones de distancia

- ```
  int ContieneSuNegado(Atomo a, Formula f)
  ```

  Devuelve *TRUE* si en *f* existe el negado de *a*, *FALSE* en caso contrario

- ```
  int ContieneContradiccion(Formula f)
  ```

  Devuelve *TRUE* si existe un átomo y su negado en la fórmula *f*, *FALSE* en caso contrario

- ```
  void and(Tableaux t,int busqueda);
  void orNegado(Tableaux t,int busqueda);
  void impNegado(Tableaux t,int busqueda);
  void dobleImp(Tableaux t,int busqueda);
  ```

  Conjunto de funciones para resolver alfa fórmulas. Reciben el *tableaux t* y, donde indique *búsqueda* aplican las modificaciones necesarias para resolver *t*(suponiendo que ahí el operador que hay es el correcto),copiando la fórmula del padre y volviendo a llamar a *Resolver*

- ```
  void andNegado(Tableaux t,int busqueda);
  void or(Tableaux t,int busqueda);
  void imp(Tableaux t,int busqueda);
  void dobleImpNegado(Tableaux t,int busqueda);
  ```

  Igual que las funciones mencionadas anteriormente(excepto que estas son para beta fórmulas)

- ```
  void Resolver(Tableaux t)
  ```

  Resuelve el *tableaux t*
  ```

- `Formula ExtraerDerecha(Formula f)`

  Devuelve la formula o átomo que haya en la derecha de *f*

  NOTA: Se han obviado las funciones cuya finalidad no es la de resolver o ayudar a resolver el *tableaux* (como las funciones que están involucradas en mostrar la solución)

4. Consideraciones finales

Una vez el *tableaux* se ha resuelto, en la función "*main*"de *Formula.c* que es

```
void ResolverTableaux(Formula oracion, ...)
```

se tiene el puntero a la raíz del árbol(dicha función es la que es llamada al final del análisis de la cadena de entrada) Con dicho puntero se llama a funciones de otros módulos que imprimen la solución por la salida estándar y otra de ellas, que hará lo mismo en un fichero *SVG* que puede abrirse desde cualquier navegador o visor de imágenes. Dicha visualización es mucho más versátil pues es posible hacer zoom en aquellas zonas de interés. Para imprimir el resultado, la idea que hay detrás es pasar las estructuras de datos definidas en este documento a otras que sirvan para ser impresas, haciendo uso de funciones de la familia *show_ascii* como

```
char* show_ascii(char* buf, Formula f)
```

```
char* show_svg(char* buf, Formula f)
```

que se encargan de pasar de las estructuras de datos a una cadena, lista para ser mostrada al usuario. Como se ha mencionado antes, todo el código está en github, junto con instrucciones para compilarse y utilizar dicho software