

# Índice

<b>Aclaraciones Generales</b>	<b>2</b>
<b>Introducción</b>	<b>3</b>
<b>Situaciones de la vida real que se pueden modelar utilizando MAX-SAT</b>	<b>3</b>
<b>Algoritmo exacto para MAX-SAT</b>	<b>3</b>
Algoritmo de fuerza bruta . . . . .	3
Algoritmo de backtracking . . . . .	4
Complejidad de algoritmos exactos . . . . .	5
<b>Heurística constructiva para MAX-SAT</b>	<b>7</b>
Aproximación del resultado . . . . .	8
Complejidad de la heurística constructiva . . . . .	8
<b>Heurística de búsqueda local para MAX-SAT</b>	<b>8</b>
Detalles de implementación . . . . .	9
Complejidad de la heurística de búsqueda local . . . . .	10
<b>Metaheurística de búsqueda tabú para MAX-SAT</b>	<b>10</b>
<b>Bibliografía</b>	<b>11</b>

## Aclaraciones Generales

- La implementación de todos los algoritmos se realizó en lenguaje C++.
- Para calcular los tiempos de ejecución de los algoritmos se utilizó la función `gettimeofday()`, que se encuentra en la librería `<sys/time.h>`. Dado que dicha función funciona solamente en sistemas operativos de tipo linux, se debe compilar con el flag `-DTIEMPOS` en este tipo de sistemas para poder hacer uso de las mismas.
- Para la realización de los gráficos se utilizó Qtiplot

## Introducción

En el presente trabajo se buscó realizar diferentes aproximaciones a la resolución del problema MAX-SAT. El problema MAX-SAT es un problema de optimización proveniente del problema de decisión SAT.

El problema SAT se basa en decidir si un conjunto de cláusulas en forma normal conjuntiva, tiene alguna asignación de las variables que las componen, tal que la evaluación de todas las cláusulas sea verdadera con dicha asignación.

El problema SAT es un problema muy importante dentro del campo de la teoría de la complejidad, esto se debe a que SAT fue el primer problema que se identificó como NP-Completo. El Teorema de Cook demuestra que el algoritmo SAT pertenece a esta clase de algoritmos.

La importancia de este algoritmo no radica solamente en haber sido el primero en ser caracterizado como NP-Completo, se demostró que el problema SAT puede ser reducido al problema 3-SAT, que es básicamente el mismo problema pero en el cual todas las cláusulas tienen un máximo de 3 literales. Además de probar la reducción, se demostró que este problema también pertenece a la clase NP-Completo (A diferencia del problema 2-SAT, para el cual se conoce un algoritmo polinomial para resolverlo). Esta reducción del problema a 3-SAT es un resultado importante ya que luego para probar que otros problemas se encuentran también en esta clase se utilizaron reducciones a 3-SAT mostrando la equivalencia en cuanto a la complejidad de resolución.

## Situaciones de la vida real que se pueden modelar utilizando MAX-SAT

### Algoritmo exacto para MAX-SAT

Como su nombre lo indica, el algoritmo exacto para Max-Sat se encarga de resolver el problema exactamente, arrojando la asignación que valida la mayor cantidad de cláusulas posibles. Dado que no se conoce ningún algoritmo polinomial para resolver este problema, se implementaron 2 algoritmos de complejidad exponencial. Por un lado se implementó un algoritmo de fuerza bruta de simple implementación pero de muy baja eficiencia, en cuanto a tiempo de ejecución. Por otro lado se implementó un algoritmo exacto mediante backtracking para poder evitar visitar todas las asignaciones de las variables posibles.

### Algoritmo de fuerza bruta

En este algoritmo la idea es muy simple, se generan absolutamente todas las asignaciones posibles que existen, siendo estas  $2^v$  donde  $v$  es la cantidad de variables. Luego, por cada una de las asignaciones se verifica cuantas cláusulas valida, en el momento que una asignación supera el máximo de cláusulas hasta el momento, se actualiza la cantidad de cláusulas validadas, así como cual es la asignación que generó este máximo.

La idea de este algoritmo es tener una resolución muy simple del problema, es claro que el tiempo de ejecución va a ser muy malo ya que se revisan todas y cada una de las asignaciones posibles, y estas crecen en orden exponencial en

función de la cantidad de variables. Sin embargo, cabe destacar que el algoritmo provee una resolución exacta del problema y con baja probabilidad de errores dada la simpleza del mismo.

A continuación se presenta el pseudocódigo del mismo:

```

maxSatExacto(Vector clausulas, int variables)
vector asignacion
int max := 0
inicializar asignacion todos en falso
Para i = 1 hasta 2^variables
    int sat := 0
    Para j = 1 hasta tamano(clausulas)
        Si haceTrue(asignacion, clausulas[j])
            sat:= sat + 1
        fin si
    fin para
    si sat > max
        actualizar max
        actualizar asignacionMax
    fin si
    asignacion := siguiente(asignacion,i+1)
fin para
devolver asignacionMax, max

```

Lo que muestra el pseudocódigo anterior es como, por cada asignación posible, se mira cada clausula y si la función *haceTrue* devuelve true, entonces se suma 1 a la cantidad de satisfechas por esa asignación. Por último, se mira cual asignación es la que tiene más clausulas satisfechas.

La función *haceTrue* lo que hace es simplemente mirar toda la clausula pasada como parámetro y ver si algun literal esta asignado como verdadera, cuando encuentra uno deja de buscar y devuelve True. En caso contrario, si llega hasta el final de la clausula, devuelve False.

Por otro lado, la función *siguiente* se encarga de modificar para la asignación para probar con todas las posibles.

## Algoritmo de backtracking

Luego de implementar un algoritmo exacto por fuerza bruta, se buscó implementar un algoritmo también exacto pero tratando de lograr un menor tiempo de ejecución. El algoritmo implementado es un algoritmo exacto basado en la técnica de backtracking para lograr mejores resultados (en cuanto a tiempo de ejecución), si bien en la siguiente sección se verá que la complejidad temporal es la misma para ambos algoritmos exactos, la técnica de backtracking provee de herramientas para no tener que consultar necesariamente por cada una de las asignaciones posibles, es en estas podas que este algoritmo mejora los tiempos de ejecución del anterior.

La idea de este algoritmo es la siguiente: se genera un arbol de asignaciones, donde cada nivel del arbol *i*, representa todas las asignaciones posibles desde la variable 1 hasta la variable *i*. Este arbol, es un arbol binario dado que se arranca

de la asignación nula y de allí se abren dos caminos, asignarle False a la variable 1, o asignarle True. Luego, cada rama se va bifurcando sucesivamente por cada variable nueva. Como se puede ver, en el peor caso que tengamos que recorrer todo el árbol la cantidad de asignaciones nuevamente está dada por  $2^v$  al igual que en el algoritmo exacto.

Una vez que se tiene el árbol de backtracking lo que se hace es comenzar a recorrer el árbol en alguna dirección determinada. Cabe destacar que en el algoritmo implementado siempre se recorre primero la rama correspondiente a asignar falso a la variable y luego la otra.

La mejora del algoritmo radica en no recorrer todas las ramas posibles, esto se realiza de la siguiente manera: Al haber recorrido la primera rama del árbol llegando hasta una hoja, ya se tiene una mejor solución posible. Luego, cuando se está explorando una rama lo que se hace es fijarse si esa rama ya posee más cláusulas insatisfechas que la mejor solución hasta el momento. En caso afirmativo, la rama ya no sirve ya que no se podrá mejorar la solución y entonces se puede descartar todo el subárbol que pende de esa rama. Entonces se realiza el backtracking para ir por otro camino posible.

Las principales diferencias con el algoritmo de fuerza bruta son:

- La implementación es bastante más complicada ya que como se realiza backtracking, se debe guardar los estados intermedios de toda la rama que se está analizando para poder volver hacia atrás y tomar un nuevo camino. En el algoritmo exacto, cada asignación se contrasta con las cláusulas originales por lo que solo se deben guardar una vez todas las cláusulas.
- El tiempo de ejecución debería ser en la mayoría de los casos. Si bien el peor caso no cambiaría es importante destacar que las podas realizadas pueden traer grandes beneficios en cuanto al tiempo de ejecución. Si, por ejemplo, ya se tuviese una solución donde  $n$  cláusulas son insatisfechas y al asignar True a la variable 1,  $n+1$  cláusulas se vuelven insatisfacibles entonces se podría podar toda una mitad del árbol.
- El algoritmo de fuerza bruta no puede ser influenciado por alguna heurística, mientras que el algoritmo con backtracking sí. Lo que se quiere notar con esto, es que por más que ya se tenga una solución con  $n$  cláusulas insatisfechas, el algoritmo por fuerza bruta tiene que probar todas las asignaciones posibles; mientras que el algoritmo de backtracking ya puede comenzar podando ramas que tengan más de  $n$  cláusulas insatisfechas.

A continuación se presenta el pseudocódigo del algoritmo exacto con backtracking:

pseudo de backtracking

## Complejidad de algoritmos exactos

En primer lugar, se analizará la complejidad del algoritmo realizado por fuerza bruta.

El mismo presenta un ciclo que se realiza  $2^v$  veces, dado que esa es la cantidad total de asignaciones diferentes, siendo  $v$  la cantidad de variables. Una vez dentro de este ciclo, encontramos otro ciclo que itera sobre la totalidad de las cláusulas, haciendo que este ciclo se realice  $c$  veces.

Luego, dentro de este ciclo se llama a la función `hacerTrue`. Esta función lo que hace es fijarse en todos los literales de la cláusula si se encuentra asignado como verdadero. Dado que el hecho de fijarse es  $O(1)$ , ya que es mirar un array, y dado que la cláusula como mucho tiene  $2^v$  literales (todas las variables negadas y sin negar), se puede ver que esta función es  $O(v)$ .

Por último, las otras operaciones que se realizan dentro del ciclo principal tienen menor complejidad que lo mostrado anteriormente ya que son asignaciones que toman  $O(1)$  o es la función siguiente que toma  $O(v)$ .

Por lo mostrado anteriormente resulta que la complejidad temporal del algoritmo es  $O((2^v) * c * v)$ . Como se puede ver, lo importante más allá de la complejidad exacta, es que este algoritmo es exponencial en función de la cantidad de variables. Cabe destacar que si bien se podrían encontrar algoritmos con mejor complejidad, esta no podría ser menor que exponencial en el caso que se quieran revisar todas las asignaciones posibles.

Complejidad del algoritmo exacto con backtracking:

Para realizar el cálculo de complejidad de este algoritmo tomaremos el peor caso posible. El mismo consiste en que ninguna rama sea podada y por lo tanto se tengan que revisar todos los nodos posibles del árbol. En este caso el ciclo principal que itera sobre todas las asignaciones posibles sería  $O(2^v)$ , ya que este es el orden de la cantidad de nodos del árbol.

Luego, hay que analizar cuáles son las operaciones que se realizan cada vez que el algoritmo se encuentra en un nuevo nodo. En primer lugar, se llama a la función `resolver`, la misma tiene un funcionamiento análogo al explicado en el algoritmo por fuerza bruta; se fija en cada cláusula si la misma se hizo verdadera, y cuenta las insatisfacibles, por lo que esta función toma  $O(c * v)$  operaciones. Luego, lo que se hace es copiar todo el estado al siguiente nodo para que se pueda procesar, al realizar esto se copian varios parámetros. Sin embargo, el orden temporal de esta copia está dado por el orden que toma copiar todas las cláusulas, dado que los demás parámetros que se copian tienen menor orden espacial ya que son solamente vectores.

Se puede ver que copiar todo el estado toma  $O(c * v)$  operaciones ya que lo que se hace es copiar absolutamente todas las cláusulas, donde cada una pueda tener hasta  $2^v$  literales por lo justificado anteriormente.

Por último, resumiendo lo anteriormente explicado se puede ver que cuando se revisa un nodo se toma  $O(c * v)$  operaciones, y en el peor caso hay que revisar  $O(2^v)$  nodos; por lo que la complejidad total de este algoritmo es  $O((2^v) * c * v)$ .

Como se puede ver, la complejidad del algoritmo de backtracking no es mejor que el algoritmo exacto por fuerza bruta, sigue siendo exponencial. De hecho, se puede notar que en el peor caso posible, el algoritmo de backtracking es más lento que el algoritmo por fuerza bruta dado el overhead que produce copiar todo el estado al siguiente nodo del árbol para que se pueda procesar.

Sin embargo, en la mayoría de los casos, el algoritmo con backtracking presenta resultados temporalmente mejores que el algoritmo de fuerza bruta ya que se podan varias ramas haciendo que no se tenga que visitar todas las asignaciones posibles.

Cabe destacar que la ventaja temporal que gana el algoritmo de backtracking, genera una mayor complejidad espacial ya que en todo momento se debe tener todos los estados de los nodos de la rama que se está visitando para poder

caminar hacia atrás. Luego la complejidad espacial del algoritmo de fuerza bruta es  $O(c*v)$  ya que se guardan una vez todas las clausulas, mientras que la complejidad espacial del algoritmo de backtracking es  $O(c*v*v)$  ya que se guardan todas las clausulas una vez por cada nodo de la rama que se esta analizando, que a lo sumo tiene  $v$  nodos.

## Heurística constructiva para MAX-SAT

En esta sección se explicará el uso de una heurística constructiva para resolver el problema.

La motivación principal de utilizar diferentes heurísticas para resolver el problema de Max-Sat es que, como se vio en la sección anterior, los algoritmos exactos conocidos son de orden exponencial por lo que solo se pueden correr con instancias relativamente pequeñas.

La idea entonces es realizar un algoritmo que no sea exacto, sino que arroje una solución aproximada, pero en un tiempo polinomial para poder correr instancias más grandes.

En esta primer aproximación, se realizó una heurística constructiva, esto quiere decir que se va construyendo una solución mediante algún criterio que se supone (y luego se verá que se puede demostrar) que es adecuado para obtener un buen resultado.

En este caso, la heurística constructiva seguirá una estrategia golosa para ir armando la solución. Lo que hace esta heurística es revisar todas las clausulas buscando cual es el literal que más se repite. Una vez encontrado dicho literal lo que se hace es asignar True a este literal y actualizar las clausulas en base a esta asignación.

Actualizar las clausulas consiste en dos tareas, en primer lugar se borran todas las clausulas que contenian este literal ya que ya fueron satisfechas mediante la asignación y se actualiza el contador de clausulas satisfechas. Por otro lado, en las clausulas restantes, se borra el literal negado ya que este no sirve para futuras elecciones porque al asignar True al literal elegido, el literal negado tendra necesariamente valor falso.

Luego de realizar esta actualización, se continua iterativamente con esta estrategia golosa hasta encontrar una asignación completa de las variables.

A continuación se muestra el pseudocódigo de la heurística constructiva:

```
Constructiva(Vector clausulas, int V)
Max := 0
Comenzar con asignacion vacia
mientras asignacion no este completa
    Tomar literal l que mas se repita
    Asignar True a l
    Para i de 1 hasta tamano(clausulas)
        Si esta(l, clausula[i])
            borrar clausula[i]
            Max := Max + 1
    fin si
fin para
Para i de 1 hasta tamano(clausulas)
```

```

        Si esta(-1, clausula[i])
            borrar(-1, clausula[i])
        fin si
    fin para
fin mientras
devolver asignacion y Max

```

## Aproximación del resultado

Es importante destacar que a priori esta heurística podría arrojar resultados que no se acercasen al verdadero valor del problema. La heurística parece razonable ya que siempre se elige el literal que más clausulas satisface, y luego se va iterando hasta que no queden más variables para asignar. Es importante destacar que si bien la heurística parece razonable se puede encontrar rápidamente casos en los cuales esta aproximación golosa no nos devuelva el resultado exacto. Dada estas observaciones, se buscó encontrar algún tipo de cota para el resultado entregado por la heurística, en función del verdadero resultado.

Dada esta heurística golosa, se puede ver que en realidad se trata de un algoritmo aproximado. Si se denota  $c_{max}$  como la cantidad máxima de clausulas satisfacibles,  $c_h$  al resultado entregado por la heurística y  $m$  al mínimo número de literales contenido en cualquiera de las clausulas, se cumple que<sup>1</sup>:

$$c_h \geq \frac{m}{m+1} * c_{max} \quad (1)$$

Dada la ecuación anterior, se tiene una cota para el resultado que depende del mínimo número de literales en cualquier clausula. Cabe destacar que en la implementación de esta heurística, y en los casos de prueba utilizados, no se hace ninguna mención particular a la cantidad mínima de literales de las clausulas. En este caso, podemos acotar por el peor caso que sería tener clausulas de un solo literal.

En dicho caso, la heurística utilizada sería un algoritmo  $\frac{1}{2}$ -aproximado, es decir que el resultado arrojado jamás se encontrara por debajo de la mitad del valor real del problema. En el caso del que el mínimo número de clausulas suba (lo que suele pasar en la práctica), la cota también se ajustará dando una mejor aproximación.

## Complejidad de la heurística constructiva

No quiero mentir mucho así que pregunto y despues escribo

## Heurística de búsqueda local para MAX-SAT

La segunda heurística implementada fue la de busqueda local. Como su nombre lo indica, lo que hace esta heurística es pararse sobre una solución y buscar localmente si se puede mover a otra solución *cercana* con más clausulas satisfechas. Dada la explicación anterior, se desprende que entonces se debe definir que significa que dos soluciones cercanas. Teniendo una asignación posible, se

---

<sup>1</sup>Cita Bibliográfica 1



define una vecindad de asignaciones mediante algún criterio que serán las asignaciones *cercanas* mencionadas anteriormente.

La idea principal de la heurística es empezar con alguna asignación posible y luego buscar en la vecindad de esta solución si existe alguna asignación que satisfaga mayor cantidad de variables, si existe esta nueva asignación, entonces se cambia la asignación máxima actual por esta nueva y se vuelve a realizar una búsqueda local, ahora con la vecindad de la nueva asignación.

Esta heurística finaliza cuando tenemos una asignación que es la mejor de todas entre ella y las asignaciones de su vecindad. Es decir, cuando se encuentra un mínimo local, la heurística para y devuelve la mejor asignación encontrada hasta el momento.

Es importante destacar que la heurística de búsqueda local necesita de una asignación inicial para comenzar a definir su vecindad y luego realizar la búsqueda. Como una primera aproximación se podría inicializar la heurística con cualquier asignación, por ejemplo todas las variables en falso, el problema es que para caso de prueba diferente el espacio de asignaciones tiene una forma diferente. Esto puede llevar a que si inicializamos en cualquier asignación, se encuentre rápidamente un máximo local que poco tenga que ver con la solución real del problema.

La solución que se utilizó para el problema explicado anteriormente es que la heurística de búsqueda local sea inicializada con el resultado proveniente de aplicar la heurística constructiva.

Es importante notar que la heurística de búsqueda local no tiene permitido moverse a una asignación que satisfaga una menor cantidad de cláusulas, por lo que se puede ver que el resultado arrojado por la búsqueda local es mejor o igual que el resultado de la heurística constructiva en el caso de que se inicialice el procedimiento con el resultado de la heurística constructiva.

De esta manera, reemplazando en la ecuación (1) (denotado  $c_{local}$  al resultado de la búsqueda local):

$$c_{local} \geq c_{constr} \geq \frac{m}{m+1} * c_{max} \quad (2)$$

Luego, dada la ecuación anterior y con una explicación análoga a la propuesta en la heurística constructiva, se puede ver que esta heurística (inicializada de esta manera) también se trata de un algoritmo  $\frac{1}{2}$ -aproximado y, de la misma manera, a medida que el  $m$  va subiendo, la cota va mejorando.

## Detalles de implementación

En primer lugar, es importante destacar como se definió la vecindad de una asignación, para tener bien definido a donde se va a realizar la búsqueda local dada una asignación. Se determinó que la vecindad de una asignación está dada por todas las asignaciones que solo distan en una variable. Es decir, las asignaciones que tienen todas las variables asignadas igual excepto por una.

Luego, se decidió crear una estructura que dada cada variable, se tiene listado en que cláusulas aparece y si el literal que aparece es la variable o su negación. Sumado a esto, se tiene listado por cada cláusula que literales tienen. Por otro lado, se tiene un vector de valores de verdad que indican los estados de las variables y un vector de enteros que indica por cada cláusula cuantos literales hacen verdadera la cláusula.

Dada esta estructura, se procede hacer la búsqueda local hasta caer en una solución inmejorable, respecto de la vecindad definida anteriormente. Es decir, se toma la asignación actual y se va modificando de a una variable y se actualizan las estructuras adecuadamente para ver si esta nueva configuración es mejor que la actual.

El pseudocódigo de la heurística de búsqueda local es:

```

BusquedaLocal(Vector clausulas, int V)
Max := HeuristicaConstructiva(clausulas, v)
asignacionMax := HeuristicaConstructiva(clausulas, v)
mientras cambie la asignacion
    asignacionMaxTemp := asignacionMax
    para toda asignacion_i en la vecindad de asignacionMax
        si asignacion_i satisface mas que asignacionMax
            asignacionMaxTemp := asignacion_i
    fin si
    fin para
    asignacionMax := asignacionMaxTemp
fin mientras
devolver asignacion y Max

```

## Complejidad de la heurística de búsqueda local

ver un poco mas para no mentir

## Metaheurística de búsqueda tabú para MAX-SAT

La metheurística de búsqueda tabu surge para mejorar la heurística de búsqueda local, ya que esta puede caer en máximos locales que esten muy lejos del máximo absoluto del problema.

Es por esto que la búsqueda tabú resulta de modificar la búsqueda local con la motivación de poder salir de asignaciones que resultan óptimas localmente; es decir, asignaciones que satisfacen una cantidad de clausulas mayor o igual a las que satisface cualquier otra asignación en su vecindad.

La metaheurística de búsqueda tabú consiste basicamente en modificar el vecindario de cada asignación. Esto se realiza excluyendo asignaciones que fueron visitadas recientemente y relajando el criterio de parada de la búsqueda local, permitiendo que se continúe la búsqueda aún cuando la mejor asignación del vecindario no aumente la cantidad de clausulas satisfechas.

Es importante destacar que la exclusión de las asignaciones visitadas más recientemente no es un detalle menor. Si no se hiciese esto se entraría en un ciclo entre el óptimo local y la mejor asignación de su vecindad, sin poder alejarnos de estas asignaciones, para buscar mejores soluciones.

Luego de lo explicado anteriormente, se denota que para poder separar las asignaciones visitadas recientemente se debe implementar algún tipo de memoria. Una aproximación inicial sería guardar una lista de longitud acotada con las últimas asignaciones visitadas y excluir dichas asignaciones del vecindario que se considere. Se denominará lista Tabú a dicha lista.

En esta primera implementación presentada, ya se pueden encontrar varias sutilezas que resultan factores importantes para el desarrollo de la metaheurística. Por ejemplo, la longitud de la lista Tabú es acotada y la cota para la misma puede ser fija, pero también podría estar expresada en función de algún parámetro o incluso podría variar durante la ejecución. Algo similar ocurre con el criterio de parada; es común que esté expresado como una combinación de restricciones en la cantidad máxima de iteraciones y la cantidad de iteraciones desde la última asignación que mejoró la cantidad conocida de cláusulas satisfechas.

Si bien la primera aproximación indica que la lista Tabú sería una lista de asignaciones, es importante destacar que esta implementación resulta ineficiente, debido a que se debe recorrer toda la lista para saber si una asignación está o no. Esto resulta ineficiente además porque la comparación entre asignaciones es lenta, ya que es necesario comparar variable por variable para ver si se realmente coinciden.

La implementación de la lista Tabú como una lista de asignaciones resulta ineficiente, debido a que se debe recorrer toda la lista para saber si una asignación está o no y porque la comparación entre asignaciones resulta lenta, ya que es necesario comparar variable por variable. Una alternativa que se encarga del último problema consiste en representar el vecindario de una asignación en términos de movimientos que se pueden realizar desde la misma para llegar hasta otra asignación. Si los movimientos que consideramos son reversibles, entonces podemos guardar en la lista Tabú los movimientos opuestos a los que realizamos, evitando de esta manera entrar en un ciclo. Vamos a llamar movimientos Tabú a aquellos que se encuentran en la lista Tabú. Podría ocurrir que un movimiento a una asignación que todavía no visitamos sea Tabú, y ésta asignación puede incluso satisfacer más cláusulas que cualquier otra visitada con anterioridad. En casos como éste sería deseable ignorar la condición de Tabú del movimiento. De esta manera surge la función de aspiración. En este problema decidimos ignorar la condición de Tabú de un determinado movimiento aplicado a una asignación si la asignación resultante satisface más cláusulas que la mejor asignación visitada hasta el momento.

Implementación: Habiendo definido conceptualmente los componentes de la búsqueda Tabú se pueden mencionar detalles de implementación. La vecindad que definimos para cada asignación está compuesta por las asignaciones resultantes de cambiar el valor de exactamente una variable de la asignación original. Cada movimiento es el cambio de una variable, por lo tanto manteniendo un arreglo de enteros del tamaño de la cantidad de variables podemos llevar un registro de cuál fue la última iteración en la que se aplicó el movimiento  $i$ . De ésta manera podemos saber si un movimiento es Tabú con sólo ver si pasaron suficientes iteraciones desde la última iteración en la que fue usado, lo que resulta más eficiente que recorrer una lista con los movimientos.

## Bibliografía

- Algorithms for the Maximum Satisfiability Problem