Índice

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 es decidir si un conjunto de clausulas en forma normal conjuntiva, tiene alguna asignación de las variables que las componenen, tal que la evaluación de todas las clasulas sea verdadera con dicha asignación.

El problema SAT es un problema muy importante dentro del campo de la teoria 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 basicamente el mismo problema pero en el cual todas las clausulas tienen un máximo de 3 literales. Además de probar la reducción, se demostro 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 tambien 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 asignacion que valida la mayor cantidad de clausulas 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 implemento 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 clausulas valida, en el momento que una asignación supera el máximo de clausulas hasta el momento, se actualiza la cantidad de clausulas 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 tamanio(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 asignacion 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 arbol la cantidad de asignaciones nuevamente esta dada por 2^v al igual que en el algoritmo exacto.

Una vez que se tiene el arbol de backtracking lo que se hace es comenzar a recorrer el arbol 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 primer rama del arbol llegando hasta una hoja, ya se tiene una mejor solución posible. Luego, cuando se este explorando una rama lo que se hace es fijarse si esa rama ya posee más clausulas insatisfechas que la mejor solución hasta el momento. En caso afirmativo, la rama ya no sirve ya que no se podra mejorar la solución y entonces se puede descartar todo el subarbol 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 esta analizando para poder volver hacia atras y tomar un nuevo camino. En el algoritmo exacto, cada asignacion se contrasta con las clausulas originales por lo que solo se deben guardar una vez todas las clausulas.

- El tiempo de ejecución deberia 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 clausulas son insatisfechas y al asignar True a la variable 1, n+1 clausulas se vuelven insatisfacibles entonces se podría podar toda una mitad del arbol.

- El algoritmo de fuerza bruta no puede ser influenciado por alguna heurística, mientras que el algoritmo con backtraking si. Lo que se quiere notar con esto, es que por más que ya se tenga una solucion con n clausulas 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 clausulas 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 clausulas, haciendo que este ciclo se realice c veces.

Luego, dentro de este ciclo se llama a la funcion haceTrue. Esta función lo que hace es fijarse en todos los literales de la clausula 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 clausula 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 funcion 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 alla de la complejidad exacta, es que este algoritmo es exponencial en función de la cantidad de variables. Cabe destacar que si bien se podrián 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 arbol. 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 arbol.

Luego, hay que analizar cuales 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 clausula si la misma se hizo verdadera, y cuenta las insatisfacibles, por lo que esta funcion 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 esta dado por el orden que toma copiar todas las clasulas, 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 clausulas, 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 arbol 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 esta visitando para poder

caminar hacia atras. 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 tamanio(clausulas)
         Si esta(l, clausula[i])
             borrar clausula[i]
             Max := Max + 1
         fin si
      fin para
      Para i de 1 hasta tamanio(clausulas)
```

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

Heurística de búsqueda local para MAX-SAT

Metaheurística de búsqueda tabú para MAX-SAT