

# Capítulo 7

## Aprendizaje Basado en Grafos

- Estos sistemas, al igual que los de programación lógica inductiva, se utilizan para dominios estructurados. Algunos de estos dominios son:
  - Bioinformática (compuestos químicos)
  - Visión por computadora
  - Recuperación de Texto
  - Análisis de archivos web-logs
- Encuentran subestructuras comunes a partir de un conjunto de grafos y son útiles para:
  - Caracterizar conjuntos de grafos
  - Discriminar diferentes grupos de grafos
  - Clasificar y Agrupar grafos
  - Construir índices de grafos
- Hay dos formas principales de buscar estas subestructuras frecuentes:
  - *Con* generación de candidatos
    - \* Primero genera subestructuras candidatas
    - \* Después verifica la frecuencia de esas subestructuras
    - \* Como lo hace el algoritmo de Agrawal para encontrar reglas de asociación

Tabla 7.1: Apriori-like

```

Apriori( $D, min\_support, S_k$ )
   $S_{k+1} \leftarrow \emptyset$ 
  for each frequent  $g_i \in S_k$  do
    for each frequent  $g_j \in S_k$  do
      for each size  $(k + 1)$  graph  $g$  formed by the merge of  $g_i$  and  $g_j$  do
        if  $g$  is frequent in  $D$  and  $g \notin S_{k+1}$  then
          insert  $g$  to  $S_{k+1}$ ;
  if  $S_{k+1} \neq \emptyset$  then
    call Apriori( $D, min\_support, S_{k+1}$ );
  return
end

```

- \* Realizan una operación "join" entre dos o más subestructuras frecuentes para generar una nueva subestructura candidata
  - \* Realizan una búsqueda Breadth First Search (BFS) para generar todos los candidatos de tamaño-k
  - \* Estos métodos tienen dos pasos muy tardados
    - Unir dos grafos frecuentes de tamaño-k para generar grafos candidatos de tamaño-k+1
    - Verificar la frecuencia de los candidatos en otro paso
    - Ejemplos de estos algoritmos son: AGM, FSG, path-join
  - \* Estos pasos son el cuello de botella para los algoritmos tipo apriori o con generación de candidatos
  - \* El sistema Subdue, aunque genera candidatos, no lo hace al estilo apriori
  - \* La tabla ?? muestra el algoritmo general de los métodos tipo Apriori.
- Sin generación de candidatos
- \* Creados para evitar el cuello de botella en que caen los algoritmos que generan candidatos

Tabla 7.2: Pattern Growth

**PatternGrowth**( $g, D, min\_support, S$ )

Input: A frequent graph  $g$ , a graph dataset  $D$ , and  $min\_support$ .

Output: A frequent substructure set  $S$ .

```
if  $g \in S$  then return;  
else insert  $g$  to  $S$ ;  
scan  $D$  once, find all the edges  $e$  such that  $g$  can be extended to  $g \otimes e$ ;  
for each frequent  $g \otimes e$  do  
    Call PatternGrowth( $g \otimes e, D, min\_suppport, S$ );  
return  
end
```

- \* Utilizan un método en que van creciendo el patrón
- \* Extienden un patrón a partir de un solo patrón
- \* Al extender el patrón ya se esta seguro de que es frecuente, ya no se generan candidatos
- \* Pueden utilizar BFS o DFS
- \* Ejemplos de estos algoritmos son: gSpan, MoFa, FFSSM, SPIN, Gaston
- \* La tabla ?? muestra el algoritmo general de los métodos tipo crecimiento de patrones.
- \* Diferentes algoritmos realizan el proceso de crecimiento de patrones de manera eficiente tratando de podar lo más posible el espacio de búsqueda.

## 7.1 Introducción a Subdue

En esta sección se presenta brevemente los conceptos en que se basan los sistemas Subdue y SubdueCL.

*Representación de Conocimiento:*

- En un sistema basado en grafos, el algoritmo de aprendizaje utiliza grafos como representación de conocimiento.
- Esto significa que la fase de preparación de datos incluye una transformación de los datos a un formato de grafo.

*Espacio de Búsqueda:*

- El espacio de búsqueda el algoritmo basado en grafos consiste en todos los sub-grafos que se pueden derivar a partir del grafo de entrada.
- Esto quiere decir que el espacio de búsqueda es exponencial
- de la misma manera que es el tiempo de ejecución de estos algoritmos
- al menos que se restrinjan de alguna manera para que corran en tiempo polinomial

*Criterio de Evaluación:*

- Una parte muy importante del algoritmo de minería de datos es el criterio de evaluación.
- Este criterio se utiliza para determinar cuales subgrafos del espacio de búsqueda son relevantes y pueden ser considerados como parte de los resultados.
- El método basado en grafos *Subdue* utiliza el principio de longitud de descripción mínima (MDL) para evaluar los subgrafos descubiertos.
- El principio MDL dice que la mejor descripción del conjunto de datos es aquella que minimiza la longitud de la descripción de todo el conjunto de datos.
- En el método basado en grafos, el principio MDL se utiliza para determinar que tan bien un grafo comprime al grafo de entrada.

- De esta manera, todos los subgrafos que se generan durante el proceso de búsqueda se evalúan de acuerdo al principio MDL y los mejores subgrafos se eligen como parte del resultado.

## 7.2 Implementación en el sistema Subdue

- El método basado en grafos descrito anteriormente fue implementado en el sistema *Subdue* (Cook and Holder 1994).
- Subdue es un sistema de aprendizaje relacional utilizado para encontrar subestructuras (subgrafos) que aparecen repetidamente en la representación basada en grafos de bases de datos.
- Una vez que la base de datos está representada con grafos, *Subdue* busca la subestructura que mejor comprime al grafo utilizando el principio MDL.
- Después de encontrar esta subestructura, *Subdue* comprime el grafo y puede iterar repitiendo este proceso.
- *Subdue* tiene la capacidad de realizar un cacheo inexacto que permite descubrir subestructuras con pequeñas variaciones.
- Otra característica importante de *Subdue* es que permite utilizar conocimiento previo representado como subestructuras predefinidas.

### 7.2.1 Representación de Conocimiento

- El modelo de representación que utiliza *Subdue* es un grafo etiquetado.
- Los objetos se representan con vértices y las relaciones con arcos.
- Las etiquetas se utilizan para describir el significado de los arcos y vértices.
- Cuando se trabaja con bases de datos relacionales, cada renglón se puede considerar como un evento y los atributos como objetos.

- Los eventos también se pueden ligar a otros eventos por medio de arcos.
- Los atributos de los eventos se describen mediante un conjunto de vértices y arcos, donde los arcos identifican los atributos específicos y los vértices especifican el valor de ese atributo para el evento.
- Una representación basada en grafos es lo suficientemente flexible para permitir tener más de una representación para un dominio dado, permitiendo al investigador experimentar para obtener la mejor representación para su dominio.
- La definición de los grafos tiene un formato específico que se da como entrada al sistema *Subdue*. Para esta sección definiremos los siguientes términos.

*Algunos Términos:*

- Un **subgrafo**  $G'$  de  $G$  es un grafo conectado cuyos vértices y arcos son subconjuntos de  $G$ .
- Una **subestructura**  $S$  es un subgrafo que tiene asociada una descripción y un conjunto de instancias en el grafo de entrada.
- Una **instancia** es una ocurrencia de una subestructura  $S$  en un grafo  $G$ .

### 7.2.2 Método de Búsqueda

- *Subdue* utiliza una búsqueda tipo beam (restringida computacionalmente) para encontrar subestructuras.
- Una subestructura es un subgrafo contenido en el grafo de entrada.
- El algoritmo inicia con un solo vértice como subestructura inicial y en cada iteración expande las instancias de aquella subestructura añadiendo un arco en cada posible manera.
- De esta forma genera nuevas subestructuras que podrían considerarse para expansión.

Tabla 7.3: Algoritmo de Búsqueda de Subdue

```

Subdue(Graph, Limit, Beam, NumBestSubs)
  ProcessedSubs = 0
  ParentList = All substructures of one vertex in Graph
  while(ProcessedSubs ≤ Limit)
    ChildList = {}
    while(ParentList ≠ {})
      Parent = (RemoveSubstructure(ParentList))
      Instances = Extend(Parent) en todas las maneras posibles
      ChildSubs = Group(Instances)
      Evaluate(ChildSubs)
      Insert substructures in ChildSubs into ChildList mod Beam
      ProcessedSubs = ProcessedSubs + 1
      Insert Parent into BestList mod NumBestSubs
    end while
    ParentList = ChildList
  end while
  return BestList
end

```

- El método de búsqueda también puede sesgarse utilizando conocimiento previo (p.e. subestructuras que creemos que pueden existir en los datos, pero que queremos estudiar con mayor detalle) dadas por el usuario (Cook and Holder 1994).
- En este caso, el usuario provee subestructuras de conocimiento previo como entrada a *Subdue*.
- *Subdue* encuentra instancias de las subestructuras de conocimiento previo en el grafo de entrada y continúa buscando extensiones de aquellas subestructuras.
- El algoritmo de búsqueda de *Subdue* se muestra en la tabla ??.

- El algoritmo de búsqueda inicia con la creación de una subestructura de cada etiqueta de vértice y sus instancias asociadas, las cuales se insertan en *ParentList*.
- Después, cada subestructura de *ParentList* se extiende en cada posible manera añadiendo un vértice y un arco o solo un arco (en el caso de que el arco ligue dos vértices que ya existan en la subestructura).
- Las instancias resultantes de la extensión se agrupan en subestructuras.
- La primera aparición de cada instancia se convierte en la definición de subestructura, y el resto se asocia a esa subestructura como instancias.
- Después, todas las subestructuras producidas se evalúan de acuerdo al principio MDL y se insertan en *ChildList* ordenadas por su valor.
- *ChildList* mantiene tantas subestructuras como el valor del parámetro *Beam*.
- Posteriormente, la subestructura *Parent* se inserta en *BestList* (la cual mantiene solo *NumBestSubs* subestructuras), y este proceso continúa hasta que *ParentList* se vacíe.
- En este momento *ParentList* y *ChildList* se intercambian y se repite el proceso.
- El número de subestructuras *Parent* consideradas en la búsqueda esta restringido por el parámetro *Límite*.
- El valor por default de este límite se deriva de el número de arcos y vértices en el grafo de entrada ( el número de vértices mas el número de arcos dividido por dos).
- *BestList* esta restringida por default a tener el mismo tamaño que la longitud del beam *Beam*, pero las restricciones de longitud se pueden modificar independientemente utilizando los parámetros del sistema.

### 7.2.3 Criterio de Evaluación

- En *Subdue* se implementó un criterio de evaluación para decidir cuales patrones se van a elegir como conocimiento importante o estructuras.

- El método del modelo de evaluación se llama *Codificación Mínima* (Minimum Encoding), una técnica derivada el *Principio de Longitud de Descripción Mínima MDLP* (Rissanen 1989), el cual dice que la mejor descripción de un conjunto de datos es aquella que minimiza la longitud de la descripción de todo el conjunto de datos.
- En relación a *Subdue*, la mejor descripción del conjunto de datos es aquella que minimiza  $I(S) + I(G|S)$ , donde
- $S$  es la subestructura utilizada para describir el grafo de entrada  $G$
- $I(S)$  es la longitud (en número de bits) requerido para codificar  $S$
- e  $I(G|S)$  es la longitud del grafo codificado  $G$  después de ser comprimido utilizando la subestructura  $S$ .
- Cada vez que *Subdue* encuentra una subestructura, la evalúa utilizando el principio MDL y compara su valor con el de las otras subestructuras.
- *Subdue* elige la subestructura que mejor comprime el grafo en términos del principio MDL.
- Después, reemplaza las instancias de la subestructura con un solo vértice que la representa a través del grafo e inicia una nueva iteración en busca de nuevas subestructuras que incluso pueden contener subestructuras encontradas en iteraciones previas.
- El número de iteraciones es un parámetro de entrada a *Subdue*.
- Existen varias formas de limitar la búsqueda de *Subdue*.
- Una es utilizando una técnica de podado (también es un parámetro de entrada a *Subdue*) que termina la búsqueda cuando la evaluación MDL ya no mejora.
- Otra forma de limitar la búsqueda es definir un tamaño máximo para las subestructuras que *Subdue* busca en términos del número de vértices.
- Es posible combinar estas técnicas para delimitar la búsqueda de *Subdue*.

## 7.2.4 Subestructuras Predefinidas

- Es posible guiar a Subdue para que encuentre subestructuras que creemos que estan en la base de datos.
- Para esto especificamos a Subdue subestructuras predefinidas en un archivo separado y también en su representación basada en grafos.
- La ventaja de utilizar subestructuras predefinidas es que guiamos la búsqueda hacia una meta específica y al mismo tiempo se reduce la complejidad de la búsqueda.
- Por supuesto que esto solo funciona si la subestructura predefinida existe en la base de datos, por lo que esta característica la explota mejor el experto en el dominio.

## 7.2.5 Macheo Inexacto de Grafos

- Subdue tiene la capacidad de encontrar subestructuras con ligeras diferencias en sus instancias.
- Estas diferencias pueden ser causa de ruido o por la naturaleza de la información.
- Algunas de estas pequeñas diferencias pueden ser un vértice adicional o uno mejor, una etiqueta diferente en un vértice, un arco que no existe en una instancia, etc.
- La manera en que Subdue maneja el macheo inexacto es asignando un costo a cada diferencia que encuentra en la nueva instancia y lleva un registro del costo total de las diferencias de la nueva instancia con respecto a la original.
- Si el costo es menos que un umbral (este umbral se da como parámetro), entonces se considera que la nueva instancia hace un macheo con la original.
- Se utilizan reglas para asignar un costo a cada tipo de diferencia, estas reglas se ajustan de acuerdo al dominio.

- El procedimiento de macheo de grafos esta restringido a ser polinomial con respecto al tamaño de los grafos que se comparan.

## 7.3 Aprendizaje de Conceptos basado en Grafos

- Los sistemas basados en lógica han dominado el área de aprendizaje de conceptos relacional, en especial los sistemas de programación lógica inductiva *Inductive Logic Programming (ILP)* (Muggleton and Feng 1992).
- Sin embargo, la lógica de primer orden también se puede representar como un grafo y de hecho, la lógica de primer orden es un subconjunto de lo que se puede representar utilizando grafos (Sowa 1992).
- Entonces, los sistemas de aprendizaje que utilizan representaciones gráficas también tienen el potencial de aprender conceptos ricos si pueden manejar el incremento en el tamaño del espacio de hipótesis.

### 7.3.1 Modelo

- Como se ha mencionado, el aprendizaje de conceptos es un proceso que consiste en la inducción de una función *concepto* a partir de ejemplos de entrenamiento positivos y negativos.
- Para el aprendizaje de conceptos basado en grafos, se utiliza un conjunto de ejemplos positivos y negativos en su representación con grafos para entrenar y encontrar el concepto que describe el dominio.
- La meta es que el concepto encontrado debe ser capaz de predecir si un nuevo ejemplo (uno que no fue utilizado durante el entrenamiento) pertenece al concepto o no.
- Los grafos son una buena representación para datos estructurales y capaz de representar FOPC.
- El espacio de hipótesis consiste de todos los subgrafos que se pueden derivar a partir de los grafos de ejemplos positivos; este espacio es

exponencial con respecto al tamaño de los grafos (número de vértices y arcos).

- El criterio de evaluación se basa en el número de ejemplos positivos y negativos que describe la hipótesis a partir del conjunto de entrenamiento.
- Una buena hipótesis es aquella que describe a los ejemplos positivos pero no a los ejemplos negativos.
- El método de aprendizaje de conceptos *SubdueCL* sigue el paradigma *set-covering*.
- Esto implica que el concepto resultante puede consistir de un conjunto de subconceptos.
- El proceso de aprendizaje inicia con el conjunto de ejemplos positivos y negativos.
- Cuando se encuentra un sub-concepto, todos los ejemplos positivos cubiertos por él se quitan del conjunto de entrenamiento y se inicia una búsqueda de un nuevo sub-concepto.
- Este proceso se repite hasta que todos los ejemplos positivos se han descrito a través de cualquiera de los sub-conceptos encontrados.
- La hipótesis resultante es un conjunto de sub-conceptos en DNF (Disjunctive Normal Form).
- El concepto resultante se utiliza para clasificar nuevos ejemplos (no incluidos en el conjunto de entrenamiento).
- El ejemplo se prueba por el primer sub-concepto en la DNF.
- Si el ejemplo descrito por el sub-concepto, entonces el ejemplo se clasifica como positivo.
- Si el ejemplo no fue descrito por el primer sub-concepto, se prueba con el segundo sub-concepto.
- Si el segundo sub-concepto describe el ejemplo, entonces el ejemplo es positivo, y si no, se utiliza el siguiente sub-concepto para probar el nuevo ejemplo.

- Este proceso continúa hasta que uno de los sub-conceptos en la DNF clasifica al ejemplo como positivo o todos los sub-conceptos se probaron sin ningún resultado positivo, lo cual significa que el ejemplo se clasifica como negativo.

### 7.3.2 Implementación

- Para extender Subdue para poder realizar la tarea de aprendizaje de conceptos se incluyó el manejo de ejemplos negativos al proceso.
- Las subestructuras que describen ejemplos positivos, pero no ejemplos negativos, son las que tienen más posibilidades de representar el concepto deseado.
- Por lo tanto, la versión de aprendizaje de conceptos de Subdue, que conocemos como SubdueCL, acepta ejemplos positivos y negativos en formato de grafo.
- Como SubdueCL es una extensión de Subdue, utiliza el núcleo de funciones de Subdue para realizar operaciones con grafos, pero el proceso de aprendizaje cambia.
- SubdueCL trabaja como un algoritmo de aprendizaje supervisado, que diferencia ejemplos positivos de los negativos utilizando un método *set-covering* en lugar de compresión de grafos.
- La hipótesis encontrada por SubdueCL consiste de un conjunto de disyunciones de conjunciones (subestructuras, por ejemplo, el concepto puede contener varias reglas).
- SubdueCL forma una de esas conjunciones (reglas) en cada iteración.
- Los grafos de ejemplos positivos descritos por la subestructura encontrada en una iteración anterior se eliminan del grafo de entrada para las iteraciones siguientes.

### 7.3.2.1 Evaluación de Subestructuras

- La manera en que SubdueCL decide si las subestructuras (o reglas) formarán parte del concepto o no es diferente a Subdue.
- SubdueCL utiliza una fórmula de evaluación para asignar un valor a cada una de las subestructuras generadas.
- Esta fórmula asigna un valor a una subestructura de acuerdo a que tan bien describe a los ejemplos positivos (o subconjunto de los ejemplos positivos) sin describir ejemplos negativos.
- De esta manera, los ejemplos positivos cubiertos por la subestructura incrementan el valor de la misma mientras que los ejemplos negativos decrementan su valor.
- En esta fórmula los ejemplos positivos que no son cubiertos y los negativos cubiertos por la subestructura se consideran errores porque la subestructura ideal será una que cubre todos los ejemplos positivos sin cubrir ningún ejemplo negativo.
- El valor de la subestructura se calcula con la siguiente ecuación:

•

$$value = 1 - Error$$

- donde el error se calcula con respecto a los ejemplos positivos y negativos cubiertos por la subestructura utilizando la siguiente fórmula:

•

$$Error = \frac{\#PosEgsNotCovered + \#NegEgsCovered}{\#PosEgs + \#NegEgs}$$

- Utilizando esta ecuación, SubdueCL elige reglas que maximizan el valor de las subestructuras y de esta manera minimiza el número de errores hechos por la subestructura utilizada para formar el concepto.
- Los ejemplos positivos no cubiertos por la subestructura y los ejemplos negativos cubiertos por ella se consideran errores.

- $\#PosEgsNotCovered$  es el número de ejemplos positivos no cubiertos y  $\#NegEgsCovered$  es el número de ejemplos cubiertos cubiertos.
- $\#PosEgs$  es el número de ejemplos positivos que quedan en el conjunto de entrenamiento (recordando que los ejemplos positivos que ya fueron cubiertos en alguna de las iteraciones anteriores ya fueron removidos del conjunto de entrenamiento), y  $\#NegEgs$  es el número total de ejemplos negativos.
- Este número no cambia porque los ejemplos negativos no se remueven del conjunto de entrenamiento.
- El problema de la ecuación 2 es que cuando dos subestructuras tienen el mismo error, nos gustaría elegir aquella que cubra más ejemplos positivos.
- Por ejemplo, suponiendo que tenemos 10 ejemplos positivos y 10 negativos.
- La subestructura  $S1$  cubre 5 ejemplos positivos y 0 negativos, y la subestructura  $S2$  cubre 10 ejemplos positivos y 5 negativos.
- En este caso ambas subestructuras tienen un error de  $\frac{1}{4}$  de acuerdo a la ecuación 2 pero preferimos elegir  $S1$  porque no cubre ningún ejemplo negativo.
- Para hacer esto, se asigna una penalización (sea  $k$ ) a los errores negativos.
- Después de alguna manipulación matemática expresamos el error con la siguiente fórmula, donde  $k$  es un peso de penalización y  $k \geq 2$ .
- $$\text{PenaltyError} = \frac{\#PosEgsNotCovered + (k * \#NegEgsCovered - \#NegEgs * (1 - k))}{\#PosEgs + \#NegEgs}$$
- El valor por default de  $k$  es de 3.
- Ahora el error de  $S1$  y  $S2$  de acuerdo a la fórmula 3 (y con el valor por default de  $k = 3$ ) es de  $\frac{5}{4}$  y  $\frac{7}{4}$  respectivamente.
- Con esta fórmula, SubdueCL preferira  $S1$  sobre  $S2$ .
- Los resultados de un análisis empírico mostraron que esta ecuación para evaluar subestructuras funciona muy bien.

Tabla 7.4: Algoritmo Principal de Subdue

```

Main( $G_p, G_n, Limit, Beam$ )
   $H = \{\}$ 
  repeat
    repeat
       $BestSub = \text{SubdueCL}(G_p, G_n, Limit, Beam)$ 
      if  $BestSub = \{\}$ 
        then  $Beam = Beam * 1.1$ 
      until( $BestSub \neq \{\}$ )
       $G_p = G_p - \{p \in G_p | BestSub \text{ covers } p\}$ 
       $H = H + BestSub$ 
    until  $G_p = \{\}$ 
  return
end

```

### 7.3.3 Algoritmo SubdueCL

- El algoritmo SubdueCL se muestra en las tablas ?? y ??.
- La función principal toma como parámetros los ejemplos positivos  $G_p$ , los ejemplos negativos  $G_n$ , el tamaño del beam (porque el algoritmo de SubdueCL utiliza una búsqueda beam), y un límite *limit* sobre el número de subestructuras a incluir en su búsqueda.
- La función principal hace llamadas a la función SubdueCL para formar la hipótesis  $H$  que describa los ejemplos positivos.
- Cada vez que se hace una llamada a la función SubdueCL se añade una subestructura a  $H$ .
- En el caso en que SubdueCL regresa *NULL*, el *Beam* se incrementa en 10%, de tal modo que SubdueCL pueda explorar un espacio de búsqueda más amplio.

Tabla 7.5: Algoritmo SubdueCL.

```

SubdueCL( $G_p, G_n, Limit, Beam$ )
   $ParentList = (AllsubstructuresofonevertexinG_p)modBeam$ 
  Repeat
     $BestList = \{\}$ 
     $Exhausted = TRUE$ 
     $i = Limit$ 
    while ( $(i > 0)$  and ( $ParentList \neq \{\}$ ))
       $ChildList = \{\}$ 
      foreach substructure in  $ParentList$ 
         $C = \mathbf{Expand}(Substructure)$ 
        Evaluate( $C, G_p, G_n$ )
        if CoversOnePos( $C, G_p$ )
          then  $BestList = BestList \cup C$ 
           $ChildList = (ChildList \cup C)modBeam$ 
         $i = i - 1$ 
      endfor
       $ParentList = ChildListmodBeam$ 
    endwhile
    if  $BestList = \{\}$  and  $ParentList \neq \{\}$ 
      then  $Exhausted = FALSE$ 
       $Limit = Limit * 1.2$ 
    until( $Exhausted = TRUE$ )
    return  $first(BestList)$ 
end

```

- Elegimos hacer incrementos del *beam* de 10% porque ese valor fue suficiente para encontrar una subestructura en la siguiente iteración para la mayoría de los experimentos.
- Además, después de que *SubdueCL* encuentra una subestructura, los ejemplos positivos cubiertos por ella se eliminan del grafo positivo.
- En la tabla ?? se muestra la función *SubdueCL*, la cual empieza a construir una *ParentList* creando una subestructura para cada vértice en el grafo con una etiqueta diferente, pero manteniendo solo tantas subestructuras como lo permita el tamaño del *Beam*.
- El operador “*mod Beam*” significa que las listas contienen tantas subestructuras como el tamaño del *Beam*.
- Posteriormente se expande cada una de las subestructuras en la lista *ParentList* con un arco o un vértice y un arco en todos los modos posibles y se evalúa de acuerdo a la ecuación presentada anteriormente.
- Aquellas subestructuras que cubran al menos un ejemplo positivo y caen dentro de los límites del tamaño del *Beam* se quedan en la lista *BestList*.
- La lista *ChildList* mantiene todas las subestructuras que fueron obtenidas de la expansión de las subestructuras de la lista *ParentList* y también está restringida por el tamaño del *Beam*.
- El parámetro *Limit* se utiliza para expandir tantas subestructuras como su valor, pero si la lista *BestList* está vacía después de expandir tantas subestructuras como el valor de *Limit* de la lista *ParentList*, entonces *Limit* se incrementa en 20% hasta que se encuentre una.
- Elegimos un valor de incremento del límite de 20% porque usualmente era suficiente para encontrar una subestructura positiva en el siguiente intento para nuestros experimentos.
- Finalmente la función **SubdueCL** regresa lo mejor de la lista *BestList* conteniendo todas las subestructuras que cubren al menos un ejemplo positivo.
- Es importante mencionar que todas las listas están ordenadas de acuerdo al valor de evaluación de las subestructuras.