Unidad 2: Minería de elementos frecuentes

Algoritmos de árboles de enumeración

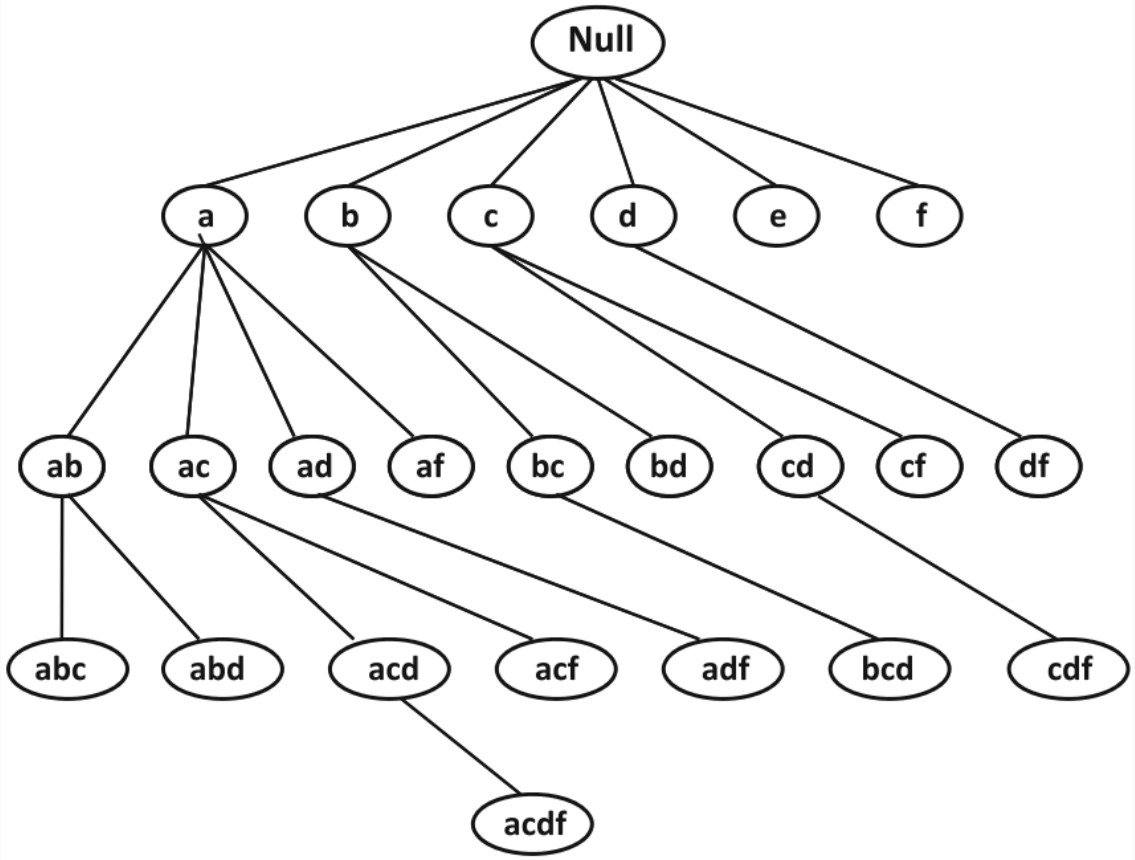
Blanca Vázquez y Gibran Fuentes-Pineda

Octubre 2020

Algoritmos de árboles de enumeración

- Es un algoritmo de minería de elementos frecuentes
- Los conjuntos de elementos candidatos se generan en una estructura de árbol, conocida como árboles de enumeración.
- El crecimiento del árbol puede compensar diferentes estrategias entre almacenamiento, acceso a disco y eficiencia computacional.

Ejemplo de un algoritmo de árboles de enumeración



A esta estructura, también se le conoce como: árbol lexicográfico ($a < b, b < c...$)

Características

- Los patrones candidatos son generados a medida que crece el árbol lexicográfico.
- Proporciona una representación abstracta jerárquica de los conjuntos de elementos.
- Esta representación, es aprovechada para explorar de forma sistemática los patrones candidatos (evitando repeticiones).
- La salida del algoritmo son los conjuntos de elementos frecuentes definidos en la estructura de árbol.

Definición

El algoritmo de árbol de enumeración se define sobre los conjuntos de elementos frecuentes de la siguiente manera:

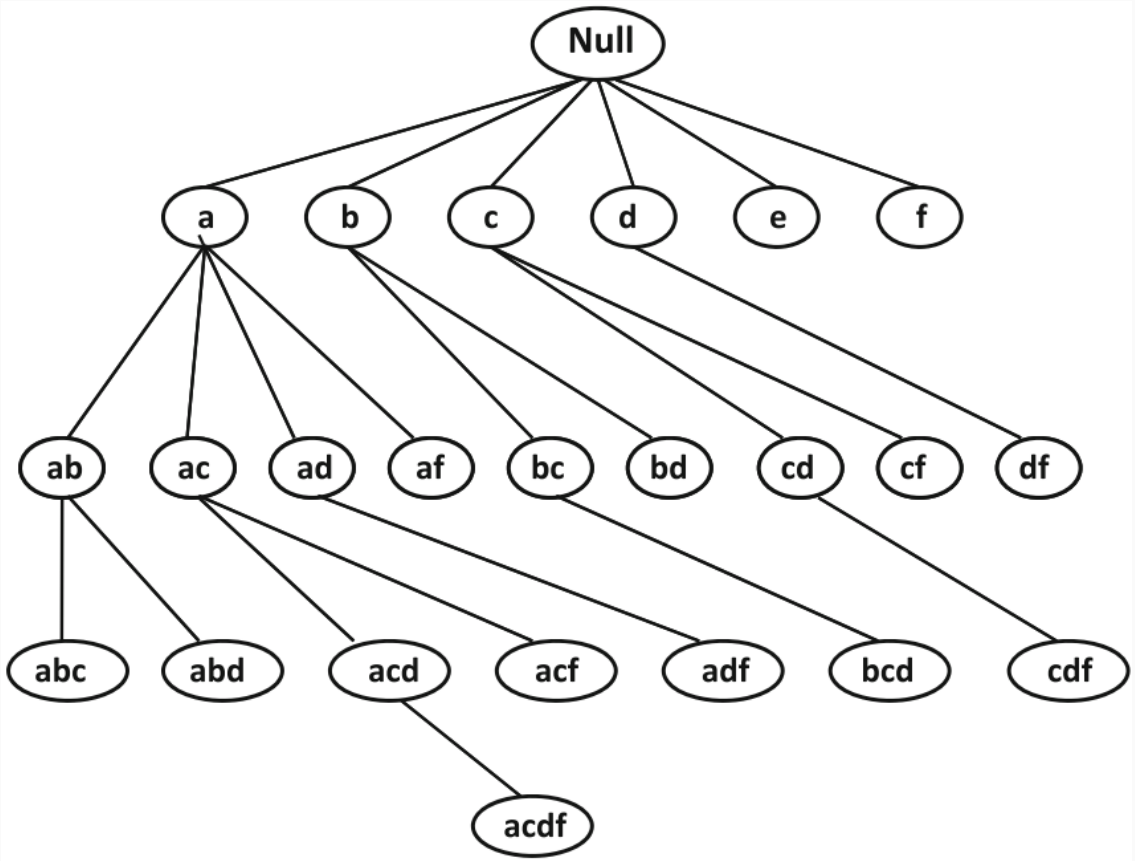
- Existe un nodo en el árbol correspondiente a cada conjunto de elementos frecuentes.
- La raíz del árbol corresponde al conjunto de elementos vacío.

Definición

El algoritmo de árbol de enumeración se define sobre los conjuntos de elementos frecuentes de la siguiente manera:

- Sea $I = \{i_1, ..., i_k\}$ un conjunto de elementos frecuentes, donde $i_1, i_2 ..., i_k$ son listados en orden lexicográfico
  - El padre del nodo I es el conjunto $\{i_1, ..., i_{k-1}\}$, por lo tanto el hijo de un nodo solamente se puede extender con elementos que aparecen lexicográficamente después de todos los elementos que ocurren en ese nodo.

Ejemplo de algoritmo de árboles de enumeración



El orden lexicográfico impone una estructura estrictamente jerárquica sobre los conjuntos de elementos.

¿Cómo crecen los árboles de enumeración?

Dependerá del algoritmo selección.

- Los algoritmos indicarán el orden y la estructura para el descubrimiento de conjuntos de elementos frecuentes
- Una extensión frecuente del árbol es: un elemento que se usa para extender un nodo.

De forma general:

- El nodo de un árbol se extiende al encontrar los elementos frecuentes en 1 (nodos candidatos).
- Los nodos candidatos, son extendidos al encontrar los elementos frecuentes en 1
- y así sucesivamente.

¿Cómo generar nodos candidatos a partir de los nodos frecuentes?

- Para que un elemento i sea considerado un nodo candidato para extender un nodo frecuente P, entonces i debe ser una extensión frecuente del padre Q de P.
  - Esto se debe a la propiedad de cierre descendente: Para definir los nodos candidatos de un nodo P, primero se deben definir los nodos frecuentes de su padre Q

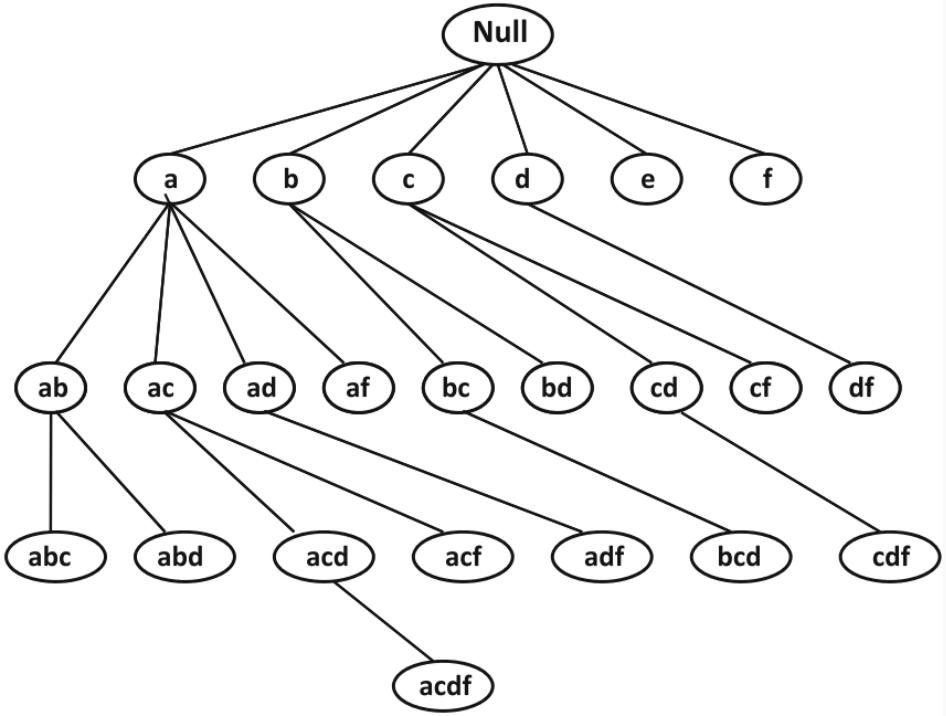
Algoritmo de árbol de enumeración genérico (parte 1)

- Seleccionamos uno o más nodo P en ET
- Determinamos las extensiones candidatas $C(P)$ para cada nodo P
- Calculamos el soporte de los candidatos
- Añadimos los elementos frecuentes a ET

Algoritmo de árbol de enumeración genérico (parte 2)

función GET($\mathcal{T}$, *minsup*)
    inicializamos el *ET* con un nodo *nulo*
    mientras cualquier nodo en *ET* no haya sido examinado **hacer**
        selecciona uno o más nodos P no examinados del *ET*
        genera extensiones candidatas $C(P)$ para cada nodo P
        determina las extensiones frecuentes $F(P) \subseteq C(P)$ para cada P con soporte
        extiende cada nodo P en *ET* con sus extensiones frecuentes en $F(P)$
    fin de mientras
    devolver *ET*
fin de función

Ejemplo de un algoritmo de árboles de enumeración



- El valor de $F(P)$ cuando $P = ab$ es {c,d}
- El valor de $C(P)$ cuando $P = ab$ es {c,d,f}
- La relación $F(P) \subseteq C(P) \subset F(Q)$ siempre se cumple

Consideraciones *ET*

- El algoritmo termina hasta que ninguno de los nodos pueda extenderse más.
- La mayoría de los algoritmos de minería de patrones pueden ser vistos como extensiones del *ET*
- Existen diferentes estrategias para el crecimiento del árbol y metodologías de conteo:
  - Eficiencia, requerimientos, costos y acceso a disco.

Estrategias de crecimiento

Estrategia de primero amplitud	Estrategia de primero profundidad
El conjunto de nodos P en una iteración, corresponde a todos los nodos en el 1er nivel del árbol.	Selecciona un solo nodo en el nivel más profundo para crear P.
Uso: en bases de datos residentes en disco, debido a que todos los nodos en un solo nivel del árbol pueden ser extendidos durante una pasada en la base de datos.	Uso: descubrimiento de patrones, debido a que la estrategia tiene una mejor capacidad de explorar el árbol en profundidad.

Descubrimiento de patrones

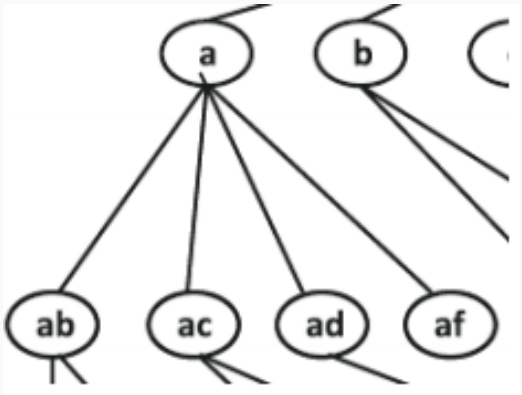
- El descubrimiento de patrones de manera temprana es útil para la eficiencia computacional en la minería de patrones.
- También la estrategia de primero profundidad ayuda en la gestión de la memoria de algoritmos basados en proyecciones.

En cuanto a las estrategias de conteo, estas tienden a ser costosas, por lo tanto, las estrategias de crecimiento tratan de optimizar el conteo mientras recorren el árbol.

Interpretación del algoritmo apriori basado en *ET*

- El algoritmo apriori construye el *ET* usando la estrategia "primero amplitud"
- Apriori genera los conjuntos candidatos $(k+1)$ mediante la fusión de 2 elementos frecuentes de los cuales los primeros $k-1$ elementos son los mismos.

Interpretación del algoritmo apriori basado en *ET*



Ejemplo: la extensión de {ab}, se obtiene con {c,d,f,} $\subseteq$ F(a)

Recordemos que: las $C(P)$ para todos los nodos P en un nivel del *ET*, pueden generarse de manera exhaustiva y no repetidas mediante combinaciones entre todos los pares de hermanos en el mismo nivel.

¿Y cómo se poda el árbol?

- El algoritmo apriori descarta algunos nodos del árbol cuando no garantizan ser elementos frecuentes.
- El algoritmo termina cuando el árbol no puede crecer más.

Proyección del árbol (TreeProjection)

- Es una familia de métodos que usa proyecciones recursivas para la construcción del *ET*
- Puede emplear estrategias de amplitud primero, profundidad primero o la combinación de ambas.
- La estrategia usada para la selección del nodo *P* define el orden en el cual los nodos serán extendidos.

Consideraciones en el algoritmo de TreeProjection

- Para indicar la DB proyectada en el nodo P se usa $T(P)$
- Cada nodo en el ET es ahora representado por el par de elementos de la base de datos (DB) proyectada
- Ahora $T(P)$ será más pequeña que la DB original
- La idea general del $T(P)$ es describir la porción de la DB con la información más relevante, a través de eliminar las transacciones y elementos irrelevantes realizadas en los niveles superiores del árbol.

Algoritmo TreeProjection

función ProjectedEnumerationTree($\mathcal{T}$, *minsup*)
  inicializar el *ET* con un nodo raíz $(Null, T)$
  mientras cualquier nodo en *ET* no haya sido examinado **hacer**
    selecciona uno o más nodos $(P, T(P))$ no examinados del *ET*;
    genera extensiones candidatas $C(P)$ para el nodo $(P, T(P))$;
    determina las extensiones frecuentes $F(P) \subseteq C(P)$ mediante el conteo de los elementos individuales en pequeñas DB proyectadas $T(P)$;
    remueve los elementos no frecuente en $T(P)$;
    para cada elemento frecuente $i \in F(P)$ **hacer**
      genera $T(P \cup i)$ a partir de $T(P)$;
      añade $(P \cup i, T(P \cup i))$ como hijo de P en *ET*;
    fin de para
  fin de mientras
  devolver *ET*
fin de función

Método de conteo vertical

- A este método se le conoce como: **intersección recursiva de listas**
- Fue introducida por los algoritmos de *Monet y Partition*.
- Cada elemento está asociado con una lista de sus identificadores de transacciones *tids*
- De forma general, este método es una transpuesta de una matriz binaria

Ejemplo

TID	Elementos	Representación binaria
1	{pan, mantequilla, leche}	110010
2	{huevos, leche, yogurt}	000111
3	{pan, queso, huevos, leche}	101110
4	{huevos, leche, yogurt}	000111
5	{queso, leche, yogurt}	001011

- El $sup(\{leche, yogurt\}) = 3/5 = 0.6$
- El $sup(\{leche, huevos, yogurt\}) = 2/5 = 0.4$

Representación vertical

Elemento	Conjunto de tids	Representación binaria
Pan	{1,3}	10100
Mantequilla	{1}	10000
Queso	{3,5}	00101
Huevos	{2,3,4}	01110
Leche	{1,2,3,4,5}	11111
Yogurt	{2,4,5}	01011

Para calcular el soporte, se hace la intersección de *tids*

- El $sup(\{pan, huevos\}) =$
- El $sup(\{huevos, leche, queso\}) =$
- El $sup(\{queso, leche\}) =$

Consideraciones

- El método de representación vertical es más eficiente al momento de buscar elementos frecuentes, sin embargo, requiere mucha memoria debido a que va almacenando las listas de *tids*.
- Para solucionar este problema, se ha propuesto dividir la DB en pequeños fragmentos los cuales se procesan de forma independiente (*partitioned ensemble*)

Consideraciones

- El método de representación vertical es más eficiente al momento de buscar elementos frecuentes, sin embargo, requiere mucha memoria debido a que va almacenando las listas de *tids*.
- Para solucionar este problema, se ha propuesto dividir la DB en pequeños fragmentos los cuales se procesan de forma independiente (*partitionedensemble*)

Método recursivo de crecimiento de patrones basados en sufijos

- Es un caso especial del algoritmo de árboles de enumeración
- Un ejemplo, son los árboles de patrones frecuentes

Árboles de patrones frecuentes

- El árbol de patrones frecuentes (Frequent-Pattern tree, FP-tree) se basa en la metodología *divide y vencerás*
- Evita generar conjuntos de elementos candidatos para maximizar el uso de recursos.
- Busca comprimir una base de datos hacia un FP-tree

Ejemplo

TID	Elementos
1	{f, a, c, d, g, i, m, p}
2	{a, b, c, f, l, m, o}
3	{b, f, h, j, o}
4	{b, c, k, s, p}
5	{a, f, c, e, l, p, m, n}

Paso 1: Escanear la BD, y calcular la frecuencia de cada elemento

Paso 2: Ordenar los elementos por frecuencia (descendente) y que cumpla con un $minSup = 0.5$

Ordenamos cada elemento frecuente de forma descendente indicando su frecuencia

Elemento	Frecuencia
f	4
c	4
a	3
b	3
m	3
p	3

Ejemplo

Paso 3: Por cada transacción, escribimos los elementos frecuentes que cumplen con el *minSup* ordenamos de forma descendente

TID	Elementos	Elementos frecuentes
1	{f, a, c, d, g, i, m, p}	{f, c, a, m, p}
2	{a, b, c, f, l, m, o}	{f, c, a, b, m}
3	{b, f, h, j, o}	{f, b}
4	{b, c, k, s, p}	{c, b, p}
5	{a, f, c, e, l, p, m, n}	{f, c, a, m, p}

Ejemplo

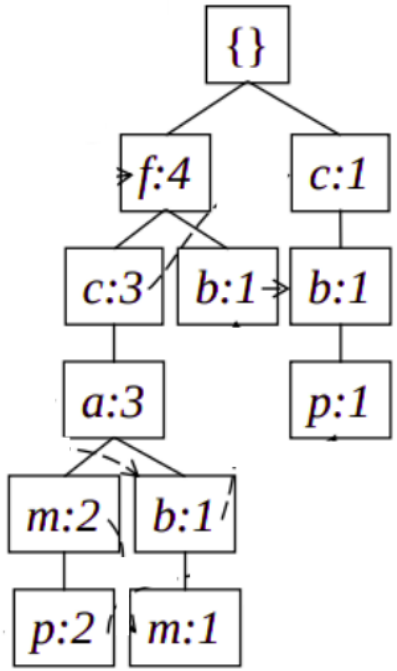
Paso 4: por cada transacción se construye una rama en el FP-tree

- Si existe una ruta con prefijo común:
  Incrementar la frecuencia de los nodos en esta ruta y agregar el sufijo
- En caso contrario crear una nueva rama.

Resultado

TID	Elementos	Elementos frecuentes
1	{f, a, c, d, g, i, m, p}	{f, c, a, m, p}
2	{a, b, c, f, l, m, o}	{f, c, a, b, m}
3	{b, f, h, j, o}	{f, b}
4	{b, c, k, s, p}	{c, b, p}
5	{a, f, c, e, l, p, m, n}	{f, c, a, m, p}

Elemento	Frecuencia
f	4
c	4
a	3
b	3
m	3
p	3



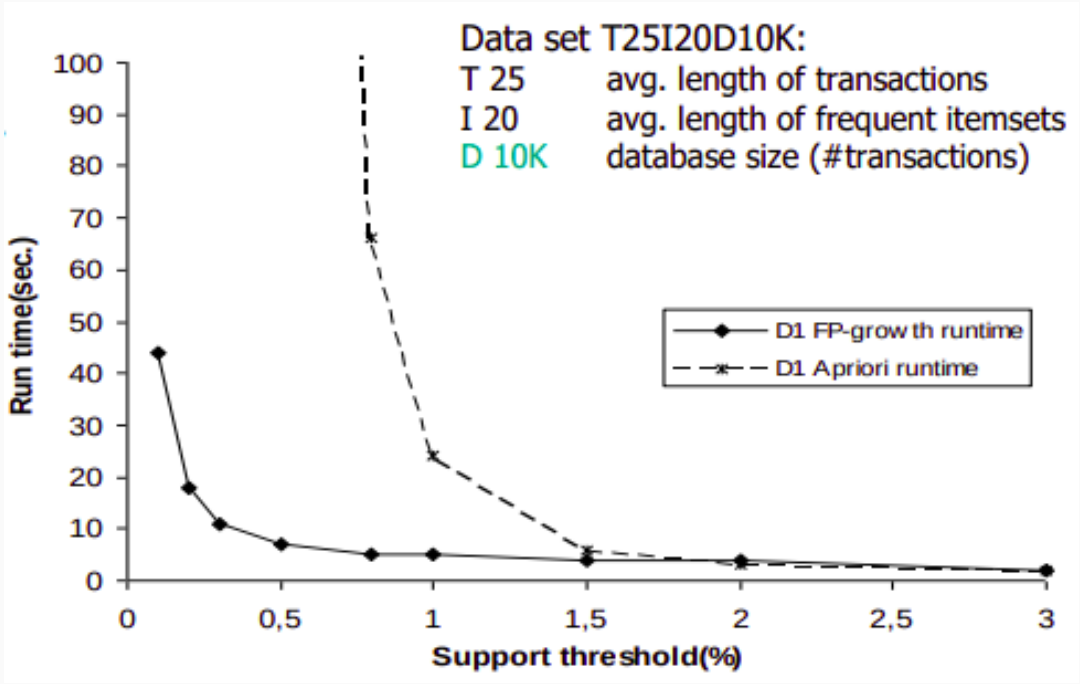
Construcción de un FP-tree a partir de una BD de transacciones

Ventajas del FP-Tree

- No rompe un patrón largo en una transacción
- Ignora elementos no frecuentes
- orden descendente de frecuencia: es más probable que se compartan los elementos más frecuentes
- El FP-tree nunca será más grande que la BD original
- FP-Tree es frecuentemente usado en la minería de patrones frecuentes (divide y vencerás)

Comparativo

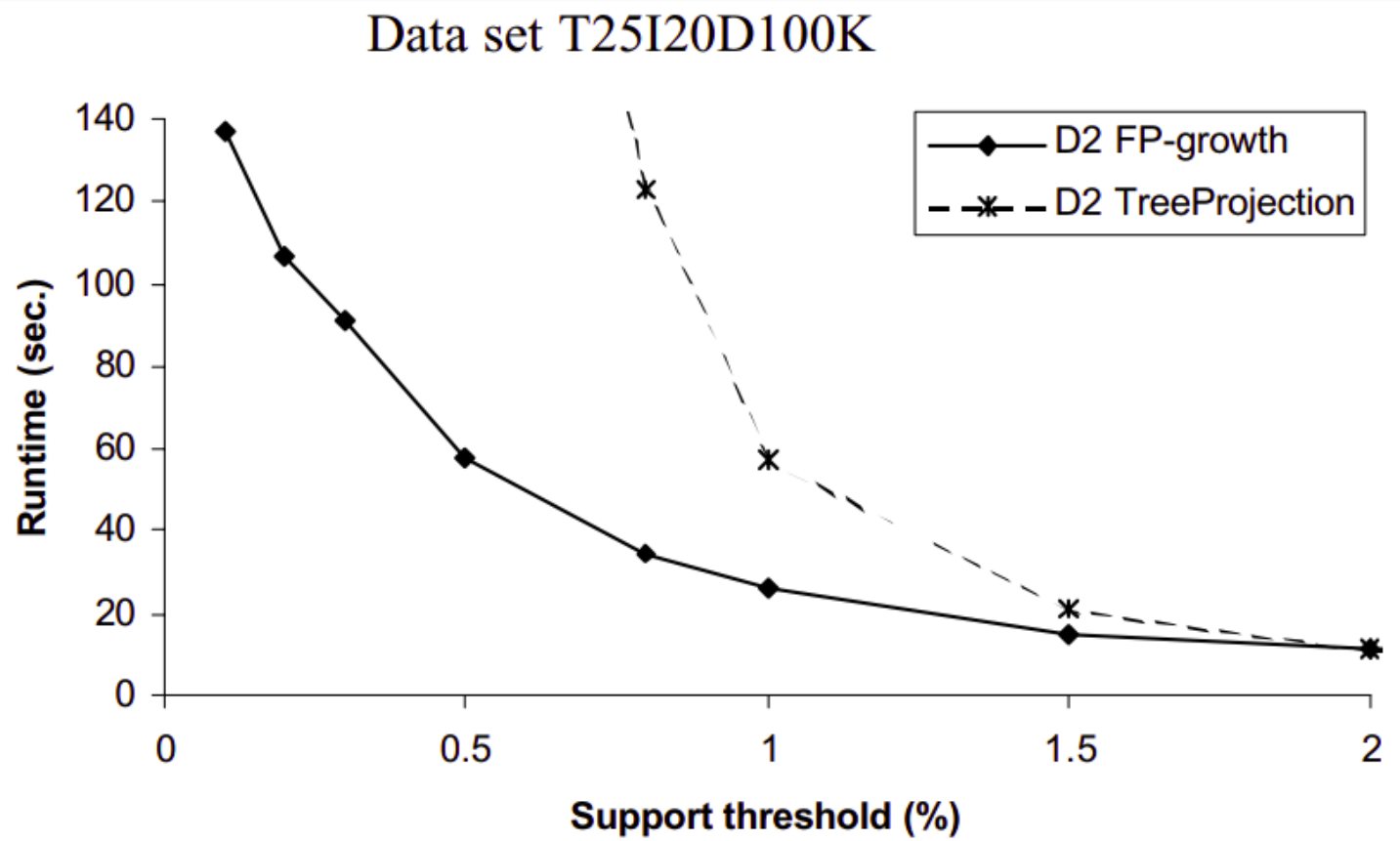
En el 2000, Han, Pei & Yin realizaron un estudio para comparar el rendimiento del algoritmo Apriori vs el FP-Tree



FP es más rápido que Apriori, incluso más rápido que TreeProjection.

Comparativo

En el 2000, Han, Pei & Yin realizaron un estudio para comparar el rendimiento del algoritmo Apriori vs el FP-Tree



FP es más rápido que Apriori, incluso más rápido que TreeProjection.

Comparativo

- El algoritmo FP no genera elementos candidatos
- El Algoritmo apriori va avanzando en primero amplitud
- FP usa una estructura compacta
- En FP, al escanear la BD, elimina elementos repetidos
- En FP, las operaciones básicas son conteo y la construcción del árbol.