Entrega 2 - Proyecto Final

Lluch, Pablo
pablo.lluch@gmail.com

Fuentes, Tomás
tafuentesc@gmail.com

Teles, Nuno
nuno_teles3@hotmail.com

6 de diciembre de 2013

1. Introducción

A continuación se presenta el Informe Final, que tiene como principal objetivo, implementar y evaluar el algoritmo que elegimos en la entrega intermedia – *Relim* (Recursive Eliminatión Algorithm). Este algoritmo está directamente relacionado con el problema de encontrar *k frecuent items sets*, en grandes volumen de datos.

La elección del algoritmo resultó de un proceso evolutivo en que valoramos distintas soluciones propuestas por distintos investigadores, para decidir cuál sería el algoritmo que mejor serviría nuestras pretensiones. Pretendíamos implementar un algoritmo que involucrase una estrategia eficiente para obtención de los ítem-sets, aún que al mismo tiempo, no complicase deberás su implementación. Sin embargo, tenemos la noción que ni siempre los algoritmos que producen mejores resultados son los más fáciles de implementar. El algoritmo FP-Growth es un buen ejemplo que comprobar esta afirmación.

Hay diversos criterios que podemos utilizar para evaluar a priori lo cuan completo es un algoritmo, por ejemplo, si consomé o no poca memoria o si tiene un tiempo de ejecución razonable. Estos dos criterios pueden ser influenciados positivamente o negativamente, dependiendo del uso que damos a las estructuras de datos usadas para almacenar y manipular las transacciones.

Por eso, para esta entrega, optamos por trabajar con el algoritmo *Relim*, esencialmente porque emplea una estructura de datos bastante simples (Lista en que cada posición guarda una lista de transacciones) para guardar las transacciones, como se podrá demonstrar el la descripción detallada del algoritmo, más adelante. Igualmente, la forma como este algoritmo procesa las transacciones (a través de eliminación recursiva) hace que esta idea se constituye una opción válida y su implementación en Python sea fácil.

Durante la descripción detallada del algoritmo, iremos señalar las modificaciones que hicimos faje al solución proyectada inicialmente (en el paper) y explicar cómo la adopción de estas, nos ayudaran a implementar eficazmente el algoritmo.

Por fin, se hará un análisis experimental, confrontando el algoritmo en distintos tipos de bases de datos y se extrapolarán conclusiones que podrán ter

como fundamento otras ideas presentes en anteriores algoritmos revisados. Se evaluó nuestra solución sobre distintos tipos bases de datos: esparzas (bajo fracción ítems presentes en cada transacción) y densas (alta fracción de ítems presentes en cada transacción).

Este informe se presentará en diferentes secciones, primero, con la identificación del problema (asociado al concepto de ítem-sets frecuente) y posteriormente para cada uno de los algoritmos revisados (estudiados en la parte intermedia), se hace una descripción general del método y se identifica cuál es su objetivo así como, ventajas competitivas y problemas que generalmente aportan de su modelo y le están asociados. Por fin, se describirá con mayor detalle en modelo que utilizaremos en el proyecto de Minería de Datos – Algoritmo Relim y su análisis experimental.

2. Problema

El problema de encontrar k itemsets frecuentes en grandes volúmenes de información ha sido un importante foco de investigación a lo largo de los últimos años, durante los cuales se han propuesto distintas implementaciones y algoritmos que buscan solucionar este problema, algunos de los cuales son puramente optimizaciones de otros algoritmos más tradicionales.

Esta técnica para minar datos es un método de análisis bastante conocido que busca encontrar correlaciones y tendencias en los datos para después ayudar a las empresas en la toma de decisiones y optimización de sus procesos.

Dada una base de datos con N transacciones, cada una con un conjunto de ítems, el proceso de encontrar los k- itemsets frecuentes en una base de datos se resume a identificar todos los itemsets en que el soporte de éste sea mayor a un umbral mínimo dado por el usuario.

3. SAM

3.1. Objetivo

El objetivo del Algoritmo SAM (Split and Merge) es encontrar todos los posibles itemsets frecuentes dada una base de datos y un umbral positivo. Al

contrario de algunos algoritmos basados en A-priori, este algoritmo usa un modelo distinto para procesar los datos consistente una busca en profundidad a través una estrategia comúnmente observada en algoritmos computacionales - dividir para conquistar. Así, para un determinado ítem a considerar, cada sub-problema consiste en encontrar todos los itemsets que contienen al ítem usando para eso un conjunto de transacciones más pequeño también designado de base de datos condicional (conjunto de transacciones que contiene el item).Todos los itemsets que no contienen el ítem a considerar constituyen también un sub-problema válido y por lo tanto, una base de datos condicional en la cual buscamos una solución. La gracia del algoritmo es usar esta idea e iterar recursivamente sobre cada un de estos sub-problemas y encontrar todos los itemset frecuentes.

3.2. Descripción

El SAM es un de los algoritmos que pertenece a la categoría de los algoritmos de eliminación recursiva que son utilizados para obtener conjuntos de itemsets frecuentes. Muy idéntico al algoritmo RELIM, ambos se basan en el mismo principio al momento procesar los itemsets. A diferencia de RELIM que usa una representación parcial vertical de los datos, SAM usa un único arreglo que mantiene en cada posición todas las listas de ítems (transacciones). Este arreglo es procesado al inicio del algoritmo usando dos operaciones Split and Merge, operaciones que tratan de construir sucesivamente las base de datos condicionales que son usadas en la recursión.

Para construir el arreglo que va servir de input al algoritmo es necesario pre-procesar las transacciones inicialmente y por eso, se procede a un conjunto de pasos. En primer lugar se determinan las frecuencias de cada uno de los ítems individualmente, de forma eliminar los ítems que son infrecuentes. Después de este paso, se ordenan cada un de los ítem presentes en cada transacción de forma creciente para poder minimizar su tiempo de ejecución. Por fin, se modifica el orden de presentación de las transacciones en la base de datos siguiendo un orden lexicográfico descendente. La estructura de datos que guarda esta información se construye a través de un arreglo que tiene para cada posición cada transacción, respectando el orden obtenido anteriormente y un contador que representa el numero de ocurrencias de esa misma transacción.

Con la ayuda de la operación de Split, en cada paso de la recursión conseguimos expandir el prefijo de cada base de datos condicional. A esto se debe

que esta operación permite una separación del ítem líder de sus restantes elementos construyendo un sub-problema de menor dimensión que consiste en las transacciones que contienen el ítem líder pero en el que este ítem fue removido. Este sub-problema es procesado independientemente sobre un nuevo arreglo de forma recursiva. Así se consigue encontrar todos los itemset frecuentes que contienen el ítem líder bastando para eso, preguntar en cada paso se el ítem de separación es frecuente. Su suporte es computado sumando todos los suportes de las transacciones que lo contienen. Por otro lado, se usa el arreglo obtenido en el paso anterior (de separación) para construir un nuevo arreglo que representa las transacciones que no contienen el ítem. Este nuevo arreglo resulta de la Operación Merge responsable por combinar el arreglo obtenido con la Operación Split y el resto del arreglo original que como sabemos no fue separado y por lo tanto es necesario para descubrir todos los ítem sets frecuentes que no contienen el ítem líder. Esta operación tiene como principal objetivo combinar posibles transacciones que tengan el mismo ítem líder y agrupar transacciones iguales reduciendo el numero de objetos a procesar. Este arreglo es después procesado recursivamente.

3.3. Ventajas

- Su estructura de datos es bastante simples porque engloba apenas un arreglo para almacenar la información a procesar y por eso, hace que el algoritmo pueda ser utilizado con almacenamiento externo (por ejemplo tabla) evitando casos que el algoritmo no pueda funcionar porque no existe espacio para cargarlo en memoria principal.

- Además, durante las Operaciones de Split y Merge en cada paso de recursión, apenas es necesario copiar el contador de ocurrencias de transacciones y el puntero incrementado para un nuevo arreglo lo que hace que en final sea necesario una única copia de las transacciones. Por fin, hay que notar el bajo consumo memoria visto que en cada paso de la recursión sólo es necesario alocar un único arreglo que resulta de la Operación Split del arreglo pasado como input ya que este arreglo puede ser utilizado para separar el próximo ítem cuando el algoritmo fuer ejecutado nuevamente en la recursión.

3.4. Desventajas

- Puede acontecer que la operación de merge atrase el algoritmo cuando las listas de transacciones presenten tamaños diferentes.

4. BitTable

4.1. Objetivo

El objetivo de este algoritmo consiste en optimizar el uso de memoria y tiempo en el que caen los algoritmos basados en A-Priori. Para lo anterior, los autores proponen una nueva estructura de datos conocida como BitTableFI, la cual permite representar la base de datos como una representación vertical comprimida de la BD, en la que cada registro asociado a un determinado ítem es un arreglo de bits en el que una entrada es 1 si el ítem está en dicha tupla ó 0 en caso contrario.

Gracias al uso de esta estructura se logró un considerable ahorro en memoria a la vez que se mejoró el tiempo de ejecución del algoritmo, mayormente gracias a la velocidad propia de las operaciones bitwise en comparación con sus equivalentes en la representación original.

4.2. Descripción

En el año 2007, Dong & Han propusieron un nuevo algoritmo para extraer itemsets frecuentes llamado BitTableFI. Más allá del algoritmo utilizado, el gran aporte de su trabajo son las estructuras de datos que utilizan para comprimir la base de datos y los itemsets frecuentes en cadenas de bits. El algoritmo se puede separar en 4 grandes pasos: primero, se identifican los items frecuentes, descartando los que no cumplan con el soporte mínimo (L_1). Posterior a esto se comprime la base de datos considerando sólo estos items frecuentes, obteniendo una representación vertical comprimida de ésta conocida como BitTable. En tercer lugar, se obtienen los itemsets frecuentes en el cual cada set está compuesto por 2 items frecuentes (L_2). Finalmente, el cuarto paso es un proceso iterativo el cual se puede subdividir en 2 sub etapas: generación de candidatos y cálculo de soporte. Este último paso se repite hasta que no se puedan obtener más candidatos.

Para comprimir la base de datos y transformarla en la BitTable, ésta se debe comprimir tanto de forma vertical como horizontal. Para realizar lo anterior, primero codificamos cada tupla de la tabla como un arreglo de bits, donde el valor de la i-ésima posición es 1 si la tupla contiene al ítem i ó 0 en caso contrario. Una vez hecho esto, construimos la BitTable de la BD la

cual poseé una entrada por cada ítem frequente donde el valor de la i-ésima entrada está dado por el arreglo de bits formado al considerar los bits de todas las tuplas en la columna del i-ésimo ítem frequente.

Una vez obtenida la BitTable, obtenemos el itemset frecuente en el que cada itemset posee exactamente 2 items frecuentes (L_2). Posterior a esto, cada itemset es representado como un arreglo de bits de largo igual a la cantidad de items frecuentes en L_1, donde el i-ésimo bit es 1 si el i-ésimo ítem es parte de éste.

Una vez obtenidas la BitTable y L_2 procedemos a calcular L_k, en el cual cada itemset está compuesto por k items. El cálculo de L_k se realiza de la siguiente forma: para cada itemset en L_{k-1}, se toma el primero y se cambia el último 1 de su representación por un 0 (A esta representación intermedia se le llama MID). Posterior a esto, se hace un and bitwise (&) entre el MID y el siguiente itemset después de éste. Si el valor retornado es MID, entonces ambas representaciones tienen un item de diferencia, por lo que las mezclamos y lo agregamos a L_k; en caso contrario, se descarta. Este proceso se repite para todos los itemsets que están después del itemset con el cual se contruyó el MID. Una vez comparados con todos, se toma el itemset que viene después del MID, se saca el MID de éste y se repite el proceso hasta haber probado con todos los itemsets de L_{k-1}.

Una vez obtenido L_k se calcula su soporte para cada itemset dentro de éste realizando un and bitwise (&) entre las columnas de la BitTable asociadas a los ítems dentro de ésta y contando la cantidad de 1's en el resultado. Hecho esto, se eliminan los itemsets que no superen el soporte mínimo establecido. Si la cantidad de itemsets en L_k al final de la poda es mayor a cero, se continúa con L_{k+1}; en caso contrario, el algoritmo se detiene ahí. En la Figura 1 se puede ver el pseudo código del algoritmo.

4.3. Ventajas

1. Ahorro en memoria: Gracias a la compactación de la base de datos en la forma de la BitTable se disminuye considerablemente el tamaño de la base de datos en disco (se habla de disminuciones del orden de 1/160 [2]).

```
BitTableFI(Database D,int MinSup)
{
    L1 = { frequent 1-itemsets};
    BitTable[] db = database_bittable_init(D,L1);
    L2=getl2froml1(L1);
    int k=2;
    while (Lk.NotEmpty())
    {
      k++;
      Ck = quick_candidateitemsets_generation(Lk-1);
      Lk = quick_candidateitemsets_support_count(Ck, db, MinSup);
    }
}
```

Figura 1: BitTable pseudo-algorithm

2. Ahorro en tiempo: Gracias al menor tamaño de la BD se pueden cargar segmentos más grandes en memoria, lo cual disminuye los accesos a disco. Adicionalmente, el uso de operaciones bitwise para determinar los itemsets candidatos y calcular el soporte de éstos resulta mucho más eficiente que las típicas operaciones de conteo realizadas sobre la representación original, lo cual significa un sustancial boost en el rendimiento del algoritmo.

4.4. Desventajas

1. El algoritmo propuesto es, en realidad, el mismo algoritmo A-priori. La única diferencia real es la estructura de datos utilizada, la cual es responsable de la mejora en rendimiento obtenida. Así, es fácil deducir que este algoritmo no puede rendir mejor que aquellos de la línea de FP-Growth, los cuales son más rápidos puesto que solo traversan la base de datos 2 veces.

2. El uso de la BitTable puede ser difícil con otros algoritmos que ya tengan una estructura de datos propia tales como FIUT [*] y FP-Growth.

5. GarNet

5.1. Objetivo

Abordar mediante métodos de inteligencia artificial el problema de encontrar asosiaciones cuantitativas, que tratan con atributos continuos. La idea es evitar discretizaciones que generen pérdida de información y sean complejas de calibrar.

5.2. Descripción

El objetivo de GarNet [1] consiste en lograr encontrar asosiaciones entre genes para armar una red de genes, la cual consiste en un grafo cuyos nodos son determinados genes y sus vínculos corresponden a asociaciones entre ellos.

Para trabajar con información acerca de los genes, se utiliza comúnmente tecnología basada en *microarrays*, los cuales permiten obtener genotipos,

medir niveles de expresión de genes y otros fines relacionados. Particularmente, interesa saber qué genes están interactuan con otros, en el sentido de que un gen puede codificar una proteína o ARN que interactúa con el producto de otro. Para esto se puede formar una red de regulación génica, que es un grafo que establece estas relaciones.

Este es un problema que puede ser potencialmente atacado usando reglas de asociación. Sin embargo, una de las dificultas es que la información que se obtiene de los *microarrays* son atributos continuos que contienen información de la concentración de nucleótidos expresados en moles. Para atacar este tipo de problemas, una opción es discretizar el valor de las variables. Sin embargo, escoger una buena discretización puede ser complejo, sobre todo si además los rangos de valores para cada atributo son muy distintos. GarNet [1] pretende atacar este problema encontrando reglas de asociaaciones cuantitativas (o QAR), que trabajan con intervalores continuos. Las asociaciones, en este caso, vienen a ser del tipo $(A \in [a, b], B \in [c, d] => C \in [e, f])$

De modo general, GarNet es un algoritmo para encontrar asociaciones entre genes basado en el algoritmo genético NSGA II [4]. NSGA II es un algoritmo genético eficiente que permite trabajar con múltiples objetivos. En general, los algoritmos genéticos permiten encontrar soluciones a problemas que no tienen un planteamiento matemático simple, y que usa búsquedas estocásticas para analizar un espacio de búsqueda amplio, como es el caso de este problema.

Para este problema en particular, los autores de [1] crea una población de individuos que son equivalentes cada uno a una regla de asociación posible. Si tenemos un total de n atributos, un cromosoma consistirá en dos partes de largo n cada una: La primera indica el intervalo con el que estamos trabajando para cada atributo, mientras que la segunda parte consiste en n números que pueden tomar los valores 0, 1 y 2. Si el valor es 0, implica que el atributo asociado no participa de la regla de asociación. Si el número es 1, entonces el (o los) atributos forman parte del antecedente de la regla de asociación. Finalmente, si el número es 2, entonces el atributo forma parte del consecuente de la regla. Como ejemplo, si tenemos el individuo de la Fig. 1, equivaldría a tener la regla de asociación $CLB1 \in [-0{,}68, 0{,}05], CLN2 \in [-1{,}11, 0{,}00] => CLB5 \in [-0{,}85, 0{,}10]$

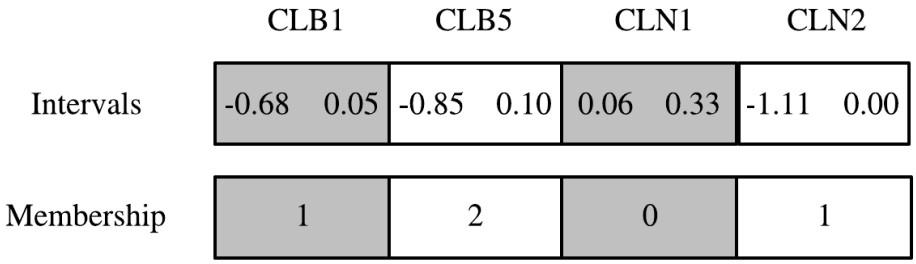

Figura 2: Cromosoma

Para diseñar un algoritmo genético, se necesitan definir:

1. Operador de Crossover - permite generar mezclas entre los distintos individuos. Aquí, por ejemplo, se puede intercambiar la posición de una de las partes de una regla de asociación por la parte de otra.

2. Operador de mutación - permite mutar o cambiar aleatoriamente a un individuo para generar mayor diversidad. Los autores de [1], por ejemplo, definieron un operador de mutación que permite cambiar el lado de la regla de asociación en que aparece un atributo dado, además de modificar el intervalo de los atributos.

3. Condición de término - Define la cantidad de iteraciones antes de terminar con el algoritmo. En este caso en particular, no se menciona la cantidad de iteraciones. Sin embargo, como cada proceso evolutivo produce un regla de asociación, los autores ejecutan el proceso completo varias veces hasta obtener un set de reglas de asociaciones candidatas.

4. Función de fitness - Nos sirve medir qué tan bueno es un individuo, pues en un algoritmo genético se seleccionan ciertos individuos de una población con mayor o menor probabilidad de acuerdo a este criterio. En este paper, se utilizan medidas clásicas asociadas a las reglas de asociación, vale decir, el soporte, confianza, y un valor de accuracy que describe qué tanto se ajusta la regla encontrada a los datos reales que tenemos.

Adicionalmente, el proceso se repite utilizando distintos datasets. Una vez se obtienen una cierta cantidad de reglas de asociaciones predefinidas, se intersectan los pares de atributos encontrados para cada dataset.

5.3. Ventajas

1. Los rangos de los atributos son variables y no es necesario definir previamente muchos parámetros ni tener en cuenta el rango de valores de los atributos

2. Permite resolver un problema complejo que un algoritmo como Apriori no podría encontrar midiendo solamente niveles de soporte y confianza

3. Uso de memoria no es elevado

4. Estructuras de datos usadas son relativamente simples

5. Permite buscar un espacio de hipótesis alto

5.4. Desventajas

1. Complejidad computacional alta por la cantidad de iteraciones que se deben ejecutar normalmente en un algoritmo genético

2. Al ser un proceso de búsqueda estocástico no se garantiza optimalidad

3. Puede ser complejo definir los operadores genéticos adecuados

4. Aborda un problema no muy general

6. Método Implementado - RELIM

6.1. Objetivo

El objetivo del RELIM es encontrar tal como en cualquier algoritmo de *ítems sets frecuentes* todos los ítems sets que cumplan un umbral dado por un usuario. Este algoritmo fue un de los algoritmos precursores en el uso de un esquema de *eliminación recursiva*, una de las diferentes técnicas que se usan para encontrar los k ítems sets frecuentes. Tal como el algoritmo SAM, la estrategia aquí adoptada consiste en efectuar una busca en profundidad primero o sea, se seleccionan todas las transacciones que contienen un ítem como líder que se verificó inicialmente como el más frecuente y después, se eliminan todas las ocurrencias de este ítem en las transacciones. Así se obtiene una base de datos más pequeña que va ser usada recursivamente para obtener

todos los ítems sets que comparten el mismo prefijo y posteriormente se iteran sobre los restantes ítems líder en la bases de datos original. Aún que Relim sea inspirado en algoritmos como FP-Growth, algoritmo clásico que consigue minimizar el tiempo de procesamiento sobre distintas bases de datos, su gran objetivo es simplificar la estructura de almacenamiento y al mismo tiempo intentar no comprometer su eficiencia.

6.2. Descripción

Este algoritmo tira partido de un mecanismo ventajoso para el procesamiento de los datos que se designa esquema de eliminación recursiva. Antes hacer uso dese mismo mecanismo, la base de datos transacciones pasa por un conjunto de pasos de pre-procesamiento que tienen como función principal preparar y organizar las transacciones para que pueden ser guardas en las estructuras de datos.

1. Inicialmente se calculan las frecuencias (suporte) de cada uno de los ítems individualmente

2. Se descartan los que no cumplan el umbral porque como sabemos un ítem set que no es frecuente, nunca podrá ser parte integrante de un ítem set frecuente.

3. En seguida se ordenan cada un de los ítems presentes en las transacciones de forma ascendente según las frecuencias encontradas en el paso anterior.

4. Se constroye la estrutura de datos que soporta el algoritmo - Contrariamente al algoritmo SAM, RELlM organiza los datos a procesar de forma diferente. Para ese efecto, representa los datos según una representación casi vertical. En cada posición del arreglo se mantiene una lista de transacciones que es indexada por cada ítem líder que existe en la base de datos.

   ítem líder es el primero item de cada transacción

   Además, cada posición se pode definir como un puntero que apunta para cada para a lista de transacciones y un contador que representa

el soporte del ítem considerado. Por veces, pude acontecer que este valor de suporte sea superior al numero de transacciones en la lista ya que existen transacciones que no poseen ningún ítem, solamente el ítem líder y por eso cuando son mapeadas en la estrutura apenas el contador es incrementado.

El mecanismo de eliminación recursiva se traduce por una función recursiva que es llamada para cada un de las listas de transacciones. Antes de procesar la lista de transacciones se testa se el valor del suporte respecta el umbral mínimo. Caso respecte, significa que el ítem set constituido por el *ítem líder* que estamos a analizar juntamente todos los items que son retornados por la recursión son frecuentes.

Para obtener todos los items set frecuentes (usando recursión) hay que procesar cada lista de lista de transacciones en una nueva estructura y para eso se usan las transacciones que el item líder apunta.

Después de una lista de transacciones ser procesada, sus transacciones son asignadas sucesivamente a las otras posiciones del arreglo (otras listas de transacciones) porque en cada una de esas transacciones podrá estar un nuevo *frecuent item-set* independiente de su *ítem líder*, algo idéntico con lo que el algoritmo SAM hace con la *operación de Merge*. La diferencia es que en este caso el algoritmo no identifica transacciones con ítems repetidos y por veces, podrán existir duplicados en una lista de transacciones.

6.3. Implementación

Como ya fue dicho anteriormente, las ventajas asociadas al algoritmo Relim resultan del abordaje simplista que es utilizado para manipular cada una de las transacciones. Por lo tanto, se tornaba crucial crear la estructura de datos en Pyton, respectando esta idea. Con esto, decidimos que la mejor manera de implementar el algoritmo seria a través de una lista, *RelimStructure*, en que cada posición posee una lista constituida por un contador de soporte y una *Queue* - destinada a almacenar las transacciones indexadas por el *ítem líder*. El uso de una Queue constituye una importante decisión de diseño porque torna la implementación más simples ya que facilita mucho la inserción de futuras transacciones así como su obtención a cualquier momento (necesarias en casa paso de la recursión) a través de las operaciones **Put** y **Get**.

En general el algoritmo se implementó como estaba previsto y se obtuvieran las siguientes funciones:

- **InsertTransaction**: función que inserta una transacción en la lista de Transacciones, usando el itemLider como index.Para hacer eso, es necesario conocer el itemindex, que el ItemLider tiene en lista Sortedscore. La inserción de una transacción implica el incremento del contador asociado al item lider y inserción el la respectiva Queue.

- **SearchTransactions**: función que retorna las transacciones que están guardadas en la TransactionQueue.Iterativamente recupera todos los valores hasta que la Queue permanezca vacía.

- **CreateRelimStructureWhithoutTransactions**: función que crea la estructura (vacía) que irá soportar el algoritmo.

- **ConstructRelimStructureWithTransactionLists**: función que inicialmente crea la estrutura vacía - CreateRelimStructureWhithoutTransactions. Por cada transacción que existe en la lista de Transacciones insertamos la transacción el estrutura vacia -InsertTransaction

- **RemoveInFrecuentItems**: función que recibe el total de Transacciones (Database, remove de la Base de Datos los item infrecuentes y actualiza la lista de suporte.

- **Relim**: función recursiva que itera sobre todas las listas de transacciones individualmente almacenadas en la RelimStructure y encuentra todos los item-sets, para cada ítem líder.

- **DataPreprocessing**: función que pre-procesa los datos (usando los pasos ya descritos) de la base de datos original y los convierte en la estructura de datos usada por el algoritmo.

El modelo de procesamiento adoptado por este algoritmo para manipular las transacciones permitió programar Relim en pocas lineas de código.

Test	Min_Thresh	Execution Time
grouptest	= 2	0.01 seconds
chess	= 3000	24,6 seconds
mushroom	= 3000	208,3 seconds

Figura 3: Tests Results

6.4. Resultados Experimentales

Nuestra implementación del algoritmo RELIM fue evaluada en dos tipos de bases de datos, esparzas – bases de datos con baja fracción de ítems y densas – bases de datos con alta fracción de ítems. Para atingir ese objetivo se realizaran 3 distintos testes:

- **Teste1: grouptest.dat** – un teste creado por el grupo para simular una base de datos con pocos items y pocas transacciones a procesar. Para este teste se utilizaran 5 items y un total 19 Transacciones.
- **Teste2: chess.dat** – un teste creado para simular el algoritmo sobre una base de datos densa, alto número de ítems 75 (valor máximo) y un total de 3196 Transacciones (Threshold próximo del número de total de transacciones).
- **Teste3: mushroom.dat** – un teste creado para para simular el algoritmo sobre una base de datos densa, alto número de ítems 119 (valor máximo) y un total de 8129 Transacciones (Threshold distante del número de total de transacciones).

Los resultados obtenidos comprueban que este algoritmo es especialmente eficaz para bases de datos esparzas (baja fracción de ítems presentes) porque la estructura de datos a procesar, después de remover items infrecuentes es considerablemente pequeña, ayudando en la computación del resultado. Cuando se analiza este algoritmo sobre el chess dataset por ejemplo, (base de datos densa), el desempeño del algoritmo no es drásticamente afectado (24,6s) porque en este caso consideramos un suporte mínimo próximo del número total de transacciones para evaluar esta situación.

Si analizarnos el desempeño del algoritmo en el mushroom dataset (bajo threshold face al número de transacciones) su desempeño es drásticamente afectado por lo que conseguimos contemplar por los resultados en la tabla. Esto se puede explicar, argumentando que en este tipo de datasets (densos), há mayor probabilidad de existieren transacciones repetidas generadas en la recursión del algoritmo y por consiguiente, tornar su procesamiento más lento. De referir que no tenemos ningún tratamiento especial a las transacciones repetidas, las insertamos normalmente como cualquier transacción.

Referencias

[1] M. Martínez-Ballesteros, I.A. Nepomuceno-Chamorro, J.C. Riquelme, Discovering gene association networks by multi-objective evolutionary quantitative association rules, Journal of Computer and System Sciences 80, 2014, pp. 118-136

[2] BitTableFI: An efficient mining frequent itemsets algorithm Jie Dong, Min Han, School of Electronic and Information Engineering, Dallan University of Technology, Dalian 116023,

[3] An effective Hash-Based Algorithm for Mining Association Rules Jong Soo Park, Ming-Syaen Chen, IBM Thomas J. Watson Research Center

[4] Aravind Seshadri, A Fast Elitist Multiobjective Genetic Algorithm: NSGA-II

[5] Christian Borgelt Keeping Things Simple: Finding Frequent Item Sets by Recursive Elimination

[6] Christian Borgelt Simple Algorithms for Frequent Item Set Mining