Trabajo de Grado
Presentación Final

Alumno: Clac, Santiago Raúl
Nro. Alumno: 1506/9
Nombre del Trabajo:

Descubrimiento de Reglas de Asociación Temporal - Una implementación -

Director: Ale, Juan María
Co-Director/a: Gordillo, Silvia
Clasificación: Desarrollo Tecnológico

Indice

OBJETIVO **3**

MOTIVACION **3**

Data Mining **4**

El problema conocido como *Mining Associations Rules* **15**

Descubrimiento de Reglas Temporales 37

Generación de los Itemsets Frecuentes 37

Generación de las Reglas 40

CONTRIBUCION **41**

DESARROLLO **42**

WEKA: Data Mining System **44**

Arquitectura de la aplicación Weka – Explorer **49**

Importación de las bases de datos a analizar 55

Selección y ejecución del algoritmo de asociación 57

Finalización de la ejecución del algoritmo de asociación 59

Especificación del prototipo desarrollado **61**

Modelos de datos **78**

Clases derivadas de clases existentes 78

Clases no derivadas de clases existentes 90

Diagrama del modelo de datos 101

Archivo de Configuración **102**

Características generales 102

Ejecución múltiple de procesos 106

EXPERIMENTACIÓN **108**

Generación de Datos Sintéticos **108**

Unificador de archivos de datos sintéticos 115

Pruebas de performance **122**

REFERENCIAS BIBLIOGRÁFICAS **142**

ANEXO I **143**

ANEXO II **145**

OBJETIVO

El objetivo de este trabajo de grado es desarrollar e implementar un prototipo que permita el descubrimiento de las reglas de asociación temporal, que cumplen ciertas restricciones, utilizando el tiempo como factor para generar reglas más precisas y descubrir otras, que sin acotarlas a un período de tiempo, no serían descubiertas.

MOTIVACION

El problema del descubrimiento de reglas de asociación radica en la necesidad de descubrir patrones en una base de datos de transacciones conocido como análisis de canasta de mercado [1], teniendo en cuenta el carácter temporal de las transacciones que representan los datos.
En grandes volúmenes de datos, como en los usados para los propósitos de Data Mining, encontramos información relacionada con los items; que no necesariamente existen a lo largo de todo el período de recolección de los datos.
Encontramos, entonces algunos items que en el momento de llevar a cabo el minado, han sido discontinuados. También podemos encontrar items nuevos que han sido introducidos después del comienzo de la recolección. Algunos de estos nuevos items podrían participar en las asociaciones, pero no pueden ser incluidos en algunas reglas debido a las restricciones de soporte.

Una forma de resolver este problema es incorporando el tiempo en el modelo de descubrimiento de reglas de asociación, estas reglas han sido llamadas **Reglas de Asociación Temporal** [2] [5].

Utilizando una función del tiempo de vida de la transacción, es posible eliminar reglas desactualizadas, de acuerdo al criterio del usuario, reduciendo la cantidad de trabajo necesario para determinar los itemset (conjunto de items) frecuentes, en la determinación de las reglas de asociación.
La idea básica para el descubrimiento de las reglas de asociación temporal es limitar la búsqueda a los itemsets frecuentes, en el período de vida de los miembros del itemset. Con esta idea, podríamos considerar el caso de algunos items que pueden ser frecuentes en algunos subintervalos estrictamente contenidos en su período de vida [5], pero no en el intervalo completo correspondiente a su período de vida.
De esta manera, cada regla tiene asociado un conjunto de intervalos de tiempo, correspondiente al período de vida de los items que participan en la regla.

Data Mining, también conocido como "descubrimiento del conocimiento en base de datos", ha sido reconocida como una nueva área para los investigadores de las bases de datos. Esta área puede ser definida como descubrimiento eficiente de las reglas de interés a partir de grandes colecciones de datos.

Data Mining

Los fundamentos

Gran parte de la ola actual de interés en data mining surge de la confluencia de dos grandes fuerzas: la necesidad del minado de los datos (*drivers*) y el significado de implementarlos (*enablers*). Los *drivers* son primariamente los cambios en el ambiente de negocios los cuales han resultado en un mercado cada vez más competitivo. Muchas organizaciones han sido forzadas a abandonar sus tradicionales enfoques para hacer negocios y han comenzado por mirar las formas de responder a los cambios en el ambiente de negocios. Los principales requerimientos que manejan esta reevaluación son:

- Enfoque en el cliente
  El requerimiento aquí es rejuvenecer las relaciones con el consumidor con un énfasis en mayor intimidad, colaboración, y sociedad *uno a uno*. A su vez, estos requerimientos han forzado a las organizaciones a preguntarse nuevas cuestiones acerca de sus consumidores y potenciales consumidores.
- Enfoque en la competencia
  Las organizaciones necesitan focalizar cada vez más en las fuerzas competitivas con la visión de construir un más moderno arsenal de armas de negocios. Algunos de las tareas a realizar son:
  - Predicción de potenciales estrategias o mayores planes de negocios para liderar la competencia.
  - Predicción de movimientos tácticos por competidores locales.
  - Descubrimiento de subpoblaciones de consumidores existentes que son especialmente vulnerables a los esfuerzos competitivos.
- Enfoque en el recurso dato
  Hay un reconocimiento creciente entre los manejadores de negocios y Tecnologías de la Información que existe actualmente una oportunidad de manejar información que es tomada del medio.
  Muchas organizaciones están empezando a utilizar sus recursos de datos acumulados como un medio/recurso de negocios crítico.

Los *enablers* son principalmente recientes avances en investigación de aprendizaje automático, bases de dato, estadística y tecnologías de visualización.
Hay un conjunto de *enablers* para data mining los cuales, cuando son combinados con las fuerzas de manejo listadas anteriormente, substancialmente incrementan el impulso hacia un acercamiento revisado a la toma de decisión del negocio.

- Flujo de datos.
- Crecimiento del Data Warehousing.
- Nuevas soluciones en Tecnología de la Información.
- Nuevas investigaciones en aprendizaje automático.

El efecto neto de los cambios en el ambiente de negocios es que la toma de decisiones se volvió mucho más complicada, los problemas se han transformado más complejos y los procesos de toma de decisiones menos estructurados. La toma de decisión actual necesita un conjunto de estrategias y herramientas para dirigir sus cambios fundamentales.

Hacia una definición

Es difícil establecer sentencias definitivas sobre un área de evolución, y ciertamente data mining es un área en muy rápida evolución. Sin embargo, necesitamos un framework dentro del cual posicionarnos y entender mejor el tema.

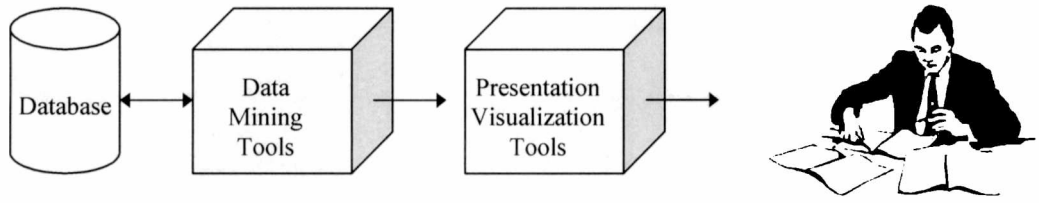

Figura 1
Posicionamiento de Data Mining

Aunque no existe una única definición de data mining que tenga la aprobación universal, la siguiente definición es generalmente aceptable:

> "Data Mining es el proceso de extraer información previamente **desconocida**, **válida** y **accionable** desde una base de datos grande y usar esta información para hacer **decisiones de negocios** cruciales".

Las palabras en "negrita" de la definición focalizan la atención en la naturaleza esencial del data mining y ayudan a explicar las diferencias fundamentales entre el data mining y los enfoques tradicionales de análisis de datos como *query-reporting* y *online analytical process* (OLAP).
En esencia data mining es distinguible por el hecho de que es dirigido al descubrimiento de la información sin una hipótesis previamente formulada.

Primero, la información descubierta debe ser previamente **desconocida**. Aunque esto suene obvio, el problema real aquí es que debe ser improbable que la información podría haber sido hipotetizada de antemano, es decir, el *data miner* está mirando algo que no es intuitivo o, por ejemplo, no contablemente intuitivo.
Data mining puede revelar información que podría incluso no haber sido hipotetizada en enfoques anteriores.
Segundo, la nueva información debe ser **válida**. Este elemento de la definición hace referencia al problema de optimismo excesivo de data mining, esto es, si los data miners buscan bastante duro en una colección de datos grande, ellos están orientados a encontrar algo de interés tarde o temprano.
Tercero, y más crítico, la nueva información debe ser **accionable**, esto es, debe ser posible transformarla en alguna ventaja para el negocio.
Sin embargo, explotar la aparente oportunidad puede requerir el uso de datos que no están disponibles o legalmente usables. Aunque parezca innecesario decirlo, una organización debe tener las políticas necesarias para llevar a cabo la acción implicada por data mining.

La habilidad para usar el dato minado para generar las **decisiones de negocios** cruciales es otra condición ambiental crítica para el data mining comercial de utilidad y apuntala la asociación fuerte del data mining con y la aplicabilidad a los problemas de negocio.

Data mining es pensado a menudo a partir de uno o más de los siguientes conceptos:
- Consultas SQL contra un gran data warehouse.
- Consultas SQL contra un número de dispares bases de datos o data warehouses.
- Consultas SQL en un ambiente paralelo masivo.
- Recuperación de información avanzada (por ejemplo, agentes inteligentes).
- Análisis de bases de datos multidimencionales (MDA).
- OLAP.
- Análisis de exploratorio de datos (EDA).
- Visualización gráfica avanzada.
- Procesamiento estadístico tradicional contra un data warehouse.

Ninguno de estos enfoques puede ser considerado como data mining porque carecen de un ingrediente esencial: *el descubrimiento de la información sin una hipótesis previamente formulada.*

Una buena regla es que, dada la naturaleza exploratoria del data mining, siempre que conozcamos la forma y probablemente el contenido de lo que estamos buscando/mirando, entonces probablemente no estamos tratando con un problema de data mining.
Todos los escenarios anteriores tienen en común una cosa: la hipótesis.
Data mining, sin embargo, eleva las metas con respecto a lo que puede lograr con el análisis de datos tradicional, haciendo preguntas que están más allá del alcance de las técnicas tradicionales y, lo más importante, tienen mucho más valor de negocios.

De todas las técnicas en análisis de datos tradicional, la estadística está más íntimamente relacionada al data mining. En efecto, es muy común escuchar decir que la estadística tradicionalmente ha sido usada por la mayoría de los analistas que ahora están usando data mining para construir modelos predictivos y para descubrir asociaciones en las bases de datos.
Lo atractivo del data mining para muchos analistas es la relativa facilidad con la cual nuevas visiones pueden ser ganadas en comparación con los enfoques estadísticos tradicionales.

La accesibilidad de data mining puede ser mostrada de muchas maneras. Por ejemplo, el data mining es siempre un enfoque libre de hipótesis, mientras que la mayoría de las técnicas estadísticas requieren el desarrollo de una hipótesis de antemano, un estadístico típicamente tiene que desarrollar manualmente las ecuaciones que verifican la hipótesis. En contraste, los algoritmos de data mining pueden desarrollar estas ecuaciones automáticamente.
Debido al esfuerzo adicional en estadística, muchos analistas limitan su uso a algunas técnicas estadísticas avanzadas, como por ejemplo, regresión lineal.

Finalmente, el tipo de dato que es aceptable también ayuda a expandir algunas de las atracciones del data mining. Considerando que las técnicas estadísticas usualmente manejan sólo datos numéricos y necesitan hacer duras suposiciones sobre su

distribución, los algoritmos de data mining típicamente pueden procesar un conjunto más amplio de tipos de datos y hacer pocas suposiciones acerca de su distribución.

La conexión con Data Warehouse

Data warehousing, aunque es un tema separado y práctico de data mining, está, en la mayoría de los casos, muy estrechamente asociado con él.
Es necesario tener presente que data warehouse no es un prerequisito para una solución data mining. En efecto, a pesar de que data warehouse es uno de los *drivers* que han aumentado su interés en data mining, la mayoría del data mining de hoy está hecho sobre archivos planos los cuales han sido extraídos directamente de fuentes de datos *operacionales.*

El objetivo de un data warehouse es ayudar a mejorar la efectividad de la toma de decisión en el negocio del manejo de datos.
El concepto está basado fundamentalmente en la distinción entre el *dato operacional* (usado para movilizar la organización) y el *dato informativo* (usado para manejar la organización).
El data warehouse está diseñado para ser un área de almacenamiento neutral para el dato informativo y se piensa que es la única fuente de datos de calidad de la compañía para la toma de decisión.
Inmon (1992) ofrece una definición aceptada de un data warehouse:

> "Un data warehouse es una colección de datos **orientada al negocio integrada, variante en el tiempo,** y **no volátil** en apoyo de las decisiones de la dirección."

Como vimos, la definición requiere ciertas características de los datos almacenados en el data warehouse, los cuales veremos en detalle:

Orientada al negocio: El dato en el warehouse está definido y organizado en términos del negocio y está agrupado bajo encabezados de temas orientados el negocio. Esta orientación al negocio es lograda a través del modelado de los datos.

Integrada: Los contenidos del data warehouse se definen tal que son válidos para la empresa e integran datos provenientes de sus fuentes, tanto externas como operacional.

Variante en el tiempo: Todos los datos en el data warehouse están identificados en el tiempo en el momento de entrada dentro del warehouse o cuando son sumarizados dentro del warehouse. Así actúa como un registro cronológico y provee la posibilidad de realizar análisis histórico y de tendencia.

No volátil: Una vez cargado dentro del data warehouse, el dato no puede ser actualizado. Así, actúa como un registro estable para reportes de consistencia y análisis comparativo.

Muchos analistas y corporaciones ven a data warehousing como uno de los mayores vehículos estratégicos para ejecutar la toma de decisiones.

Claramente, preparar un data warehouse no es una tarea trivial, especialmente si el objetivo es servir a la empresa en su totalidad. Así en los años recientes muchas organizaciones han optado por el *data mart*, el cual es más especializado, más accesible

y mucho más pequeño que un data warehouse a nivel empresarial. Por lo cual, es un excelente primer paso para muchas organizaciones.
Para aquellas organizaciones que seguramente tienen data warehouses repletos de información, un data mart es un dispositivo usado para procesamiento especializado.

El paso *desde un data warehouse al data mining* es uno que las organizaciones con una visión progresiva de los sistemas de soporte de decisión están tomando cada vez más. La razón para moverse desde un data warehouse a data mining surge de la necesidad de incrementar la influencia que una organización puede conseguir de un enfoque warehouse existente y la necesidad de manejar data warehouse de grandes volúmenes.

Luego de implementar una solución data mining, la organización podría decidir integrar la solución en un enfoque de manejo de datos más extenso para la toma de decisión del negocio. En este sentido tenemos el paso desde un data mine a un data warehouse.

Data Mining e Inteligencia Comercial

El término inteligencia comercial es usado como un término global para todos los procesos, técnicas y herramientas que soportan la toma de decisión comercial basado en tecnología de la información.
Este enfoque puede abarcar desde una simple planilla de cálculo a la tarea de una inteligencia competitiva mayor.

El Data Mining es un nuevo componente importante de la inteligencia comercial.

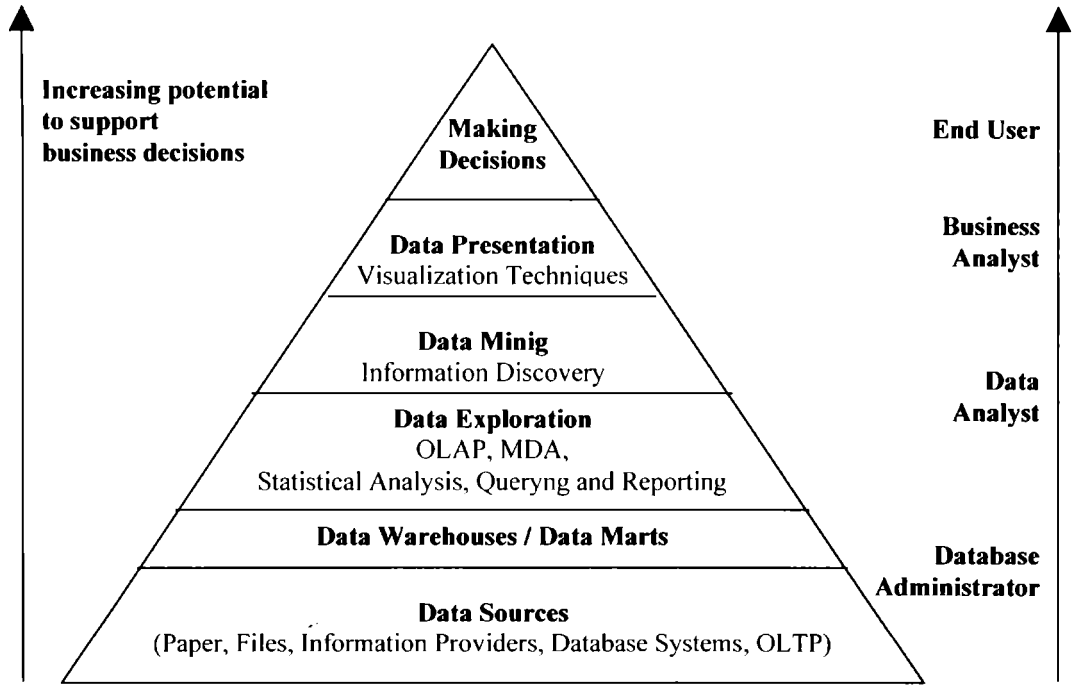

Figura 2
Data Mining e Inteligencia Comercial

En la **Figura 2** vemos el rol que ocupa el Data Mining en la pirámide comparativa de acuerdo al tipo de información manejada y a quién está dirigida.

En general, el valor de la información para el soporte de la toma de decisiones aumenta desde la base de la pirámide hasta la parte superior. Una decisión basada en datos de las capas inferiores, donde hay típicamente millones de registros de datos, podría afectar sólo a una única transacción de un consumidor. Mientras que una decisión basada en los datos sumarizados de las capas superiores es mucho más probable que sea sobre iniciativas de la compañía o departamentos o incluso la dirección mayor.
Por consiguiente, generalmente se encuentran diferentes tipos de usuarios en las diferentes capas.

Aplicación del Data Mining en los Negocios

La tabla muestra las tres áreas de negocios generales donde el data mining es aplicado y lista algunas de las aplicaciones comunes en cada área.

Manejo del Mercado	Manejo de los Riegos	Manejo del Fraude
✓ Target marketing	✓ Forecasting	✓ Fraud detection
✓ Customer relationship management	✓ Customer retention	
✓ Market basket analysis	✓ Improved underwriting	
✓ Cross selling	✓ Quality control	
✓ Market segmentation	✓ Competitive analysis	

Aplicaciones de Manejo de Mercado

El manejo de mercado es una de las áreas de aplicación mejor establecidas para el data mining. Por ejemplo, el área de aplicación mejor conocida es *database marketing*.

El objetivo es identificar el destino (*target*) y a partir de ese momento realizar efectivas campañas de marketing y promociones a través del análisis de bases de datos corporativas. Cuando la información disponible es mezclada con la publicidad, esta información se convierte en un arma importante para atraer nuevos consumidores.

Los algoritmos de data mining separan los datos, identificando grupos de consumidores "modelo", los cuales comparten las mismas características. Claramente estos grupos son un blanco perfecto para los esfuerzos del marketing.
Este es un juego de mutua ganancia, porque tanto los consumidores como los vendedores se ven beneficiados: los consumidores perciben un valor importante en el número reducido de mensajes de advertencia, y los vendedores se benefician limitando sus costos de distribución y obteniendo una respuesta a su campaña.

Otra área de aplicación para el data mining es la determinación de patrones de compra de los consumidores. Los vendedores pueden determinar mucho sobre el comportamiento de sus consumidores, como la secuencia en la cual ellos toman servicios financieros cuando su familia crece, o como ellos cambian sus autos. Entendiendo estos patrones, los vendedores pueden anunciar *just-in-time* a los consumidores, asegurándose que el mensaje está enfocado y probablemente obtenga una respuesta.
El propósito de enfocar en la vida de los consumidores más estrechamente como individuales, con sus propios conjuntos de valores, expectativas, y hábitos, y verlos como inversiones para toda la vida, es a menudo referido como *marketing uno a uno* o *marketing a un segmento de uno*.

Las campañas de Cross-selling constituyen otra área de aplicación donde el data mining es ampliamente usado. Cross-selling es donde un minorista o proveedor de servicios hace atractivo para clientes que compran un producto o servicio, comprar también un producto o servicio asociado.

Aplicaciones de Manejo de Riegos

El manejo de riegos cubre no sólo los riegos asociados con seguros e inversiones, sino también los riegos de negocios más grandes que surgen desde amenazas competitivas, pobre calidad de producto, reducción de clientes.

El riesgo es un aspecto principal de la industria del seguro, y el data mining está bien preparado para predecir las pérdidas de propiedad o por accidente de los asegurados. Estas predicciones tienen usualmente la forma de reglas que soportan subscriptores.

Además, entender la exposición de pérdida total más precisamente puede soportar mejoras en la carpeta de asegurados total.
El precio variable para pólizas de seguro es una de las formas en las cuales los aseguradores pueden dirigir diferencias en los riegos financieros asociados con diferencias entre los asegurados.
La reducción significa la pérdida de clientes, especialmente en manos de los competidores. La reducción es un problema creciente en un ambiente de mercado cada vez más competitivo, y el data mining es usado en las industrias de finanzas, minoristas, y telecomunicaciones para predecir probables pérdidas de consumidores.

El enfoque general es construir un modelo de un consumidor vulnerable, esto es, un consumidor el cual muestra características típicas de uno quien es probable que nos deje por una compañía competidora.
El análisis de estos consumidores a menudo muestra patrones no intuitivos.

Las organizaciones minoristas usan el data mining para entender mejor la vulnerabilidad de ciertos productos para competir en las ofertas y cambiar los patrones de compra de los consumidores.
Los patrones de compra históricos de los consumidores son analizados para identificar grupos de consumidores con preferencias por producto o lealtad a la marca.

Aplicaciones de manejo de fraude

La naturaleza humana indica que algún nivel de fraude es inevitable en todas las industrias. Sin embargo, algunos sectores, generalmente aquellos donde hay muchas transacciones, parecen sufrir más que la mayoría.
Por esta razón, muchas organizaciones (en las áreas como cuidados de la salud, ventas minoristas, servicios de tarjetas de crédito, y telecomunicaciones) están avanzadas en el uso de data mining para resguardarse del fraude o del potencial fraude.

El enfoque general es por ahora uno bastante familiar: usar los datos históricos para construir un modelo de comportamiento fraudulento o potencial comportamiento fraudulento y usar data mining para ayudar a identificar casos similares de su comportamiento.

Áreas de aplicación emergentes y futuras

Dos de las áreas en vías de desarrollo son *text mining*, la aplicación de la tecnología de data mining tradicional al contenido no estructurado de las bases de datos de textos, y *web analytics*, el cual propone usar la tecnología de data mining en conjunción con Internet.

Text Mining: El objetivo no es la crítica literaria, por el contrario, el objetivo es hacer que el gran volumen de información textual sea dócil para el análisis rápido y la comprensión.
Típicas aplicaciones son el indexado automático de documentos y el análisis de título automático.
Considerando que las técnicas de investigación tradicional son típicamente contrarias al enfoque de *word-matching* literal y la inventiva del usuario, el enfoque de *text mining* abre la posibilidad de seleccionar información relacionada que el usuario no especificó inicialmente o incluso pensaba posible.

Web Analytics: Web analytics reúne dos de las áreas de enfoque más grandes en computación actual, la Internet y data mining.
La idea es aplicar el data mining a la actividad de *log* (registro de información) de los servidores Web para desarrollar análisis del comportamiento de los usuarios en Internet.
Actualmente, los links de hypertexto Web son fijos. Los desarrolladores de los sitios han proporcionado links intentando probablemente suponer los que el usuario quiere hacer luego.
Con data mining, los patrones de navegación históricos del usuario pueden ser analizados para sugerirle dinámicamente sitios relacionados para que el usuario visite.

El Proceso de Data Mining

En general, cuando una persona piensa sobre data mining, hace hincapié inicialmente en el aspecto actual de minado y descubrimiento.
La idea suena intuitiva y atractiva. Sin embargo, el minado de los datos es sólo un paso en el proceso completo.

La **Figura 3** ilustra el proceso como un proceso iterativo de muchos pasos. Como vemos, el proceso de Data Mining comienza y finaliza con los objetivos del negocio.

Los objetivos del negocio manejan el proceso de data mining completo.
Ellos son la base sobre la cual el proyecto inicial es establecido y la medida por la cual los resultados finales serán juzgados, y ellos constantemente guiarán al equipo a lo largo de muchos pasos en el proceso.

Aunque los pasos son ejecutados en el orden en el cual son presentados, el proceso es altamente iterativo, con posibles vueltas a uno o muchos pasos anteriores.
Además, el proceso está lejos de ser autónomo.

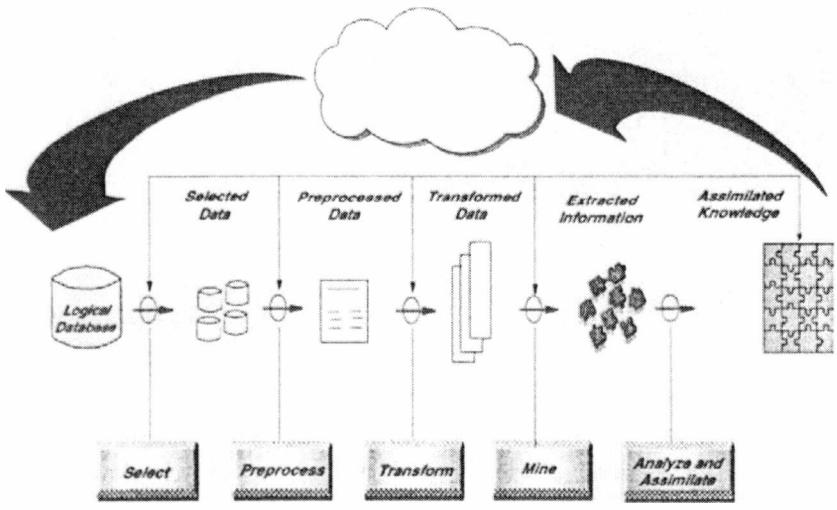

Figura 3

A pesar de los recientes avances en tecnología, data mining sigue siendo un ejercicio de muchísima labor intensiva.
Hablando de labor intensiva, no todos los pasos del proceso son de igual peso en términos de típico tiempo y esfuerzo gastado.

La **Figura 4** nos muestra en esfuerzo requerido por cada paso del proceso:

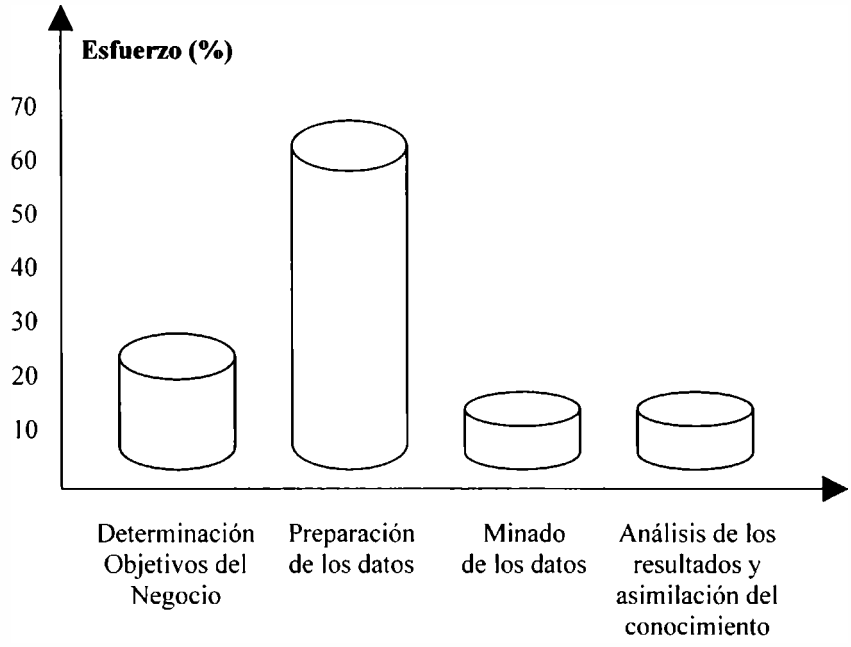

Figura 4
Esfuerzo requerido por cada paso en el proceso de Data Mining

Como podemos ver, el 60% del tiempo está relacionado con la preparación de los datos para el minado, resaltando la dependencia crítica sobre la depuración, de los datos relevantes.
El paso de minado de los datos propiamente dicho, típicamente constituye el 10% del esfuerzo total.

Ahora veremos en detalle cada uno de los pasos que componen el proceso de Data Mining.

Paso 1: Determinación de los Objetivos del Negocio
Claramente define los problemas o desafíos del negocio.
Este es un ingrediente esencial en cualquier proyecto de data mining. Aunque suene intuitivo y directo, no lo es.
El data mining por el propio bien del data mining es raramente exitoso.

Paso 2: Preparación de los Datos
Se divide en tres pasos:

Paso 2.1: *Selección de los Datos*
Identifica todos las fuentes internas y externas de información y selecciona a cuales de los subconjuntos de datos identificados es necesario realizarle la aplicación de data mining.

Paso 2.2: *Pre-procesamiento de los Datos*
Estudia la calidad de los datos para preparar el camino para el futuro análisis y determinar el tipo de operación de minado que será posible y valioso realizar.

Paso 2.3: *Transformación de los datos*
Transforma los datos en un modelo analítico. Modela los datos para satisfacer el análisis propuesto y los formatos de los datos requeridos por los algoritmos de data mining, muchos de los cuales tienen requerimientos particulares. La presentación de un modelo de datos analítico legítimo de los datos a los algoritmos de minado es crítico para el éxito.

Paso 3: Minado de los Datos
Minar los datos transformados en el paso 2.3.
Este es el corazón de la tarea, y, a parte de seleccionar la combinación apropiada de algoritmos de data mining, es rápido y automatizado.

Paso 4: Análisis de los Resultados
Interpretar y evaluar la salida del paso 3.
El enfoque de análisis utilizado puede variar de acuerdo a la operación de data mining, pero típicamente involucrará alguna técnica de visualización.

Paso 5: Asimilación del Conocimiento
Incorporar la visión comercial adquirida en el paso 4 en los negocios y los sistemas de información de la organización.

Para entender mejor el proceso de data mining, se definen los roles de los participantes en los pasos de data mining:

- Analista Comercial:
Además de entender el dominio de la aplicación, interpreta los objetivos y los traslada en requerimientos comerciales que servirán para definir los datos y los algoritmos de minado a ser usados.

- Analista de Datos:

Además de entender las técnicas del análisis de datos, y tener un gran conocimiento en estadísticas, tiene la especialización y la habilidad necesarias para transformar los requerimientos del negocio en operaciones de data mining y seleccionar la correspondiente técnica de data mining para cada operación.

- Especialista en el manejo de Datos:

Es especialista en técnicas de manejo de datos y recoge datos desde los sistemas operacionales, bases de datos externas, o data warehouses.

El problema conocido como *Mining Associations Rules*

El problema conocido como *mining associations rules* fue introducido en [9].

Básicamente, tenemos que dado un conjunto de transacciones donde cada transacción es un conjunto de items, una regla de asociación es una expresión de la forma X => Y, donde X e Y son conjuntos de items disjuntos.
El significado intuitivo de cada regla es que las transacciones en la base de datos que contienen los items en X, tienden a contener también los items en Y.

Como ejemplo podemos tomar la siguiente regla: el 98% de los consumidores que compran neumáticos y accesorios para autos, también compran algunos servicios para el automotor, en este caso, el 98% es llamado la *confianza* de la regla. El *soporte* de la regla X => Y es el porcentaje de transacciones que contienen ambos X e Y, es decir que contienen los items que componen X y los items que componen Y.
La regla X => Y es contenida en la base de datos de transacciones con confianza *c* si c% de las transacciones de la base de datos que contienen a X, también contienen Y.
La regla X => Y tiene soporte *s* si s% de las transacciones de la base de datos contienen X U Y.

El problema de *mining associations rules* es encontrar todas las reglas que satisfacen el mínimo soporte y la mínima confianza especificada por el usuario, es decir que tienen soporte y confianza mayor que el mínimo soporte y mínima confianza especificada por el usuario.

En la actualidad existen muchos y muy diferentes algoritmos desarrollados que pueden ser utilizados para el descubrimiento de las reglas de asociación. Cada uno de ellos tiene un enfoque particular, dependiendo de la visión de su autor, de las características de los datos de entrada, de los parámetros involucrados en la búsqueda, y del método utilizado para el descubrimiento de las reglas de asociación, entre otros factores.
En general, el problema de descubrir todas las reglas de asociación puede ser descompuesto en dos subproblemas:

Encontrar todos los *itemsets* (conjuntos de items) que tienen soporte de transacción por encima del mínimo soporte (especificado por el usuario). Los itemsets cuyo soporte está por encima del mínimo soporte son llamados *large itemsets*.
Usar los *large itemsets* para generar las reglas de asociación deseadas.

Algunos de los más populares e importantes algoritmos de descubrimiento de las reglas de asociación son: AIS [9], SETM [10], Apriori, AprioriTid, AprioriHybrid [1], DIC [3], TDIC [2], entre otros.

De acuerdo al momento en que surgió y el propósito para el cuál fue desarrollado cada uno de los algoritmos, podemos organizarlos de siguiente forma, para conocer cuál ha sido la motivación que llevó al investigador a imaginar y, posteriormente, desarrollar el algoritmo y cuáles han sido las características de los algoritmos precedentes que los han influenciado en el desarrollo del mismo:

Algoritmo AIS

La **Figura 5** muestra un esquema del algoritmo definido en [9] para el descubrimiento de los itemsets frecuentes.
Dado un conjunto de items *I*, un itemset X+Y de items en *I* se dice que es una extensión del itemset X si $X \cap Y = \varnothing$.
En este caso particular, el parámetro *dbsize* representa el número total de tuplas de la base de datos.

```
procedure LargeItemsets
begin
  let Large set L = ∅;
  let Frontier set F = {∅};

  while F ≠ ∅ do begin
    // realiza una pasada sobre los datos
    let Candidate set C = ∅;
    forall database tuples t do
      forall itemsets f in F do
        if t contains f then begin
          let Cf = itemsets candidatos que son extensiones de f y están contenidas en t;
          forall itemsets cf in Cf do
            if cf ∈ C then
              cf.count = cf.count + 1;
            else begin
              cf.count = 0;
              C = C + cf;
            end
        end

    let F = ∅;
    forall itemsets c in C do begin
      if count(c) / dbsize > minsupport then
        L = L + c;
      if c puede ser utilizado como una frontera en al siguiente pasada then
        F = F + c;
    end
  end
end
```

Figura 5
Esquema del algoritmo AIS

El algoritmo realiza múltiples pasadas sobre la base de datos.
El *conjunto frontera* para una pasada, consiste de aquellos itemsets que son extendidos durante la pasada.
En cada pasada, el soporte de ciertos itemsets es calculado (medido). Estos itemsets, llamados itemsets candidatos, son derivados desde las tuplas de la base de datos y los itemsets contenidos en el *conjunto frontera*.

Asociado con cada itemset hay un contador que almacena el número de transacciones en las cuales el correspondiente itemset aparece. Este contador es inicializado a cero cuando el itemset es creado.

Inicialmente, el conjunto frontera consiste de sólo un elemento, el cual es el conjunto vacío.
Al final de la pasada, el soporte para un *itemset candidato* es comparado con el *mínimo soporte* para determinar si el itemset es un *itemset frecuente*.
En este mismo momento, se determina si el itemset puede ser agregado al *conjunto frontera* para la siguiente pasada.
El contador de soporte para el itemset es preservado cuando el itemset es agregado al *conjunto frontera (frecuente)*.
El algoritmo termina cuando el conjunto frontera se vuelve vacío.

De lo anterior, tenemos que el procedimiento para la generación de los itemsets candidatos puede ser realizado siguiendo el razonamiento que a continuación se detalla.
Un itemset no presente en alguna de las tuplas de la base de datos nunca se transformará en candidato para la medición.
El proceso consiste en leer una tupla por vez desde la base de datos y chequear que *conjuntos frontera* están contenidos en la tupla leída.
Los itemset candidatos son generados a partir de estos itemsets frontera extendiéndolos recursivamente con otros items presentes en la tupla. Un itemset que se espera que sea *no frecuente*, no es extendido.
Con el objetivo de no replicar las diferentes formas de construir el mismo itemset, los items son ordenados y el itemset X es extendido únicamente por los items que son posteriores en el orden que alguno de los miembros de X.

Decidir que itemsets poner en el siguiente conjunto frontera resulta un poco difícil.
Una forma de pensar sería suponer que es suficiente seleccionar sólo los itemsets frecuentes maximales (en términos de inclusión de conjuntos). Sin embargo, esta elección es incorrecta, ya que puede provocar que el algoritmo pierda algunos itemsets frecuentes.

Hasta el momento, sabemos que ninguna de las extensiones de un itemset que estamos incluyendo en el siguiente conjunto frontera han sido consideradas en la pasada actual. Pero, dado que los itemsets son actualmente frecuentes, ellos todavía pueden procesar extensiones que son frecuentes.
Por consiguiente, estos itemsets deben ser incluidos en el conjunto frontera para la siguiente pasada. Ellos no conducen a ninguna redundancia porque ninguna de sus extensiones han sido medidas (contadas) hasta ahora.
De hecho, si un itemset candidato era frecuente pero no se esperaba que fuera *small* (no frecuente), entonces este no debería estar en el conjunto frontera para la próxima pasada porque, de acuerdo a la forma en que el algoritmo está definido, todas las extensiones de un itemset ya han sido consideradas en esta pasada.
Un itemset candidato que es *small* no debe ser incluido en el siguiente conjunto frontera porque el soporte para una extensión de un itemset no puede mayor que el soporte para el itemset.

Por último, veremos algunas mejoras para manejar el hecho de que podemos encontrarnos con que no tenemos suficiente memoria para almacenar todos los conjuntos frontera e itemsets candidatos en una pasada.
Para ello, se han definido dos métodos de poda:
- Poda basada en contar las tuplas restantes en la pasada
- Poda basada en funciones de poda sintetizadas

Como vimos, es posible determinar durante una pasada que un itemset candidato podría eventualmente no transformarse en frecuente y descartarlo tempranamente. Este método de poda es conocido como *remaining prune optimization*, y su mayor virtud es salvar memoria y esfuerzo de medición.
Supongamos que un itemset candidato $X + Y$ es una extensión del itemset frontera X y que el itemset X aparece en un total de x tuplas. Supongamos también que $X + Y$ está presente en la cth tupla que contiene a X.
En el momento de procesar esta tupla, supongamos que el contador de tuplas (incluyendo esta tupla) que contienen a $X + Y$ es s.
Esto significa que existen al menos $x - c$ tuplas en las cuales $X + Y$ puede aparecer.
Entonces, comparamos *maxcount* = $(x - c + s)$ con *minsupport* por *dbsize*. Si *maxcount* es menor, entonces X + Y está destinado a ser *small* y puede ser descartado.

Ahora consideremos otra técnica que puede podar los itemsets candidatos tan pronto como son generados. Esta técnica es conocida como *pruning function optimizacion* y está impulsada por cada función de poda posible entendida como el precio de la transacción total.
El *precio de la transacción total* es una función acumulativa que puede ser asociada con un conjunto de items como una suma de precios de items individuales en un conjunto.
Si conocemos que hay menos que la fracción del mínimo soporte de transacciones que compraron por un valor mayor a Υ pesos de items, entonces inmediatamente podemos eliminar todos los conjuntos de items para los cuales su precio total excede Υ.
Cada itemset no tiene que ser medido e incluido en el conjunto de itemsets candidatos.

En general, no conocemos como son estas funciones de poda. Por el contrario, sintetizamos las funciones de poda de los datos disponibles. Las funciones de poda que podemos sintetizar, tienen la siguiente forma:

$$w_1 I_{j1} + w_2 I_{j2} + \ldots + w_m I_{jm} \leq \Upsilon$$

donde cada valor binario $I_{ji} \in I$.

Los pesos w_i son seleccionados de la siguiente manera:
Primero, ordenamos los items en orden decreciente de acuerdo a su frecuencia de ocurrencias en la base de datos. Entonces, el peso del i-ésimo item Ij, estará en el orden

$$w_i = 2^{i-1} \varepsilon$$

donde ε es un número real pequeño similar a 0.000001.

Una función de poda separada es sintetizada por cada itemset frontera. Estas funciones difieren en sus valores para Υ.
Habiendo determinado los itemsets frontera en una pasada, no queremos hacer una pasada auxiliar sobre los datos para determinar las funciones de poda. Por lo tanto, debemos coleccionar información para determinar Υ para un itemset X mientras X es un itemset candidato y está siendo medido anticipándose a que X pueda transformarse en un itemset frontera en la siguiente pasada.

Afortunadamente, conocemos que sólo los itemsets candidatos que se espera que sean *small* pueden transformarse en conjunto frontera.
Por lo tanto, sólo es necesario recolectar información de estos itemsets y no de todos los itemsets candidatos.

Algoritmo SETM

Fue motivado por el deseo de usar SQL para computar los *large itemsets*.
Como el algoritmo AIS, el algoritmo SETM también genera los candidatos "on-the-fly", basado en las transacciones leídas de la base de datos. De esta manera, genera y cuenta todos los itemsets candidatos que el algoritmo AIS genera.
Sin embargo, permite usar la operación JOIN del SQL estándar para la generación de candidatos.

Además, SETM separa la generación de candidatos del conteo.
Para lo cual, graba una copia de los itemsets candidatos junto con el TID (identificador de la transacción) de la transacción generada en una estructura secuencial.
Al final de la pasada, el contador de soporte de los itemsets candidatos es determinado por *ordenamiento* y *agregación* de la estructura secuencial.

SETM recuerda los TIDs de las transacciones generadas con los itemsets candidatos.
Para evitar la necesidad de utilizar la operación *subset*, el algoritmo usa esta información para determinar los itemsets frecuentes contenidos en la transacción leída.
$L_k \subseteq C_k$, y es obtenido eliminando aquellos candidatos que no tienen mínimo soporte.
Asumiendo que la base de datos está ordenada por TID, SETM puede fácilmente encontrar los itemsets frecuentes contenidos en una transacción en la siguiente pasada ordenando L_k por TID.
Para ello, necesita visitar todos los miembros de L_k sólo una vez en orden TID, y la generación de candidatos puede ser llevada a cabo usando la operación relacional *merge-join* [10].

La desventaja de este algoritmo radica principalmente en el tamaño de los conjuntos de candidatos C_k.
Por cada itemset candidato, el conjunto candidato tiene ahora tantas entradas como el número de transacciones en las cuales el itemset candidato está presente.
Sin embargo, cuando estamos leyendo para contar el soporte de los itemsets candidatos al final de la pasada, C_k está en un orden incorrecto y necesita estar ordenado de acuerdo a los itemsets que contiene.
Después de contar y quitar los itemsets que no tienen mínimo soporte, el conjunto resultante L_k necesita otra ordenación sobre TID antes que pueda ser usado para generar los candidatos en la siguiente pasada.

En [1], Agrawal-Srikant proponen otra forma de descubrir los itemsets frecuentes: hacer que los algoritmos para descubrir los itemsets frecuentes realicen múltiples pasadas sobre los datos.

En la primer pasada, cuentan el soporte de los itemsets individuales (los itemsets que tienen un sólo item) y determinan cuáles de ellos son frecuentes (*large*), es decir que tienen mínimo soporte.
En cada pasada subsecuente, comienza con los conjuntos de itemsets encontrados en la pasada anterior y los utiliza para generar los nuevos potenciales itemsets frecuentes, llamados *itemsets candidatos* y el soporte de estos itemsets candidatos, es contado a medida que se procesan los datos.
Al final de la pasada, se determina cuáles de los itemsets candidatos son actualmente frecuentes, y se convierten en datos base para la próxima pasada.
Este proceso continúa hasta que no sean encontrados nuevos itemsets frecuentes o se alcance el final de la base de datos.

Los algoritmos Apriori y AprioriTid que ellos proponen difieren fundamentalmente de los algoritmos AIS y SETM en términos de cuáles de los itemsets candidatos son contados en una pasada y la forma en que estos itemsets candidatos son generados.
En los algoritmos AIS y SETM, los itemsets candidatos son generados "on-the-fly" a medida que los datos son leídos en una pasada.
Específicamente, después de leer una transacción, se determina cuáles de los itemsets encontrados frecuentes en las pasadas previas están presentes en la transacción. Los nuevos itemsets candidatos son generados extendiendo estos itemsets frecuentes con otros items en la transacción.
Como vimos, la desventaja de estos algoritmos es que genera y cuenta muchos itemsets candidatos que pueden no ser frecuentes al final del proceso.

A diferencia de estos, los algoritmos Apriori y AprioriTid generan los itemsets candidatos que son contados en una pasada usando sólo los itemsets frecuentes encontrados en las pasadas anteriores, sin considerar las transacciones en la base de datos.
La idea intuitiva es que algún subconjunto de los itemsets frecuentes debe ser frecuente.
Además, los itemsets candidatos que contienen k items son generados *uniendo* los itemsets frecuentes que tienen k-1 items, y eliminando aquellos que contienen algún subconjunto que no es frecuente.
Este procedimiento resulta en la generación de un número mucho menor de itemsets candidatos.

El algoritmo AprioriTid tiene la propiedad adicional que la base de datos no es usada completa para contar el soporte de los itemsets candidatos después de la primer pasada, en lugar de esto, una codificación de los itemsets candidatos usados en las pasadas anteriores es empleado para este propósito.
En las últimas pasadas, el tamaño de las codificaciones se puede hacer mucho más pequeño que la base de datos.

Para mayor simplicidad, los autores asumen que los items de una transacción están ordenados por el orden lexicográfico.
Utilizando la notación c[1].c[2]...c[k] representan un *k-itemset c* consistiendo de los items **c[1]**, **c[2]**, ..., **c[k]**, donde **c[1] < c[2] <...< c[k]**.

Asociado con cada itemset, existe un campo contador para almacenar el soporte del itemset.

Algoritmo Apriori

El primer paso del algoritmo simplemente cuenta la ocurrencia de los items para determinar los 1-itemsets frecuentes; para lo cual recorre toda la base de datos de transacciones.
Un paso subsecuente, llamado el paso *k*, consiste de dos fases. Primero, los itemsets frecuentes L_{k-1} encontrados en la (k-1) ésima pasada son usados para generar los itemsets candidatos C_k, usando la función ***apriori-gen***.
Luego, la base de datos es recorrida y el soporte de los candidatos es contado.

Ya que la base de datos de transacciones es recorrida completa cada vez que queremos calcular el soporte de los k-itemsets, necesitamos un rápido conteo, es decir que, necesitamos determinar eficientemente los candidatos de C_k que están contenidos en la transacción *t*, para lo cual se utiliza la función ***subset***.

Aquí se presenta un esquema del algoritmo Apriori:

1) L_1 = {1-itemsets frecuentes}
2) **For** (k=2, $L_{k-1} \neq \emptyset$, k++) **do begin**
3) C_k = apriori-gen(L_{k-1}) //nuevos itemsets candidatos
4) **forall** transacciones $t \in D$ **do begin**
5) C_t = subset(C_k, t) //candidatos contenidos en t
6) **forall** itemsets candidatos $c \in C_k$ **do**
7) c.count++
8) end
9) Lk = { $c \in C_k$ | c.count ≥ minSup }
10) end
11) Answer = $\cup_k L_k$

Generación de los candidatos Apriori

La función ***apriori-gen*** toma como argumento L_{k-1}, el conjunto de todos los (k-1)-itemsets frecuentes y retorna un super conjunto conteniendo todos los k-itemsets frecuentes.La función realiza dos pasos para cumplir con su cometido.

En primer lugar, se realiza el *paso de join*. Este paso recibe el conjunto de itemsets L_{k-1} y retorna un super conjunto conteniendo todos los k-itemsets, generados a partir del conjunto de itemsets L_{k-1}, de la siguiente manera:

Insert into C_k
Select p.item$_1$, p.item$_2$, ..., p.item$_{k-1}$, q.item$_{k-1}$
From L_{k-1} p, L_{k-1} q
Where
p.item$_1$ = q.item$_1$ AND
p.item$_2$ = q.item$_2$ AND
...
p.item$_{k-2}$ = q.item$_{k-2}$ AND
p.item$_{k-1}$ < q.item$_{k-1}$

El siguiente paso es el *paso de prune*. En este paso se borran todos los itemsets $c \in C_k$ tal que algún (k-1) subconjunto de c no esté en L_{k-1}:

Forall itemsets $c \in C_k$ **do**
  Forall (k-1)-subsets s de c **do**
    If ($s \notin L_{k-1}$) **then**
      Delete c **from** C_k

Veamos un ejemplo para clarificar estos conceptos.
Sea L_3 = {{1 2 3}, {1 2 4}, {1 3 4}, {1 3 5}, {2 3 4}}. Luego del paso de *Join* tenemos C_4 = {{1 2 3 4}, {1 3 4 5}}.
En el paso de *Prune* borramos el itemset {1 3 4 5} porque el itemset {1 4 5} no está en L_3. De esta manera, tenemos solamente {1 2 3 4} en C_4.

Es necesario mostrar que $Ck \supseteq Lk$. Para demostrar esta suposición basta con ver que un subconjunto de un itemset frecuente debe también tener mínimo soporte.
La condición $p.item_{k-1} < q.item_{k-1}$ simplemente asegura que no se van a generar itemsets candidatos duplicados.

Los autores proponen una variante en la forma en que el soporte de los candidatos es contado, ellos proponen *"contar el soporte de los candidatos de múltiples tamaños en la misma pasada"*.
Es decir que en lugar de contar sólo el soporte de los candidatos de tamaño *k* en la pasada k-ésima, es posible también contar el soporte de los candidatos C'_{k+1}, donde C'_{k+1} es generado a partir de C_k.
Esta variación puede resultar muy beneficiosa en los pasos posteriores cuando el costo de contar y guardar en memoria adicional los candidatos C'_{k+1} - C_{k+1} se vuelve menor que el costo de recorrer completamente la base de datos.

La función ***subset***
Los itemsets candidatos C_k son almacenados en un *hash-tree*.
Un nodo de un *hash-tree* contiene una lista de items (nodo hoja) o una *hash-table* (nodo interior).
En un nodo interior, cada bitácora de la *hash-table* apunta a otro nodo.
La raíz del *hash-tree* está definida como de profundidad 1. Un nodo interior de profundidad *d* apunta a nodos de profundidad *d+1*.
Los itemsets son almacenados en las hojas.

Cuando se agrega un itemset *c*, se comienza desde la raíz y se desciende el árbol hasta alcanzar una hoja.
En un nodo interior de profundidad d, se decide cual de las ramas se elige aplicando una función de hash al d-ésimo item del itemset *c*.
Cuando el límite de itemsets contenidos en una hoja excede el umbral permitido, el nodo hoja se convierte en un nodo interior.

Comenzando de la raíz, la función *subset* busca todos los candidatos contenidos en una transacción *t*, de la siguiente manera:

- Si está en un nodo hoja, busca cuáles de los itemsets de la hoja están contenidos en *t*, y agrega referencias a ellos en el conjunto resultado.
- Si está en un nodo interior, que fue alcanzado aplicando la función de hash al item *i*, entonces recursivamente aplica la función de hash sobre cada item que viene después de *i* en *t* al nodo en la correspondiente bitácora.

Algoritmo AprioriTid

El algoritmo AprioriTid también usa la función *apriori-gen* para determinar los itemsets candidatos antes de comenzar la pasada.
La característica interesante de este algoritmo es que la base de datos **d** no es usada para contar el soporte de los itemsets candidatos después de la primer pasada. En su lugar, el conjunto $\cap_k$ es usado para este propósito.
Cada miembro del conjunto $\cap_k$ es de la forma < TID, $\{X_k\}$> donde cada X_k es un k-itemset potencialmente frecuente, presente en la transacción con identificador TID.
Para k=1, $\cap_1$ corresponde a la base de datos **d**.
Para k>1, $\cap_k$ es generado por el algoritmo (ver paso 10 del algoritmo).
El miembro de $\cap_k$ correspondiente a la transacción *t* es:

$$< t.\text{TID}, \{c\ C_k \mid c \text{ está contenido en } t\}>.$$

Si una transacción no contiene algún k-itemset candidato, entonces $\cap_k$ no tendrá una entrada para esta transacción.

A continuación, se presenta un esquema del algoritmo AprioriTid:

1) L_1 = {1-itemsets frecuentes}
2) $\cap_1$ = base de datos *D*
3) **For** (k=2, $L_{k-1} \neq \varnothing$, k++) **do begin**
4) C_k = apriori-gen(L_{k-1}) //nuevos itemsets candidatos
5) $\cap_k = \varnothing$
6) **forall** transacciones $t \in \cap_{k-1}$ **do begin**
7) //determinar los itemsets candidatos en C_k contenidos
//en la transacción con identificador *t*.TID
$C_t = \{ c \in C_k \mid (c - c[k]) \in t.\text{set-of-itemsets y}$
$(c - c[k-1]) \in t.\text{set-of-itemsets} \}$
8) **forall** itemsets candidatos $c \in C_t$ **do**
9) *c*.count++
10) **if** ($C_t \neq \varnothing$) **then** $\cap_k$ += <*t*.TID, C_t >
11) end
12) Lk = { $c \in C_k \mid c.\text{count} \geq \text{minSup}$ }
13) end
14) Answer = $\cup_k L_k$

Consideremos la base de datos de la **Figura 6** y asumamos que el mínimo soporte es igual a 2 transacciones.

Invocando a la función ***apriori-gen*** con L_1 en el *paso 4* obtenemos los itemsets candidatos C_2.
Desde el *paso 6* hasta el *paso 10*, contamos el soporte de los candidatos en C_2 iterando sobre las entradas en $\cap_1$ y generamos $\cap_2$.
La primer entrada en $\cap_1$ es { {1} {3} {4} }, correspondiente a la transacción con TID 100.
El conjunto C_t, en el *paso 7* correspondiente a esta entrada *t* es { {1 3} }, porque { { 1 3} } es un miembro de C_2 y ambos, ({1 3} – {1}) y ({1 3} – {3}) son miembros de *t*.set-of-itemsets.
Invocando a la función ***apriori-gen*** con L_2 obtenemos C_3.
Haciendo una pasada sobre los datos con $\cap_2$ y C_3, generamos $\cap_3$.
Debemos notar que no hay entradas en $\cap_3$ para las transacciones con TIDs 100 y 400, debido a que no contienen alguno de los itemsets en C_3.
El candidato {2 3 5} en C_3 se transforma en frecuente y es el único miembro de L_3.
Cuando generamos C_4 utilizando L_3, C_4 se transforma en vacío y el proceso se da por finalizado.

Database

TID	Items
100	1 3 4
200	2 3 5
300	1 2 3 5
400	2 5

$\bar{C}_1$

TID	Conjunto de Itemsets
100	{ {1}, {3}, {4} }
200	{ {2}, {3}, {5} }
300	{ {1}, {2}, {3}, {5} }
400	{ {2}, {5} }

L_1

Itemset	Soporte
{1}	2
{2}	3
{3}	3
{5}	3

C_2

Itemset	Soporte
{1 2}	1
{1 3}	2
{1 5}	1
{2 3}	2
{2 5}	3
{3 5}	2

$\bar{C}_2$

TID	Conjunto de Itemsets
100	{ {1 3} }
200	{ {2 3}, {2 5}, {3 5} }
300	{ {1 2}, {1 3}, {1 5}, {2 3}, {2 5}, {3 5} }
400	{ {2}, {5} }

L_2

Itemset	Soporte
{1 3}	2
{2 3}	2
{2 5}	3
{3 5}	2

C_3

Itemset	Soporte
{2 3 5}	2

$\bar{C}_3$

TID	Conjunto de Itemsets
200	{ {2 3 5} }
300	{ {2 3 5} }

L_3

Itemset	Soporte
{ {2 3 5} }	2

Figura 6: Ejemplo

Para el correcto funcionamiento del algoritmo, es necesario contar con una estructura de datos, con características particulares, que permita llevar a cabo las siguientes funciones:

- Asignación de un número único a cada itemset candidato, llamado su ID.
- Inclusión de cada conjunto de itemsets candidatos C_k en un vector indexado por los IDs de los itemsets en C_k. Un miembro de C_k es de la forma < TID, {ID} >.
- Almacenamiento de cada $\cap_k$ en una estructura secuencial.

La función *apriori-gen* genera un k-itemset candidato c_k uniendo dos (k-1)-itemsets frecuentes.
Además, la estructura contiene dos campos adicionales: *generadores* y *extensiones*. El campo generadores de un itemset candidato c_k almacena los IDs de dos (k-1)-itemsets candidatos cuyo *join* generó c_k. El campo extensiones de un itemset c_k almacena los IDs de todos los (k+1)-candidatos que son extensiones de c_k.
De esta manera, cuando un candidato c_k es generado uniendo l^1_{k-1} y l^2_{k-1}, se graban los IDs en el campo generadores de c_k.
Al mismo tiempo, el ID de c_k es agregado al campo extensiones de l^1_{k-1}.

Algoritmo D.I.C. (Dynamic Itemset Counting)

Muchos investigadores están enfocados en derivar algoritmos eficientes para encontrar itemsets frecuentes. El algoritmo mejor conocido para esta tarea es *Apriori* [12], el cual, como todos los algoritmos para encontrar los itemsets frecuentes, cuenta con la propiedad que un itemset únicamente puede ser frecuente si y sólo si todos sus subconjuntos son frecuentes.

Como vimos, primero cuenta todos los *1-itemsets* y determina aquellos cuyo contador excede el mínimo soporte, dando lugar a los *1-itemsets frecuentes*. Luego, los combina para formar los *2-itemsets candidatos* (potenciales frecuentes), los cuenta y determina cuales de ellos son *2-itemsets frecuentes*. Y continúa combinando los *2-itemsets frecuentes* para formar los *3-itemsets candidatos*, los cuenta para determinar cuales son *3-itemsets frecuentes*, y así sucesivamente.
De esta manera, el algoritmo desarrolla tantos pasos sobre los datos como el número máximo de elementos que existe en un itemset candidato, chequeando en la pasada *k* el soporte para cada uno de los candidatos en C_k.

Los dos factores importantes que gobiernan la performance de este algoritmo son: el número de pasadas hechas sobre los datos y la eficiencia de las pasadas.

Para corregir ambos problemas los autores introducen el concepto Dynamic Itemset Counting (DIC) [3], dando lugar al algoritmo con igual nombre.

Este algoritmo reduce el número de pasadas hechas sobre los datos mientras guarda el número de itemsets, los cuales son contados en cualquier paso relativamente inferior comparado con los métodos basados en "prueba".

La idea intuitiva detrás de DIC es que el algoritmo trabaja como un *tren* (*train*) corriendo sobre los datos, el cual se detiene a intervalos de M (es un parámetro definido antes de comenzar el proceso de los datos) transacciones separadas.
Cuando el *tren* alcanza el final del archivo de transacciones, decimos que tiene hecha una pasada sobre los datos y comienza de nuevo desde el inicio para la siguiente pasada.
Los *"pasajeros"* del *tren* son los itemsets. Cuando un itemset está sobre el *tren*, se cuentan sus ocurrencias en las transacciones que son leídas.

Comparando esta metáfora con Apriori, todos los itemsets deben "subir" al tren al inicio de una pasada y "bajar" al final.
Los *1-itemsets* toman la primer pasada, los *2-itemsets* toman la segunda pasada, y así sucesivamente.

En DIC, la idea fue agregar flexibilidad para permitir a los pasajeros "subirse" en cualquier "parada" con tal de que ellos se bajen en la misma "parada" la próxima vez que el tren pasa por dicha "parada".
Por consiguiente, el itemset ha "visto" todas las transacciones en el archivo.
Esto significa que puede comenzar a contarse un itemset puesto que se sospecha que puede ser necesario contarlo en lugar de esperar hasta el final del archivo de la pasada anterior.

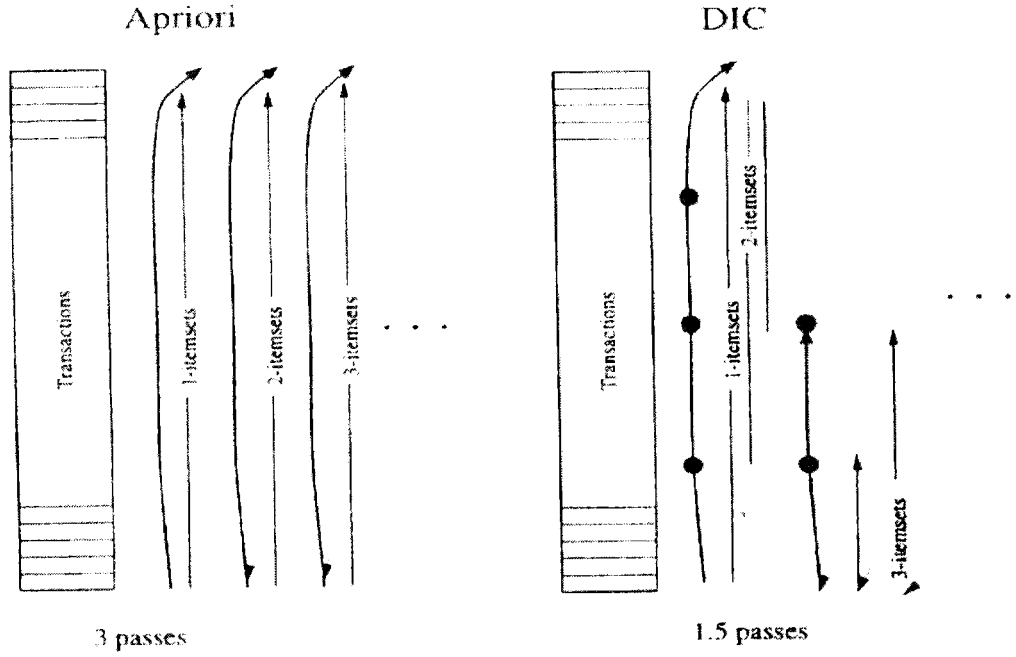

Figura 7
Comparación de las pasadas realizadas
por los algoritmos Apriori y DIC

Para contar los itemsets frecuentes, supongamos que los itemsets forman un grafo con el itemset vacío en la parte inferior, y el conjunto de todos los items en la parte superior. Algunos items son frecuentes (denotados por cajas), y el resto son no frecuentes.
La estructura del grafo es similar a la que se muestra en la **Figura 8.1**.

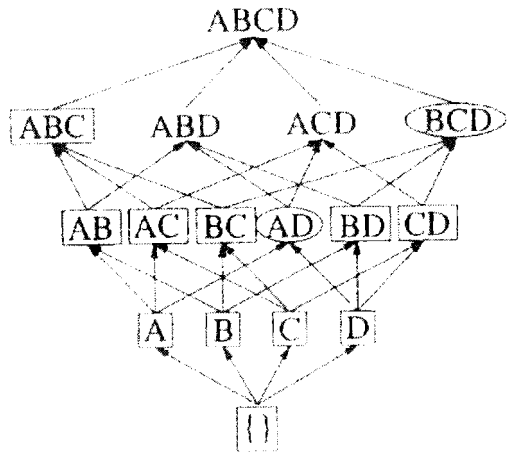

Figura 8.1
Grafo de itemsets

Para mostrar que los itemsets son frecuentes es necesario poder contarlos. De hecho, debemos contarlos, dado que generalmente queremos conocer las cantidades (ocurrencias) de cada uno de ellos.
Sin embargo, no es factible contar todos los itemsets que son no frecuentes. Afortunadamente, es suficiente contar simplemente el mínimo (los itemsets que no incluyen algún otro itemset no frecuente) dado que si un itemset es no frecuente, todos sus superconjuntos son no frecuentes también.

Los itemsets no frecuentes minimales son denotados por círculos. Ellos forman la parte superior de la ligadura entre los itemsets frecuentes y los no frecuentes.

Un algoritmo que cuenta todos los itemsets frecuentes debe encontrar y contar todos los itemsets frecuentes y los itemsets no frecuentes minimales (esto es, todas las cajas y círculos).

El algoritmo DIC, marca los itemsets de 4 maneras diferentes:
- *Solid box*: itemset frecuente confirmado (itemset que finalizó el conteo, que excede el umbral del soporte).
- *Solid circle*: itemset no frecuente confirmado (itemset que finalizó el conteo, que está por debajo del umbral del soporte).
- *Dashed box*: itemset frecuente sospechado (itemset que se está aún contando, que excede el umbral del soporte).
- *Dashed circle*: itemset no frecuente sospechado (itemset que se está aún contando, que está por debajo del umbral del soporte).

El algoritmo DIC trabaja como sigue:

1. El itemset vacío es marcado con un *solid box*.
   Todos los 1-itemsets son marcados con *dashed circles*. Todos los demás itemsets son desmarcados. (Ver **Figura 8.2**)
2. Leer M transacciones. Por cada transacción, incrementar los respectivos contadores de los itemsets marcados con *dashes*.
3. Si un *dased circle* tiene un contador que excede el umbral del soporte, transformarlo en un *dashed box*.
   Si algún subconjunto inmediato del itemset tiene todos sus subconjuntos como *solid box* o *dashed box*, agregar un nuevo contador para el itemset y marcarlo con un *dashed circle*. (Ver **Figura 8.3** y **Figura 8.4**)
4. Si un *dashed* itemset ha sido contado pasando por todas las transacciones, entonces marcarlo como *solid* y detener su conteo.
5. Si estamos al final del archivo de transacciones, retomar la ejecución del algoritmo desde el principio. (Ver **Figura 8.5**)
6. Si algún *dashed itemset* permanece, ir al paso 2.

De esta forma DIC comienza contando sólo los 1-itemsets y agrega rápidamente contadores para los 2, 3, 4,..., k-itemsets.
Luego de unos pocos pasos sobre los datos (usualmente menos de dos para valores pequeños de M), finaliza contando todos los itemsets.

En una primera aproximación, podríamos querer que M sea tan pequeño como sea posible, para empezar contando los itemsets muy tempranamente en el paso 3.
Sin embargo, los pasos 3 y 4 pueden incurrir en un considerable *overhead*, por lo tanto los autores aconsejan no reducir M por debajo de 100. [3]

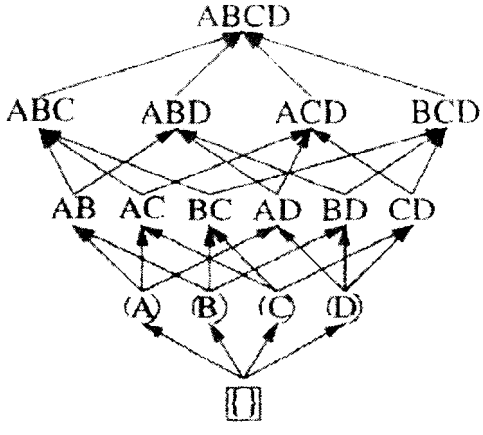

Figura 8.2
Inicio del algoritmo DIC

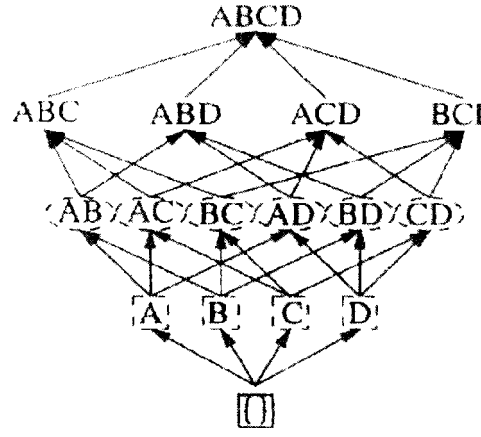

Figura 8.3
Después de *M* transacciones

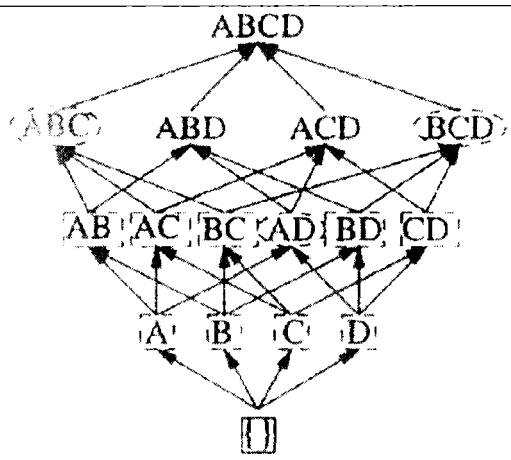

Figura 8.4
Después de $2M$ transacciones

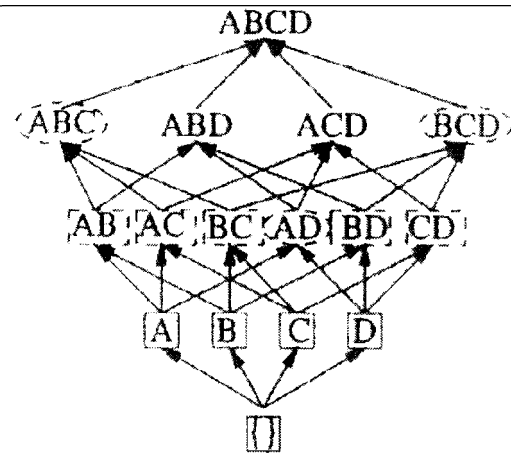

Figura 8.5
Después de una pasada

La implementación del algoritmo DIC requiere de una estructura de datos que le permita guardar la historia de muchos itemsets a la vez. Además, debe soportar las siguientes operaciones:

1. Agregar nuevos itemsets.
2. Mantener un contador para todos los itemsets.
3. Mantener los estados de los itemsets para la transición del estado "*activas*" al estado "*contadas*" (transformación de *dashed* a *solid*) y desde no frecuente a frecuente (transformación de *circle* a *box*).
4. Detectar cuando las transiciones enumeradas en el punto 3 pueden ocurrir.
5. Cuando los itemsets se transforman el frecuentes, determinar qué nuevos itemsets deben ser agregados como *dashed circle* dado que ahora potencialmente son frecuentes.

La estructura de datos usada es exactamente como el *hash tree* (árbol de hash) usado en Apriori con una pequeña información extra almacenada en cada nodo.
La estructura en forma de *tree* tiene las siguientes propiedades:

- Cada itemset está ordenado por sus items.
- Todos los itemsets se están contando, o ya han sido contados y tienen un nodo asociado, como un prefijo.
- El itemset vacío es el nodo raíz.
- Todos los 1-itemsets están vinculados al nodo raíz, y sus ramas son etiquetadas por el nodo que representan. Todos los otros itemsets son vinculados a sus prefijos conteniendo todos menos su último item; los cuales son etiquetados por su último item.
- Todos los nodos almacenan el último item del itemset que representan, un contador, una marca para indicar en qué lugar del archivo se empezó a contar, su estado, y sus ramas si es un nodo interior.

Hay dos importantes beneficios del algoritmo DIC. El principal es la performance.
Si los datos son bastante homogéneos a lo largo del archivo y el intervalo M es razonablemente pequeño, el algoritmo generalmente desarrolla un orden de dos pasadas. Esta característica hace al algoritmo más rápido que Apriori el cual hace tantas pasadas como el máximo tamaño de un itemset candidato.

Por el contrario, si los datos no son bastante homogéneos, el recorrer el archivo puede llevar un orden random.
Además de la performance, DIC provee considerable flexibilidad porque tiene la habilidad de agregar y eliminar itemsets a medida que los procesa.

Una debilidad de DIC es que el algoritmo es sensible a la homogeneidad de los datos. En particular, si los datos son muy correlativos, al algoritmo podría no darse cuenta de que un itemset es frecuente hasta que es contado en la mayoría de la base de datos.
Si esto ocurre, no podríamos cambiar la hipotética frontera y comenzar contando alguno de los super conjuntos de itemsets hasta que tenemos casi terminado el conteo del itemset en toda la base de datos.

Algoritmo T.D.I.C. (Temporal Dynamic Itemset Counting)

Como vimos, el objetivo del descubrimiento de las reglas de asociación es descubrir todos las posibles asociaciones que cumplen ciertas restricciones, como pueden ser: mínimo soporte, confianza e interés.

Sin embargo, es posible descubrir asociaciones interesantes con un alto nivel de confianza, pero con pequeño soporte.
Este problema es causado por la forma en que el soporte es calculado (el denominador representa el número total de transacciones en el período completo de tiempo de la base de datos, cuando los items involucrados podrían no existir en dicho período).
Para evitarlo, los autores limitan el total de transacciones a aquellas que pertenecen al período de vida del itemset, estas asociaciones pueden ser ahora descubiertas, ya que pueden contar con suficiente soporte.

Otra dificultad es el gran número de reglas que podrían ser generadas, para lo cual se utiliza un indicador de la edad de la regla como factor para reducir el número de reglas que son presentadas al usuario. [2]

El problema del descubrimiento de las reglas de asociación viene de la necesidad de descubrir patrones utilizando la información de cada una de las transacciones (compra y venta) en un supermercado.
Dado que las transacciones son temporales, se espera encontrar patrones que dependen del tiempo.

En grandes volúmenes de datos, como los usados para propósitos de data mining, podemos encontrar información relacionada a productos que no necesariamente existen a lo largo de todo el período de recolección de los datos.
De esta manera, podemos encontrar productos que, en el momento de llevar a cabo el minado han sido discontinuados.
Puede haber también nuevos productos que fueron introducidos después de que comenzó la recolección. Algunos de estos productos podrían participar en las asociaciones, pero no pueden ser incluidos en ninguna regla por las restricciones de soporte.

Para ejemplificar, supongamos que el número total de transacciones de la base de datos es 30.000.000 y que fijamos un mínimo soporte de 0,5 %.
Entonces, un producto particular debe aparecer en, al menos, 150.000 transacciones para que sea considerado frecuente.
Sin embargo, supongamos que estas transacciones fueron registradas durante los últimos 30 meses, a razón de 1.000.000 por mes.
Ahora, si tomamos un producto que ha sido vendido durante los últimos 30 meses y tiene mínimo soporte, tenemos que aparece un promedio de 5.000 transacciones por mes.
Consideremos ahora otro producto que fue incorporado en los últimos 6 meses y que aparece en 20.000 transacciones por mes.
El número total de transacciones en las cuales aparece es 120.000, por ésta razón, no es frecuente, aunque sea 4 veces más popular que el primero. Sin embargo, si consideramos sólo las transacciones generadas desde que el producto apareció en el mercado, su soporte puede ser superior que el mínimo estipulado.
En nuestro ejemplo, el soporte del producto sería de 2 %, relativo a su período de vida (*lifespan)* dado que en 6 meses el total de transacciones sería 6.000.000 y el producto aparece en 120.000 de ellas.
Por consiguiente, estos nuevos productos aparecerían en reglas de asociación interesantes y potencialmente útiles.

Una forma de resolver este problema es incorporando el "*tiempo*" en el modelo de descubrimiento de las reglas de asociación. Estas reglas son llamadas "*Reglas de Asociación Temporal*".

De esta manera, sería posible considerar el caso de algunos productos que pueden ser frecuentes sólo en algún subintervalo estrictamente contenido en su período de vida, pero no en el intervalo completo correspondiente a su lifespan.
Otra característica de esta idea es que puede ser usada para eliminar reglas desactualizadas, de acuerdo al criterio definido por el usuario.
Más aún, también sería posible eliminar itemsets obsoletos como una función del período de vida, reduciendo la cantidad de trabajo utilizada en la determinación de los itemsets frecuentes y en la generación de las reglas de asociación.

Las reglas de asociación temporales introducidas en [2] son una extensión del modelo no temporal.

La idea básica es limitar la búsqueda de los itemsets al período de vida de los miembros del itemset. Así, cada regla tiene un *frame* de tiempo asociado, denominado *lifetime*, correspondiente al período de tiempo de los items que participan en la regla.
Si la magnitud del *lifetime* de una regla excede un mínimo estipulado por el usuario, entonces analizamos si la regla es frecuente en ese período.
Este concepto permite encontrar reglas que, con el punto de vista tradicional, no sería posible descubrir.

El desarrollo propuesto está basado en tener en cuenta el período de vida (*lifespan*) del itemset, es decir el período entre la primera y la última vez que el item aparece en las transacciones de la base de datos.
Por esta razón, el concepto de *soporte temporal* es introducido.

El soporte de un itemset es computado en el intervalo definido por su lifespan y se define el soporte temporal como el ancho del intervalo mínimo. [2]

El modelo temporal - Definiciones

Sea $T = \{..., t_0, t_1, t_2, ...\}$ un conjunto de tiempos, contablemente finito, sobre el cual un orden lineal $<_T$ está definido, donde $t_i <_T t_j$ significa que t_i ocurre antes o es anterior a t_j.

Sea $\mathbf{R} = \{A_1, ..., A_p\}$ donde los A_i son llamados items y **d** es una colección de subconjuntos de **R**, llamado la base de datos de transacciones.
Cada transacción **s** en **d** es un conjunto de items tal que $\mathbf{s} \subseteq \mathbf{R}$.
La definición de **R** incluye todos los items de **d**, independientemente del momento en el cual aparecen.
Asociado a **s** tenemos un *timestamp* t_s que representa el tiempo real de la transacción **s**.

Como ejemplo, supongamos que **R** = {A, B, C, D, E, F, G, H, I, W, X} y **d** es la base de datos de 12 transacciones con identificadores 100, ..., 1200 y timestamps 1, .., 12, tal y como se detalla a continuación:

IDs	TIDs	Items
1	100	A, B, C, F, H, I, X
2	200	A, B, C, G, X
3	300	C, D, I
4	400	A, C, I
5	500	D, E, H, I
6	600	A, F, G
7	700	B, C, I
8	800	C, H
9	900	B, E, G
10	1000	A, C, D, W
11	1100	D, F, H, I, X
12	1200	A, E, G, I, X

Figura 9: Ejemplo

Supongamos también que tenemos los siguientes valores $\tau = 4$ y $\sigma = 0.4$, para el soporte temporal y el soporte de las transacciones, respectivamente.

Todos los items tienen un lifespan en la base de datos, el cual explícitamente representa la duración temporal de la información del item. El lifespan de un item A_i está dado por el intervalo $[t_1, t_2]$, con $t_1 < t_2$.
Sea A_i un item de R, entonces con cada item A_i y la base de datos **d**, asociamos un lifespan definido por un intervalo de tiempo $[A_i.t_1, A_i.t_2]$ o simplemente $[t_1, t_2]$ donde A_i está comprendido.
Llamaremos al lifespan del itemset A_i como l_{Ai}.

Sea $\mathbf{X} \subseteq \mathbf{R}$ un conjunto de items, decimos que **s** *contiene a* **X**, o que **X** *está verificado en* **s**, si $\mathbf{X} \subseteq \mathbf{s}$.
El conjunto de transacciones en **d** que contienen a **X** es indicado por

$$\mathbf{V(X, d) = \{ s \mid s \in d \text{ y } X \subseteq s \}}.$$

Si la cardinalidad de **X** es k, entonces **X** es llamado k-itemset.
El lifespan de un k-itemset X, con k > 1, es [t, t'] donde

$t = \max \{ t_1 \mid [t_1, t_2]$ es el lifespan de un item A_i en X $\}$ y
$t' = \min \{ t_2 \mid [t_1, t_2]$ es el lifespan de un item A_i en X $\}$

Sea $\mathbf{X} \subseteq \mathbf{R}$ un conjunto de items y l_x su lifespan. Si **d** es el conjunto de transacciones de la base de datos, entonces d_{lx} es el conjunto de transacciones de **d** cuyos timestamps $t_i \in l_x$.
Por $|d_{lx}|$ indicamos el número de transacciones de d_{lx}.

La incorporación del tiempo permite determinar si un itemset es frecuente, computando la proporción entre el número de transacciones que contienen el itemset y el número de transacciones en la base de datos, tal que su *timestamp* (tiempo de validez) está incluido en el lifespan del itemset.

Evidentemente, es necesario filtrar items, y por consiguiente los itemsets con un período de vida muy corto, tienden a no ser tenidos en cuenta para el proceso.
Por esta razón, se define el *soporte temporal* como la amplitud del lifespan de un itemset.
También se define un *umbral* para el soporte temporal como sigue: si l_d es el lifespan de la base de datos y $|l_d|$ es su duración, entonces el umbral del soporte temporal τ es una fracción de $|l_d|$.

El soporte de ***X*** en **d** sobre su lifespan l_x, denotado por $s(\boldsymbol{X}, l_x, \mathbf{d})$, es la fracción de transacciones en **d** que contienen a ***X*** durante el intervalo de tiempo correspondiente a l_x: $|\mathbf{V}(\boldsymbol{X}, \mathbf{d})| / |dl_x|$.
La frecuencia de un conjunto ***X*** es su soporte.

Dado un umbral del soporte $\sigma \in [0,1]$ y el umbral del soporte temporal τ, decimos que ***X*** es *frecuente* en su lifespan l_x si $s(\boldsymbol{X}, l_x, \mathbf{d}) > \sigma$ y $|dl_x| >= \tau$.
En este caso, decimos que ***X*** tiene *mínimo soporte* en l_x.

Del ejemplo de la **Figura 9**, tenemos que el 1-itemset candidato {A} está definido en el intervalo [100,1200].
Además, vemos que aparece en las transacciones 1, 2, 4, 6, 10 y 12, es decir que ocurre 6 veces en la base de datos.
Si contamos la cantidad de transacciones que ocurren en el intervalo de vida del itemset {A}, tenemos un total de 12 transacciones.
De los valores anteriores, tenemos que el itemset {A} tiene un soporte igual a 0.5, ya que 0.5 es el resultado de dividir 6 por 12.
Por otro lado, tenemos que el soporte temporal del itemset {A}, calculado como la amplitud del intervalo de vida, es 12.

Con estos resultados, llegamos a la conclusión que el itemset {A} es frecuente en el intervalo de vida [100,1200] y tiene un soporte igual a 0.5 y un soporte temporal igual a 12.
En este caso, decimos que el itemset tiene un intervalo de frecuencia [100,1200].

Tomando otro caso, tenemos el 1-itemset candidato {C}, el cual está definido en el intervalo de vida [100,1000]. El itemset ocurre en las transacciones 1, 2, 3, 4, 7, 8 y 10, es decir, ocurre en 7 oportunidades.
Luego, vemos que hay definido un total de 10 transacciones en la base de datos en dicho intervalo. Entonces, el cálculo del soporte para el itemset {C} nos da 0.7, es decir 7 sobre 10.
Como vimos el soporte temporal del itemset {C} en su intervalo de vida es igual a la amplitud de dicho intervalo, es decir, 10 unidades.
Con estos resultados, llegamos a la conclusión que el itemset {C} es frecuente en el intervalo [100,1000] y tiene un soporte igual a 0.7 y un soporte temporal igual a 10.

Luego, con estos dos itemsets frecuentes encontrados, generamos el 2-itemset candidato {A, C}.
Para calcular el intervalo de vida de este itemset, tomamos la intersección de los intervalos de frecuencia de los itemsets que lo componen, es decir, [100,1200] y [100, 1000], respectivamente; dándonos como intervalo de vida para el itemset {A, C}el intervalo [100, 1000].
Luego, tenemos que este itemset ocurre en las transacciones 1, 2, 4 y 10. Por lo tanto, ocurre 4 veces. Además, tenemos un total de 10 transacciones definidas en la base de datos en el intervalo de vida de itemset.
Por lo tanto, el soporte para el itemset es igual a 0.4, ya que es el valor de dividir 4 sobre 10.
El soporte temporal del itemset es igual a 10, que la amplitud del intervalo [100, 1000] en donde el itemset tiene validez.
De lo anterior, tenemos que el itemset {A, C} es frecuente en [100, 1000], con los valores 0.4 y 10, para el soporte y el soporte temporal, respectivamente.

Una regla de asociación temporal expresa que un conjunto de items tiende a aparecer junto con otro conjunto de items en la misma transacción, en un frame de tiempo específico.
Una *Regla de Asociación Temporal* para **d** es una expresión de la forma $\mathbf{X} \Rightarrow \mathbf{Y}\ [t_1, t_2]$ donde $\mathbf{X} \subseteq \mathbf{R}$, $\mathbf{Y} \subseteq \mathbf{R} \setminus \mathbf{X}$ y $[t_1, t_2]$ es un intervalo de tiempo correspondiente al lifespan de $\mathbf{X} \cup \mathbf{Y}$ expresado en una granularidad determinada por el usuario.

Una regla de asociación temporal tiene tres factores asociados: *soporte*, *soporte temporal*, los cuales ya han sido detallados, y *confianza*.
La confianza de una regla $\mathbf{X} \Rightarrow \mathbf{Y}\ [t_1, t_2]$, denotada por $conf(\mathbf{X} \Rightarrow \mathbf{Y}, [t_1, t_2], \mathbf{d})$ es la probabilidad condicional que una transacción en **d**, seleccionada al azar en el intervalo de tiempo $[t_1, t_2]$, que contiene **X** también contiene **Y**.
La probabilidad condicional varía de acuerdo al subintervalo considerado dentro de fl_{XUY}.

Entonces, podemos expresarlo como:

$$conf(\boldsymbol{X} \Rightarrow \boldsymbol{Y}: fl_{X \cup Y}) = \{ s(\boldsymbol{X} \cup \boldsymbol{Y}, [t, t']) / s(\boldsymbol{X}, [t, t']) \mid [t, t'] \subseteq fl_{X \cup Y} \}$$

donde

$$fl_{X \cup Y} = \{ [t_1, t_2] / \mid [t_1, t_2] \mid \geq \tau \text{ y } s(\boldsymbol{X} \cup \boldsymbol{Y}, [t_1, t_2]) \geq \sigma \text{ y } \neg\exists [tj, t_k] ([t_1, t_2] \subset [t_j, t_k] \text{ y } s(\boldsymbol{X} \cup \boldsymbol{Y}, [t_j, t_k]) \geq \sigma) \}$$

es decir que $[t_1, t_2]$ es el *intervalo frecuente maximal* de $fl_{X \cup Y}$.

Una regla de asociación temporal X => Y : $fl_{X \cup Y}$ está contenida en **d** con

soporte $s_1, s_2, \ldots, s_p$

soporte temporal $|fl_{X \cup Y}|$, y

confianza $c_1, c_2, \ldots, c_p$

si

$s_1\%, s_2\%, \ldots, s_p\%$ de las transacciones de **d** en $fl_{X \cup Y} = \{\text{subinterval}_1, \text{subinterval}_2, \ldots, \text{subinterval}_p\}$contienen **X** U **Y** y

$c_1\%, c_2\%, \ldots, c_p\%$ de las transacciones de **d** que contienen **X** también contienen **Y**, en el conjunto de intervalos de tiempo [t, t'] tal que $[t, t'] \subseteq fl_{X \cup Y}$.

Retomando con el ejemplo de la **Figura 9**, tenemos los 1-itemsets frecuentes {A} y {C}, y el 2-itemset frecuente {A, C}; con los valores de soporte y de soporte temporal calculados como vimos.
Con estos itemsets frecuentes, generamos la regla de asociación temporal A => C, definida en el intervalo [100,1000].
La confianza de la regla de asociación temporal A => C: [100,1000] es igual al cociente entre el soporte del 2-itemset frecuente {A, C} en el intervalo [100,1000], y el soporte del 1-itemset frecuente {A} en el intervalo [100,1000].
De los resultados obtenidos anteriormente, tenemos que el soporte del 2-itemset {A, C} en el intervalo [100,1000] es 0.4.
Luego, debemos calcular el soporte del 1-itemset frecuente {A}en el intervalo de la regla de asociación. Analizando el conjunto de transacciones, vemos que el valor de soporte para el itemset {A}en [100,1000] es 0.5.
Con estos valores, tenemos que la confianza de la regla de asociación temporal A => C en el intervalo [100,1000] es 0.8 (0.8 = 0.4 / 0.5).

Caracterización Temporal de los Itemsets

Asociado a cada itemset tenemos un *histograma* el cual se construirá durante el proceso de búsqueda de los itemsets frecuentes. En el histograma se acumulará el número de transacciones en las cuales el itemset aparece en su período de tiempo completo.
El ancho del intervalo para el histograma, llamado *unidad de tiempo*, es proporcionado por el usuario.

En el proceso de descubrimiento de los itemsets frecuentes tenemos dos casos.
En primer lugar, los itemsets que son frecuentes en todo su período de vida. En segundo lugar, los itemsets que no son frecuentes en todo su período de vida, pero sí en algún subintervalo maximal contenido en su período de vida.

Para determinar los itemsets que no son frecuentes a lo largo de todo su período de vida, pero sí en algún subintervalo maximal, utilizamos el función *a-posteriori*, la cual busca los intervalos frecuentes maximales contenidos en I_X, para un itemset X, es decir If_X.

El función *a-posteriori* recibe como entrada los siguientes valores:
- un vector *S* con información recogida en el histograma,
- un vector *Q* con el número total de transacciones por período de tiempo en que está divido el histograma,
- el umbral para el mínimo soporte (τ) y
- el umbral para el mínimo soporte temporal (σ).

Una vez finalizada su ejecución, la función retorna el conjunto de intervalos frecuente maximal, If_X.
Este conjunto puede ser vacío, en cuyo caso, se asume que el itemset no es frecuente en ningún intervalo maximal contenido en su lifespan.

El método consiste en recorrer S hacia atrás. Si se encuentra un intervalo satisfactorio, es decir, un subintervalo en el cual el itemset en cuestión es frecuente, se marca el final del intervalo con *h*, y se busca nuevamente desde el final hasta *h*, sino se avanza *h* una unidad y se repite el proceso.

En detalle, sea la unidad de tiempo *l* de S correspondiente con la unidad de tiempo *j* de Q, *sum1* es la suma de los intervalos del histograma, *sum2* es el número total de transacciones en el mismo período, y supongamos que el histograma tiene *n* unidades de tiempo, entonces:

```
1.  h = 1; g = 0;
2.  while n ≥ h do begin
3.      m = 0; g = g + 1;
4.      sum1 = Σ_{i=h to n} S(i);
5.      sum2 = Σ_{k=j+h-1 to j+n-1} Q(k);
6.      fr = sum1 / sum2;
7.      while fr < σ and (n -m) * |TimeUnit| ≥ τ do begin
8.          fr = (sum1 - S(n - m)) / (sum2 - Q(j + n - m))
9.          m = m + 1;
10.     endwhile
11.     if (n - m) * |TimeUnit| ≥ τ then begin
12.         hl = h; h = n - m; hs = h;
13.         Interval_g = <[lwr bound T. Unit j + hl - 1, upper bound T. Unit j + hs - 1], fr>;
14.         m = 0;
15.     else h = h + 1;
16. endwhile.
```

El histograma describe, hasta cierto punto, el comportamiento temporal de los itemsets. De hecho, podemos considerar el histograma de un itemset como series de tiempo, las cuales son patrones útiles para "minar".

Descubrimiento de Reglas Temporales

Dado un conjunto de transacciones **d**, y los niveles mínimos de soporte, soporte temporal y confianza; el problema de descubrir las reglas de asociación temporal es generar todas las reglas de asociación que cumplen con las restricciones de soporte, soporte temporal y confianza previamente especificados.

Existen unas pocas modificaciones convenientes para soportar el descubrimiento de las reglas de asociación temporal. A saber:

Fase 1T:

Encuentre todos los conjuntos de items (itemsets) $\mathbf{X} \subseteq \mathbf{R}$ tal que son frecuentes, es decir que su frecuencia excede el mínimo soporte σ establecido.

Fase 2T:

Use los itemsets frecuentes X para encontrar las reglas: verificar para cada $\mathbf{Y} \subseteq \mathbf{X}$, con $Y \neq \varnothing$, si la regla $X \setminus Y \Rightarrow Y$: $fl_{X \cup Y}$ es satisfecha con suficiente confianza, es decir que excede la mínima confianza θ establecida en el intervalo [t, t'] para todo [t, t'] en $fl_{X \cup Y}$.

Generación de los Itemsets Frecuentes

Cualquiera de los algoritmo propuestos en la literatura ([1], [3], [7], [11]) para el descubrimiento de las reglas de asociación, puede ser convenientemente modificado para satisfacer el descubrimiento de las reglas de asociación temporal.

En particular, se ha elegido el algoritmo D.I.C., Dynamic Itemset Counting, porque usualmente **d** es muy grande y este algoritmo optimiza el número de pasadas sobre los datos.
El algoritmo ha sido llamado *Temporal Dynamic Itemset Counting*.

La entrada del algoritmo es la base de datos de transacciones donde cada registro contiene una transacción, la cual tiene un identificador temporal asociado, el umbral para el mínimo soporte (τ), el umbral para el mínimo soporte temporal (σ), un valor *M* indicando la cantidad de transacciones de **d** que se leen juntas, y el umbral para los itemsets obsoletos (t_0).
La salida es el conjunto de cada itemset frecuente en el lifespan donde el itemset es frecuente (F).

Como en la notación original ([1], [11]), L_k representa el conjunto de los k-itemsets frecuentes. Cada miembro de L_k tiene asociado los siguientes campos:

i. Identificación del itemset.
ii. Límite inferior (t_1) y superior (t_2) del período de vida del itemset.
iii. Contador de soporte (Fr) en el intervalo $[t_1, t_2]$.
iv. Número total de transacciones (FTr) encontradas en el intervalo $[t_1, t_2]$.
v. Histograma del itemset, con el número de transacciones conteniendo el itemset en cada intervalo.

C_k es el conjunto de k-itemsets candidatos, es decir, itemsets potencialmente frecuentes que tienen asociado la misma información que los miembros de L_k.

Además, C_k usa un campo auxiliar para computar correctamente el número de transacciones en $[t_1, t_2]$ cuando t_2 no está definido todavía; y un marcador para indicar si el itemset completó una pasada o no.
TRAIN es el conjunto conteniendo cada itemset cuyo soporte estamos contando aún.

TDIC(d, M, F) comienza buscando los 1-itemsets y luego lee M transacciones, analizando si son frecuentes. Luego de esto, combina los 1-itemsets frecuentes para formar los 2-itemsets candidatos. Estos últimos son analizados hasta el final de porción siguiente de M transacciones intersecados con su lifespan, para verificar si ellos también son frecuentes, en cuyo caso, son combinados para formar los 3-itemsets, y así sucesivamente.
Cuando el algoritmo completa una pasada, verifica si durante la última pasada generó algún nuevo itemset de tamaño mayor que el existente al comienzo de la pasada, en cuyo caso, comienza nuevamente desde el principio de **d**.

Podría haber algunos itemsets no frecuentes en su período de vida completo, pero si en algún subintervalo contenido en su lifespan.
De la misma manera que ocurre en el algoritmo DIC, para estos casos se utiliza la función *a-posteriori*, la cual, dado un itemset X, nos permite encontrar todos los intervalos maximales contenidos en el lifespan de X, y generar los nuevos candidatos a partir de ellos.
Por otro lado, vemos que algunos itemsets podrían ser borrados de L_1 porque son obsoletos, ya que tienen lifespan $[t_1, t_2]$ y $t_2 < t_0$.

Los itemsets candidatos C_k de tamaño k, basados en los itemsets frecuentes L_{k-1} de tamaño k-1 obtenidos hasta el momento, son generados por medio de la función *apriori-gen*, con una estructura similar a la que definió anteriormente, pero con algunas ligeras modificaciones.

El lifespan de un k-itemset, con $k > 1$, es obtenido de la siguiente manera: si el k-itemset **u** es obtenido uniendo los (k-1)-itemsets **v** y **w**, entonces el lifespan de **u** es el conjunto de intervalos:

$$\{ [\mathbf{u}.t_1, \mathbf{u}.t_2] / (\exists\, [\mathbf{v}.t_1, \mathbf{v}.t_2], [\mathbf{w}.t_1, \mathbf{w}.t_2]) ([\mathbf{v}.t_1, \mathbf{v}.t_2] \cap [\mathbf{w}.t_1, \mathbf{w}.t_2]) = \varnothing$$
$$\text{y } \mathbf{u}.t_1 = \max \{ \mathbf{v}.t_1, \mathbf{w}.t_1 \} \text{ y } \mathbf{u}.t_2 = \min \{ \mathbf{v}.t_2, \mathbf{w}.t_2 \}) \}$$

Entonces, se lee la base de datos de transacciones para computar el soporte de los itemsets candidatos de TRAIN.
Para ello, la función *subset* es usada, puesto que determina si cada **c** miembro de TRAIN está contenido en la transacción **s**.
El timestamp t de **s** debe satisfacer $t \in l_c$.

En la **Figura 10** se muestra el pseudo código del algoritmo TDIC. Dicho código nos permite obtener una idea de la cantidad de pasadas que se realizan sobre las transacciones de la base de datos y la forma de computar el soporte de los itemsets candidatos para determinar los itemsets frecuentes.
Del mismo modo, podemos ver como, una vez finalizada la pasada, son eliminados los itemsets que recorrieron todas las transacciones de la base de datos y no son frecuentes.

```
pass ← 1; kf ← 0; k ← 1;
while kf <> k do begin
   kf = k
   while not end of d do begin
      read dM      /* dM is a group of M transactions of d ordered by their timestamp */
      foreach s ∈ dM do begin
         if pass = 1 then
         check if each 1-itemset c ∈ s exists in C1 otherwise insert c in C1 and TRAIN;
      Cs ← subset (TRAIN, s)   * Cs contains every itemset c in TRAIN such that appears
      foreach c ∈ Cs do begin
         c.Fr ← c.Fr + 1;
         if s.t  c.t2 then begin
            c.t2 ← s.t;
            FTr ← FTr + FTr';
            FTr' ← 0;
            endif;
         end;
      foreach c ∈ TRAIN do
         if s.t ∈ [c.t1, c.t2] then
            c.FTr' ← c.FTr'+ 1;
         else
                  if c is a 1-itemset add 1 to c.FTr';
         end;
      end;
      for j = pass to k do begin
      LL = { c   c ∈ Cj ∧ c.Fr ≥ σ × FTr ∧ (c.t2 - c.t1 + 1) ≥ τ};  * obtain the frequent
                                                                        j-itemsets */
         if |Lj| > 1 then
            Lj' ← LL - Lj
         else
            Lj' ← LL;   Lj' contains the new j-itemsets */
         Lj ← LL;
      endfor;
      if |Lk|  1 then
         k ← k + 1;
      for j = pass + 1 to k do begin
         Cj' ← apriori-gen(Lj-1, Lj-1')
         If Lj-1 <> Lj-1' then
            Cj' ← Cj' ∪ apriori-gen (Lj-1', Lj-1);
            Cj ← Cj ∪ Cj';
            TRAIN ← TRAIN ∪ Cj';
      endfor;
      foreach c ∈ TRAIN do          /* this step deletes from TRAIN every itemset */
         if c.complete = Yes then   /* that has completed a pass over d          */
            delete c from TRAIN;
     endwhile;
pass ← pass + 1;
prune-a-posteriori (Lpass);
endwhile;
for j = pass + 1 to k do
   a-posteriori(Lj);
endfor;
F = ∪i=1,...,k Li
```

Figura 10
Esquema del algoritmo T.D.I.C.

Generación de las Reglas

Para generar las reglas de asociación, es necesario encontrar todos los subconjuntos propios de cada itemset frecuente.
En otras palabras, dado un itemset frecuente Z, debemos encontrar, para cada subconjunto propio X de Z, las reglas

$$X \Rightarrow (Z\text{-}X) : fl_z \text{ tal que } s(Z, fl_z) / s(X, fl_z) \geq \theta.$$

Uno de los problemas que surgen para computar la confianza,

$$conf(\boldsymbol{X} \Rightarrow \boldsymbol{Y}: fl_{XUY}) = s(\boldsymbol{X} \cup \boldsymbol{Y}, fl_{XUY}) / s(\boldsymbol{X}, fl_{XUY})$$

es la determinación de $s(\boldsymbol{X}, fl_{XUY})$, es decir, el soporte de X en el intervalo fl_{XUY}.
Evidentemente, $s(\boldsymbol{X}, fl_{XUY})$ puede no ser igual a $s(\boldsymbol{X}, fl_X)$, dado que $fl_{XUY} \subseteq fl_X$.
Por otro lado, en la **Fase 1T** hemos calculado $s(\boldsymbol{X}, fl_X)$, pero no $s(\boldsymbol{X}, fl_{XUY})$.
Luego, si $\boldsymbol{XY}$ es un itemset frecuente de tamaño k, tenemos 2^k subconjuntos posibles, entonces deberíamos computar nuevamente la frecuencia para $2^k - 2$ itemsets en I_{XUY}.

Una forma de evitarlo, es usando una estimación. En el caso más simple, si consideramos que todos los itemsets X tienen una distribución temporal uniforme, entonces el cambio de apariencia en algún subconjunto de fl_X, en particular en fl_{XUY}, sería mínimo. Consecuentemente, seríamos capaces de estimar $s(\boldsymbol{X}, fl_{XUY})$ como $s(\boldsymbol{X}, fl_X)$.
La otra opción sería usar el histograma para calcular el soporte en el nuevo intervalo si necesitamos una estimación más exacta.
La elección de alguna de las dos opciones propuestas depende básicamente de la exactitud con la cual queremos obtener el soporte de la premisa, y posteriormente, la confianza de la regla.

Otro punto de decisión no tan fuerte, pero si relativamente importante es la estructura de la base de datos de transacciones. Puesto que si conocemos de antemano que la distribución de los items en las transacciones de la base de datos es relativamente uniforme, entonces no encontraremos demasiadas diferencias cuando computemos el soporte de la premisa de la regla en un intervalo u otro.
Si disponemos de dicha información, podríamos acelerar el proceso de generación de las reglas, utilizando el soporte para la premisa antes calculado, aunque el intervalo no sea el mismo.

CONTRIBUCION

La contribución de este trabajo consiste en la implementación de un modelo de reglas de asociación temporales que, mediante un desarrollo realizado en *Java* incorporado en el sistema *WEKA*, permite obtener un conjunto de reglas de asociación más precisas tomando el tiempo como factor para acotarlas en los intervalos dónde dichas reglas cobran mayor significado y validez de acuerdo a los criterios definidos por el usuario.

Además, de la experimentación llevada a cabo con la implementación del prototipo que se basa en el modelo de reglas antes descripto, tomando como entrada una base de datos sintética generada utilizando un generador de datos de dominio público, y una aplicación de unificación de los datos parciales generados a partir del generador de datos sintéticos, pudimos comprobar con los resultados obtenidos (tiempos de ejecución y valores del proceso) que estábamos en presencia de una herramienta capaz de dar una visión más general de los datos y las relaciones existentes entre los mismos a fin de identificar intervalos o zonas en las que es más conveniente focalizar el estudio y la aplicación de tecnologías para realizar un mejor aprovechamiento de la información y relaciones existentes.

Por último, el prototipo desarrollado, a medida que avanza la ejecución del proceso de búsqueda de los itemsets frecuentes y la generación de reglas de asociación, va generando archivos temporales de proceso en file-system. Estos archivos temporales de proceso pueden ser utilizados como entrada por otras aplicaciones, y ser tomados como referencia para continuar el proceso utilizando un conjunto de datos ampliado a fin de evitar el re-procesamiento de los mismos.

DESARROLLO

Como se mencionó anteriormente, el desarrollo propuesto tiene como objetivo analizar, desarrollar e implementar un prototipo que permita descubrir las reglas de asociación temporal, utilizando el tiempo como factor para incrementar el número de reglas encontradas.

Para ello, se ha tomado como base el algoritmo TDIC para el descubrimiento de las reglas de asociación temporal, propuesto en [2] [5], a partir de una base de datos de transacciones, en el problema conocido como análisis de la *canasta de mercado*, seleccionando los itemsets frecuentes y combinándolos para determinar las reglas de asociación temporal.

El enfoque está basado en identificar el período de vida de los items, es decir el primero y el último momento en el cual el item aparece en la base de datos de transacciones. A partir de ahí, computamos el soporte de un itemset en el intervalo definido por su período de vida, o subintervalos contenidos en él, y definimos el soporte temporal como la amplitud del subintervalo mínimo.

Como vimos, en ciertos casos un itemset puede no ser frecuente en el intervalo correspondiente a su período de vida completo, pero sí en uno o más subintervalos comprendidos en su período de vida. Por consiguiente, puede participar en las reglas de interés en estas porciones de su *lifespan*.
Estos subintervalos no dependen estrictamente de los datos, sino del umbral del soporte mínimo y el umbral del soporte temporal mínimo definido en el momento de realizar la búsqueda.

Como lo expresado en [2] [5], focalizamos nuestro interés en los intervalos que son *frecuentes en su lifespan y maximales con respecto al soporte temporal.*

La definición de intervalo maximal indica que no existe ningún otro subintervalo, contenido en el lifespan del itemset, en donde el itemset sea frecuente y cuya amplitud sea mayor que la amplitud del subintervalo que se está analizando.
Esta visión nos permite trabajar con intervalos de amplitud mayor, resignando quizás un valor de soporte más grande, pero manteniendo la restricción de que está por encima del umbral definido por el usuario.
De esta manera, una vez finalizado el proceso, obtendremos un conjunto de itemsets "menos frecuentes" pero con intervalos de frecuencia de mayor amplitud.

Luego, asociamos a cada itemset un *histograma*, el cuál se construirá durante el proceso de búsqueda de los itemsets frecuentes.
En el histograma se cuenta el número de transacciones que contienen el itemset en su período de vida.
El algoritmo desarrollado, mientras encuentra los itemsets frecuentes, genera el histograma.
El histograma puede ser visto como una colección de intervalos de tiempo $[ht_i, ht_i+\Delta_t]$, cuya amplitud Δ_t está definida por el usuario. En cada intervalo se especifica la cantidad de transacciones que contienen al itemset propietario del histograma, tal que el

identificador temporal de la transacción está comprendido en el lifespan del itemset, y en el período comprendido por el subintervalo $[ht_i, ht_i+\Delta_t]$.

Como vimos, el descubrimiento de las reglas de asociación temporal puede hacerse en dos fases. La primera fase consiste en encontrar todos los itemsets que son frecuentes en su lifespan, ó en los intervalos maximales comprendidos en su lifespan. La segunda fase, utilizando los itemsets frecuentes encontrados en la fase anterior, consiste en determinar las reglas de asociación.

De lo expresado en [2] [5], vemos que el tiempo juega un papel muy importante en el desarrollo del algoritmo, ya que su utilización nos permite determinar el intervalo en el cual el itemset realmente tiene importancia, y por consiguiente desechar todas las transacciones que tienen un identificador temporal (*timestamp*) que no está incluido en el lifespan del itemset, con los siguientes beneficios:
- reducir considerablemente la cantidad de transacciones a tener en cuenta por cada itemset,
- eliminar aquellas transacciones que están desactualizadas, de acuerdo al criterio del usuario, e
- incluir otras transacciones que, con el método de búsqueda tradicional, podrían no haberse tenido en cuenta por las limitaciones de soporte.

Estos beneficios nos permiten aumentar considerablemente la performance del algoritmo y, finalmente, obtener itemsets frecuentes y reglas de asociación de "mejor calidad".

El trabajo desarrollado ha consistido en la adaptación y/o modificación del sistema denominado *Weka* (ambiente de análisis de conocimiento).
Estas modificaciones están focalizadas principalmente en la adaptación de los procesos de selección e importación de las bases de datos a analizar, en la inclusión del algoritmo TDIC, desarrollado para el descubrimiento de los itemsets frecuentes y su posterior utilización en la construcción de las reglas de asociación temporal; y en la adaptación de la salida una vez finalizado el proceso de búsqueda.

Se ha desarrollado e implementado el algoritmo TDIC [2] [5], para el descubrimiento de las reglas de asociación temporal; junto con un conjunto de clases complementarias para mantener los estados intermedios de los objetos durante el proceso de búsqueda y generación de los itemsets frecuentes; y para ala construcción de las reglas de asociación temporal.
Además, han sido desarrolladas algunas herramientas y/o clases para el manejo y liberación de la memoria virtual durante la ejecución de dicho proceso.

Paralelamente, se desarrollaron otras aplicaciones que están relacionadas con la generación de las bases de datos sintéticos con características temporales, y procesos relacionados con el análisis de los resultados obtenidos, utilizados en el análisis de la performance del algoritmo desarrollado.

WEKA: Data Mining System

El sistema llamado *Weka* (Waikato Environment for Knowledge Analysis) fue desarrollado en la Universidad de Waikato, Nueva Zelanda, por Ian Witten y Eibe Frank.
La idea del desarrollo de este sistema fue proporcionar una herramienta para la exploración de los algoritmos de asociación; y la experimentación de los resultados obtenidos, utilizando los algoritmos de clasificación.

El sistema fue escrito en Java y fue testeado bajo los sistemas operativos Linux, Windows, y Macintosh.
Java permite proveer una interfase uniforme a muy diferentes algoritmos de aprendizaje, junto con métodos para pre y post procesamiento y evaluación de los resultados de los esquemas de aprendizaje sobre alguna base de datos dada.

Existen varios niveles diferentes en los cuales *Weka* puede ser usado. Como primer punto, provee implementaciones de los algoritmos de aprendizaje en estado de desarrollo que pueden ser aplicados a las bases de datos a través de la línea de comando. También incluye una variedad de herramientas para transformar las base de datos, como los algoritmos de discretización.
Además, es posible realizar el pre procesamiento de una base de datos, incluirla en un esquema de aprendizaje, y analizar el clasificador resultante y su performance.

El recurso más importante para navegar a través del software es la *documentación online*, la cual ha sido automáticamente generada desde el código fuente y reflejada en la estructura.
La documentación online es muy usada básicamente cuando se procesa la base de datos desde la línea de comando, ya que es la única lista completa de algoritmos disponibles. *Weka* está constantemente creciendo y la documentación online está siendo actualizada, debido a que es generada automáticamente desde el código fuente.

Una forma de usar *Weka* es aplicar los métodos de aprendizaje a una base de datos y analizar la salida para extraer información sobre los datos.
Otra forma es aplicar varios "learners" y comparar su performance para elegir uno para la predicción.

Los métodos de aprendizaje son llamados *classifiers*. Todos tienen la misma interfase en la línea de comando, y existe un conjunto de opciones comunes.
La performance de todos los *classifiers* es medida por un módulo de evaluación común.
La implementación de los esquemas de aprendizaje actual es el recurso más valioso que *Weka* provee, pero las herramientas de pre procesamiento de los datos, llamados *filters*, son también muy valiosas.
Como los *classifiers*, los *filters* tienen una interfase común en la línea de comando y hay un conjunto básico de opciones que los *filters* pueden ejecutar.

El enfoque principal de *Weka* está puesto en los algoritmos de clasificación y filtrado. Sin embargo, también incluye implementaciones de algoritmos para el aprendizaje de las reglas de asociación y para "clustering" de datos.
En la mayoría de las aplicaciones de data mining, el componente de aprendizaje de máquina es sólo una pequeña parte de un sistema de software grande.

Supongamos que tenemos algún dato y queremos construir un árbol de decisión a partir de él. Una situación común es que el dato esté almacenado en una hoja de cálculos o en una base de datos.
Sin embargo, *Weka* espera que el archivo a procesar esté en formato *ARFF*, porque es necesario tener información del tipo de cada atributo, la cual no puede ser automáticamente deducida de los valores del atributo.
Antes de poder aplicar algún algoritmo a los datos, el mismo debe ser convertido a formato *ARFF*.

Antes de meternos de lleno en los algoritmos de aprendizaje, veamos cómo es el contenido de un archivo *ARFF*.
La estructura de dichos archivos consiste de una lista de todas las instancias, con los valores de los atributos de cada instancia separados por comas.
Una vez que el archivo de datos fue convertido a formato *ARFF*, necesitamos agregarle el nombre de la base de datos con el tag *@relation*; el nombre, tipo y valores de cada uno de los atributos con el tag *@attribute*; y la sección de datos, es decir, las transacciones propiamente dichas, encabezadas con el tag *@data*.

Weka está organizado en paquetes que se corresponden con una jerarquía de directorios. Para entender esta estructura es necesario conocer como están organizados los programas Java. Cada programa Java es implementado como un conjunto de clases, relacionadas a través de sus métodos.
En la Programación Orientada a Objetos (POO), cada clase es una colección de variables junto con algunos *métodos* que operan sobre estas variables. Juntos, las *variables* y los *métodos*, definen el comportamiento de un objeto perteneciente a la clase.
Un *objeto* es simplemente una instanciación de la clase, que tiene valores particulares asignados a todas las variables de la clase.
Un *constructor* es un tipo especial de método que es llamado siempre que un objeto de una clase es creado, usualmente inicializando las variables que, colectivamente, definen el estado del objeto.
Otro tipo de método particular es *main*, el cual indica que la clase puede ser corrida desde la línea de comando.

En *Weka*, la implementación de un algoritmo de aprendizaje particular es representado por una clase.
Cada vez que la JVM (Java Virtual Machine) ejecuta una clase que representa un algoritmo de aprendizaje, crea una nueva instancia alocando memoria para construir y almacenar las variables de la clase.

Los programas grandes son usualmente particionados en más de una clase.
Java permite que las clases sean organizadas en *paquetes*. Un paquete es simplemente un directorio conteniendo una colección de clases relacionadas.
Como cada paquete corresponde a un directorio, los paquetes son organizados en una jerarquía en forma de árbol de directorios.

Weka está formado por un conjunto de paquetes. Algunos de los paquetes más importantes son:

- *weka.associations*,

- *weka.attributeSelection*,
- *weka.classifiers*,
- *weka.clusterers*,
- *weka.core*,
- *weka.estimators*,
- *weka.filters*

En la documentación de *Weka*, todos los paquetes están divididos en dos partes: el *índice de interfase*, y el *índice de clase*. El primero es la lista de todas las interfases que provee el paquete y el último es la lista de todas las clases contenidas dentro del paquete.

Una interfase es muy similar a una clase, la única diferencia es que la interfase no hace nada realmente por sí misma, sino que es meramente una lista de métodos sin implementaciones.
Otras clases pueden declarar que "*implementan*" una interfase particular, y proveer código para sus métodos. De esta forma, nos aseguramos que una clase que "*dice implementar*" una interfase, implementa los métodos que en ella se definen.

El paquete *core* es central en el sistema *Weka*, ya que contiene clases que son accedidas desde la mayoría de las otras clases.
Las clases más importantes del paquete *core* son Attribute, Instance, e Instances.
Un objeto de la clase Attribute representa un atributo, es decir, un *item* en particular, y contiene el nombre y el tipo del atributo (numérico, nominal o string); y en caso de ser un atributo nominal, la lista de valores posibles.
Los principales métodos de un objeto de la clase Attribute son los que permiten enumerar los valores del mismo, verificar la igualdad con otro atributo, retornar la cantidad de valores almacenados, y agregarle nuevos valores o eliminar algún valor existente.

Un objeto de la clase Instance contiene los valores de los atributos de una instancia particular, es decir una transacción de la base de datos.
Los métodos más importantes de esta clase son los que permiten agregar, modificar y/o eliminar los atributos o un atributo particular de la instancia, verificar si dos instancias tienen igual encabezado, y concatenar los items de dos transacciones.

Por otro lado, tenemos clase Instances. Un objeto de la clase Instances almacena un conjunto ordenado de instancias, en otras palabras, un *dataset* o base de datos de transacciones.
Los métodos más importantes de esta clase son los que posibilitan la importación de la instancia a través del archivo ARFF; agregar, eliminar y/o modificar una instancia o sus atributos; almacenar y enumerar los nombres de los atributos que la componen; almacenar y recuperar los *tags* del encabezado del archivo importado, entre otros.

El paquete *classifiers* contiene la implementación de la mayoría de los algoritmos de clasificación y predicción numérica. La clase más importante de este paquete es Classifier, la cual define la estructura general de cualquier esquema para clasificación o predicción numérica.
Esta clase contiene dos métodos, buildClassifier y classifyInstance, los cuales deben ser implementados por cada uno de los algoritmos de aprendizaje.

Por convención en *Weka*, los métodos buildClassifier se encargan de la inicialización interna completa del modelo antes de generar un nuevo clasificador.

En la jerga de la POO, los algoritmos de aprendizaje son representados por subclases de Classifier, y por consiguiente, automáticamente heredan estos dos métodos.
Cada esquema redefine e implementa estos métodos acorde a cómo ellos construyen un clasificador y como ellos clasifican las instancias.
Esta técnica, proporciona una interfase uniforme para construir y usar clasificadores desde cualquier código Java.

Otra clase importante es DistributionClassifier, la cual es subclase de Classifier y define el método distributionForInstance, el cual retorna una distribución de probabilidad para una instancia dada.
Cualquier clasificador que puede calcular probabilidades de clases es una subclase de DistributionClassifier e implementa este método.

El paquete *associations* contiene dos clases muy importantes: ItemSet y Apriori, las cuales en conjunto implementan el algoritmo Apriori [1].
Estas clases están alojadas en un paquete diferente porque las reglas de asociación son fundamentalmente diferentes de los clasificadores.
La clase Apriori implementa el *algoritmo Apriori* tal cual ha sido especificado en [1].
Esta clase extiende la clase Associator. Por lo tanto, implementa el método buildAssociations el cuál genera todos los itemsets frecuentes con mínimo soporte, y a partir de ellos, genera las reglas de asociación que tienen mínima confianza.
Inicialmente el *algoritmo Apriori* sólo puede trabajar con atributos nominales.

La clase ItemSet almacena un conjunto de items en orden lexicográfico, el cual está determinado por la información de la cabecera del conjunto de instancias utilizado para la generación del conjunto de items.
La definición inicial de esta clase asume que el conjunto de items es de tipo entero.
También contiene un contador para almacenar la cantidad de items del ItemSet y la cantidad total de transacciones procesadas.
Los métodos más importantes de esta clase son los relacionados con la creación del *itemset*, la manipulación de sus items (atributos), la comparación cono otro *itemset* y, finalmente, la generación y su posterior visualización de las reglas de asociación derivadas del *itemset*.
Debido a que esta clase está íntimamente relacionada con la ejecución del *algoritmo Apriori*, esta clase también contiene todos los métodos necesarios para la ejecución de dicho algoritmo.

El paquete *clusterers* contiene la implementación de dos métodos para el aprendizaje no supervisado: CODWEB y el algoritmo EM.

El paquete *estimators* contiene subclases de una clase genérica Estimator, la cual computa diferentes tipos de distribución de probabilidad. Estas subclases son usadas por el *algoritmo Naive Bayes*.

Junto con los esquemas de aprendizaje actuales, tenemos las herramientas para pre-procesar un dataset, llamadas *filters*, las cuales son un componente importante de *Weka*. En el paquete *filters*, la clase Filter es la análoga a la clase Classifier descripta anteriormente. Esta clase define la estructura general de todas las clases que contienen algoritmos de filtrado.
Como los clasificadores, los filtros pueden ser usados desde la línea de comando.

La selección de atributos es una técnica importante para reducir la dimensión de un dataset. El paquete *attributeSelection* contiene varias clases para hacerlo.
Estas clases son usadas por la clase AttributeSelectionFilter, para seleccionar un conjunto de atributos usando una clase de selección de atributos, desde *weka.filters*.

Existen algunas convenciones que hay que tener en cuenta a la hora de implementar clasificadores en *Weka*.

- La primera convención es que cuando un método buildClassifier es invocado, debe resetear el modelo.
  Cuando buildClassifier es llamado sobre un dataset, el mismo resultado debe ser siempre obtenido, sin tener en cuenta cuán a menudo el clasificador ha sido aplicado antes a otros datasets. Sin embargo, buildClassifier no debe resetear la variables de clase que corresponden a opciones específicas del esquema.
  También al llamar buildClassifier no se deben cambiar los datos de entrada.
- La segunda convención es que cuando el esquema de aprendizaje no pueda hacer una predicción, el método classifyInstance del clasificador debe retornar Instance.missingValue y el método distributionForInstance debe retornar cero probabilidades para todas las clases.
- La tercera convención concierne a los clasificadores para predicción numérica. Si un clasificador es usado para predicción numérica, classifyInstance sólo retorna el valor numérico que predice.
- La cuarta convención es que cada clasificador implementa el método toString() que retorna una descripción textual de sí mismo.

Hay dos tipos de algoritmos filtros en *Weka*: DiscretizeFilter, los cuales deben acumular estadísticas del dataset de entrada antes de poder procesar las instancias; y AttributeFilter, que pueden procesar cada instancia independientemente.
El primer paso para usar un filtro es indicarle el formato de entrada ejecutando el método inputFormat, el cual toma un objeto de la clase Instances y usa la información de sus atributos (cabecera) para interpretar futuras instancias de entrada.
El formato de los datos de salida del filtro puede ser determinado llamando al método outputFormat.

Al igual que con los clasificadores, existen convenciones para escribir filtros:

- La primera, un filtro nunca debe cambiar los datos de entrada o agregar instancias al dataset usado para proveer el formato de entrada.
- La segunda, cuando se invoca el método inputFormat debe inicializarse el estado interno del filtro, pero no alterar cualquier variable correspondiente a las opciones de la línea de comando provista por el usuario.
- Tercero, las instancias de entrada al filtro nunca deben ponerse directamente en la cola de salida del filtro. Por el contrario, deben ser reemplazadas por nuevos objetos de la clase Instance.

Arquitectura de la aplicación Weka – Explorer

El trabajo desarrollado se basó en modificar y adaptar la interfase *Explorer* del sistema *Weka*, permitiendo, entre otras cosas, la inclusión del algoritmo TDIC en el conjunto de *associators* que están definidos originalmente en el sistema.

Estas modificaciones fueron introducidas tratando de respetar el *look-and-feel* de la aplicación original, incorporando funcionalidades que están relacionadas con la manera en la cuál se importan las transacciones de la base de datos; la forma en la cual se listan los atributos que forman las transacciones; el tipo de valor de los atributos de las transacciones de la base de datos; la inclusión de nuevos algoritmos de asociación; y una vez ejecutado el algoritmo de asociación seleccionado, la forma en la cual se muestran y guardan los resultados obtenidos y los datos intermedios generados durante el proceso en el *filesystem*.

Para permitir centrar la atención exclusivamente en el objetivo del trabajo realizado, se "desactivaron" algunas *pestañas* ya que su funcionalidad no está relacionada con el trabajo de grado, y podía entorpecer el correcto desarrollo del mismo o la interpretación de las nuevas funcionalidades introducidas, dejando "activas" sólo las *pestañas* (plantillas) que permiten la especificación y carga de la base de datos a ser analizada, llamada Preprocess, y la que permite la selección y ejecución del algoritmo de asociación sobre el cual estamos realizando la exploración, llamada Associate.

En lo que respecta a la carga de la base de datos a analizar, y la posterior muestra de los atributos involucrados en la pantalla, se modificó la interfase original porque el sistema estaba preparado para importar archivos con extensión *ARFF*.
El problema radica en que, originalmente, el sistema asume que todas las transacciones de la base de datos tienen definido la misma cantidad de items. La única variación entre una transacción y otra está en el valor (cantidad de ocurrencias) de un item particular en dicha transacción.
También se asume que el valor de los items es de tipo nominal.

Como vimos, los archivos de base de datos *ARFF* están encabezados por un conjunto de *tags*.
El tag *@relation* define el nombre de la base de datos.
El tag *@attribute* define el nombre del atributo (item de la transacción) y, dependiendo del tipo asociado al atributo tenemos que el valor del mismo puede ser real, o nominal (una lista de valores encerrados entre llaves).
Y el tag *@data* que lista las transacciones de la base de datos.
Por simplicidad, se asume que los caracteres que definen la finalización de una transacción son RETORNO_DE_CARRO o FIN_DE_LINEA.
De esta manera, cada transacción de la base de datos está definida en una línea, comenzando desde el margen izquierdo, listando el valor de los atributos de la transacción, de izquierda a derecha, separados por comas.

Teniendo en cuenta que todas las transacciones de la base de datos tienen los mismos atributos (por ende, la misma cantidad de atributos) y sólo varía el valor de cada uno de ellos; tenemos que para especificar el valor de un atributo particular, no es necesario indicar el nombre del mismo ya que se asume que el valor i-ésimo de una transacción

(recordar que los valores están separados por comas) pertenece el atributo definido en el tag *@attribute* en la cabecera del archivo en el i-ésimo orden, de arriba hacia abajo.

Con el objetivo de analizar el problema conocido como *canasta de mercado*, vemos que esta situación es irreal ya que, si suponemos que la base de datos de transacciones pertenece a un momento particular de la facturación en caja de un supermercado, implicaría que todos los consumidores compraron exactamente los mismos productos, variando únicamente la cantidad de los mismos.

Para poder representar más fielmente este problema, se ha definido un nuevo tipo de archivo de base de datos, el cual ha sido llamado *TGF*, y al que se le han suprimido las líneas que hacían referencia al tag *@attribute*, entre otras cosas.

Con las modificaciones introducidas, un archivo de base de datos de transacciones de tipo *TGF*, tiene definida una cabecera en donde se indica, con el tag *@relation* el nombre de la base de datos; y posteriormente, con el tag *@data*, la lista de transacciones que constituyen la base de datos propiamente dicha.
Por lo tanto, se ha eliminando el tag *@attribute* y todas las líneas que hacían referencia al mismo ya que no conocemos de antemano que items están contenidos en todas las transacciones de la base de datos en el momento de generar la base de datos.

Además, en lo que respecta al contenido de las transacciones, se ha modificado la estructura de las mismas en varios aspectos:
- En primer lugar, posibilitando que las transacciones tengan un número diferente de atributos involucrados.
- En segundo lugar, agregando un *identificador unívoco* a cada transacción y;
- En tercero lugar, para representar el carácter temporal de las transacciones, agregando el *identificador temporal* que representa el instante de tiempo en el cual ocurrió dicha transacción.

De esta manera, al haber eliminado la cabecera del archivo que identificaba los items involucrados en todas las transacciones, y teniendo en cuenta que la cantidad de atributos de cada transacción es variable, se decidió cambiar el contenido de los atributos de las transacciones, de manera tal que en lugar de indicar la cantidad de ocurrencias (valor numérico) del atributo ubicado en la posición i-ésima de una transacción definido en la cabecera con el tag *@attribute*, ahora se indica el nombre del atributo involucrado, sin especificar la cantidad del mismo.
Por lo tanto, actualmente no se indica la cantidad de ocurrencias de un item particular en una transacción, sino la ocurrencia, en caso de estar presente, de dicho item en la transacción.

A modo de ejemplo, supongamos que tenemos el archivo *ARFF* que se muestra en la **Figura 11** (**A** y **B**).
En la **Figura 11-A** vemos los datos que forman la cabecera del archivo de base de datos. En la primera línea se indica el nombre lógico de la base de datos y en las líneas posteriores se detallan los atributos que forman las transacciones de la base de datos En Mientras que en la **Figura 11-B** vemos el detalle de las 7 transacciones que forman la base de datos. Cada transacción contiene los datos del tiempo, recogidos en un momento particular.

Definición de los atributos presentes en todas las transacciones

```
@relation weather

@attribute outlook {sunny, overcast, rainy}
@attribute temperature real
@attribute humidity real
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}
```

Figura 11-A
Ejemplo de base de datos ARFF

Especificación de las transacciones de la base de datos

```
@data
sunny,85,85,FALSE,no
sunny,80,90,TRUE,no
overcast,83,86,FALSE,yes
rainy,70,96,,FALSE,yes
rainy,68,80,FALSE,yes
rainy,65,70,TRUE,no
overcast,64,65,TRUE,yes
```

Figura 11-B
Ejemplo de base de datos ARFF

Luego, con las modificaciones introducidas, tenemos la estructura de un archivo de base de datos *TGF* que contiene las transacciones de un supermercado (con nombres de productos ficticios), ocurridas en el intervalo de tiempo que va desde 1 hasta 50, inclusive:

Nombre de los atributos (items) de la transacción. Los atributos de una transacción respetan el orden lexicográfico.

Transac. de la base de datos

```
@relation supermarket

@data
1, 1, PROD3, PROD20
2, 3, PROD8, PROD27, PROD29, PROD33
3, 3, PROD0, PROD6, PROD11, PROD16, PROD19, PROD21, PROD32, PROD45
4, 7, PROD4, PROD17, PROD20, PROD27, PROD39
5, 8, PROD28, PROD41, PROD44, PROD48
6, 20, PROD17, PROD25, PROD28, PROD44
7, 21, PROD8, PROD10, PROD15, PROD25, PROD36, PROD46, PROD47
8, 21, PROD22, PROD26, PROD28, PROD40
9, 21, PROD8, PROD16, PROD19, PROD24
10, 23, PROD3, PROD8, PROD29, PROD30, PROD36, PROD43, PROD46, PROD49
11, 32, PROD9, PROD20, PROD26, PROD29
12, 36, PROD4, PROD6, PROD19, PROD20, PROD47
13, 37, PROD9, PROD12, PROD33, PROD37, PROD49
14, 37, PROD0, PROD2, PROD27, PROD28
15, 42, PROD13, PROD26, PROD36, PROD37, PROD38, PROD49
16, 45, PROD13, PROD21, PROD41, PROD42
17, 46, PROD11, PROD14, PROD43, PROD46
18, 46, PROD40, PROD43
19, 50, PROD23, PROD30
```

Figura 12
Ejemplo de base de datos TGF

Identificador temporal (timestamp) de la transacción

Identificador unívoco de la transacción

Tal y como ocurre con los archivos *ARFF*, en los archivos *TGF* se asume que los caracteres que definen la finalización de una transacción son RETORNO_DE_CARRO y FIN_DE_LINEA.
Como vemos, cada una de las transacciones de los archivos *TGF* tiene definido, como primer valor, un identificador. Este identificador es unívoco en toda la base de datos y permite hacer referencia a la transacción en la base de datos con la seguridad que sólo vamos a obtener un elemento como resultado.

El valor numérico que aparece en la segunda posición de la transacción es el identificador temporal o *timestamp*. Este identificador es un valor que indica el momento en el cual ocurrió la transacción en la base de datos.

Volviendo al ejemplo de la *canasta de mercado*, supongamos que tenemos más de un lugar donde ocurren las transacciones, es decir, existe más de una caja registradora por donde los consumidores pueden pasar los productos a comprar y donde los mismos son registrados en la base de datos.
Con esta estructura, vemos que en el mismo instante de tiempo se pueden registrar varias transacciones, sin importar los productos (items o artículos) que cada uno de los consumidores está comprando.
Por lo tanto, si dos o más transacciones ocurren en el mismo instante de tiempo, tienen el mismo *timestamp* en la base de datos.

Como vemos en la **Figura 13**, las transacciones 7, 8 y 9 tienen el mismo *timestamp*, por lo que podemos deducir que ocurrieron en el mismo instante de tiempo.

```
7, 21, PROD8, PROD10, PROD15, PROD25, PROD36, PROD46, PROD47
8, 21, PROD22, PROD26, PROD28, PROD40
9, 21, PROD8, PROD16, PROD19, PROD24
```

Figura 13
Identificador temporal de la transacción

Como se dijo anteriormente, el prototipo desarrollado para el descubrimiento de las reglas de asociación temporal se incluyó en la aplicación *Weka* tratando de respetar el *look-and-feel* que originalmente tiene definido.

Con esta idea, se modificó la aplicación *Weka* original para que en el momento de su ejecución, se visualice el formulario que referencia el trabajo de grado en cuestión.

Figura 14
Formulario de inicio de la aplicación.

Inmediatamente después de visualizar el formulario inicial, se abre el formulario de selección del modo de análisis definido por *Weka* (Ver **Figura 15**).

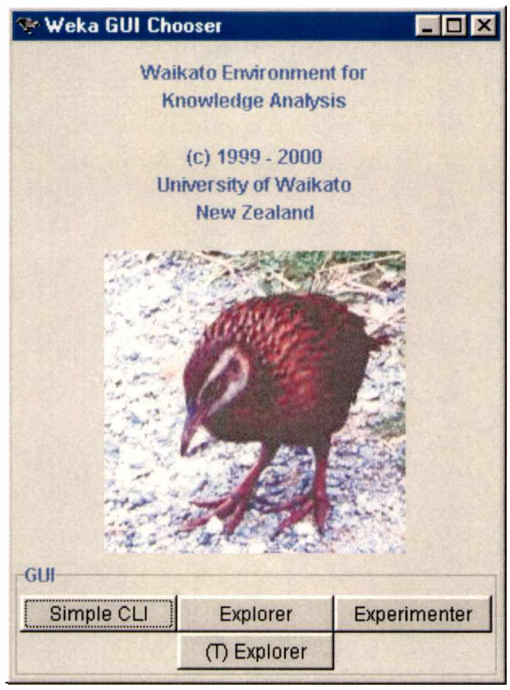

Figura 15
Formulario de selección del tipo de análisis.

Originalmente, este formulario permite abrir tres formularios diferentes.
El botón [Simple CLI] abre un formulario que emula la línea de comandos. Allí se pueden ejecutar diferentes comandos predefinidos (*java*, *break*, *kill*, *cls*, *exit*, y *help*). En particular, el comando *java* permite ejecutar las clases de la aplicación desde la línea de comandos.
El botón [Explorer] permite analizar una base de datos, ejecutar los algoritmos de clasificación y filtrado, y los algoritmos para el aprendizaje de las reglas de asociación y "clustering" de datos.

Por último, el botón [Experimenter] permite *experimentar* con los resultados obtenidos de las exploraciones previamente realizadas sobre las bases de datos seleccionadas.

Para incluir la nueva funcionalidad, con la idea que la funcionalidad original continuase estando vigente y accesible, se incluyó un botón [(T) Explorer] que permite llevar a cabo parte de la funcionalidad proporcionada por el botón [Explorer], pero con la posibilidad de realizar la exploración de los algoritmos de asociación con características temporales sobre los datos a analizar.

Haciendo click en el botón [(T) Explorer] vemos el formulario que se visualiza en la **Figura 16.**
Si tenemos en cuenta las características estéticas, vemos que ambos formularios son similares. La única modificación estética introducida en el formulario *Weka Knowledge Temporal Explorer*, con respecto al formulario *Weka Knowledge Explorer*, es que se han deshabilitado algunas "pestañas" que no estaban relacionadas directamente con el prototipo desarrollado para el trabajo de grado en cuestión.

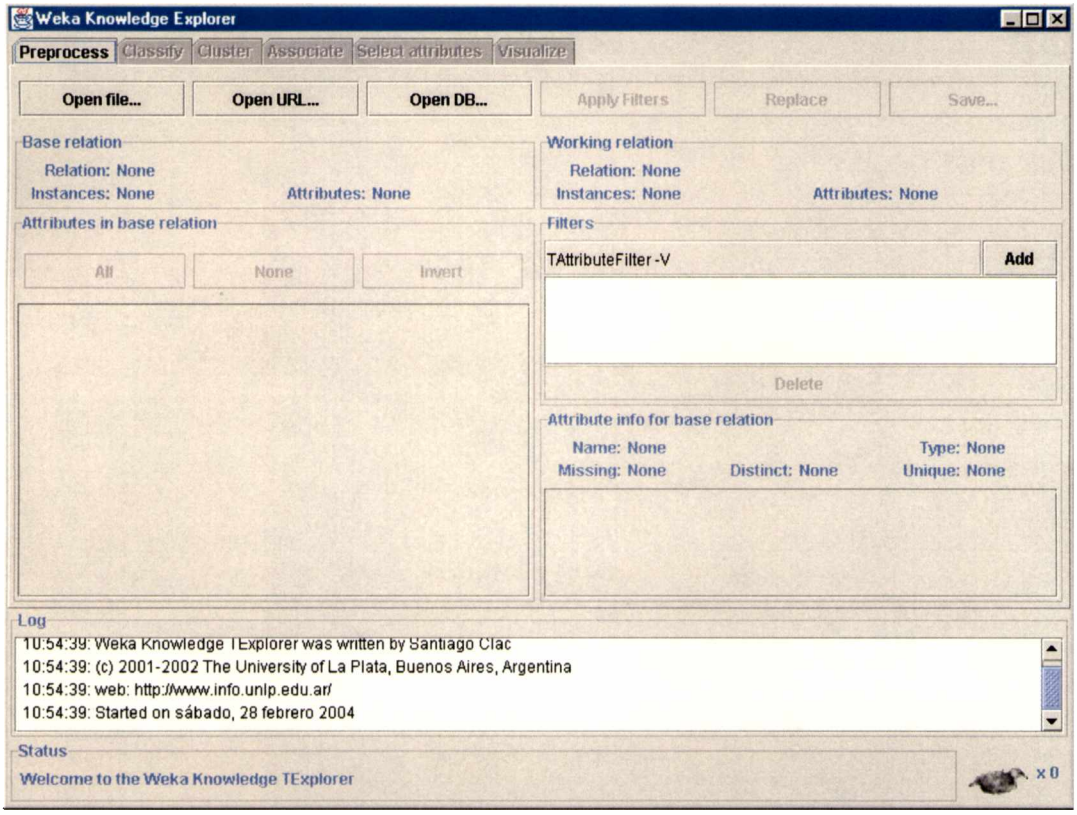

Figura 16
Vista inicial del formulario de selección e importación
de la base de atributos a analizar.

Importación de las bases de datos a analizar

Cuando en la pestaña Preprocess del formulario *Weka Knowledge Explorer* original se selecciona un archivo a procesar, a través de la poción [Open file...], se muestran los archivos con extensión *.arff* de las carpetas.
Una vez finalizada la importación del archivo seleccionado, la interfase muestra el nombre de los items (atributos) involucrados en las transacciones, los cuales son obtenidos de la cabecera del archivo ARFF, leyendo la información residente en el tag *@attribute*.

Con las modificaciones introducidas, en el formulario *Weka Knowledge Temporal Explorer* para la exploración de reglas de asociación temporal, cuando intentamos seleccionar un archivo de base de datos a procesar, vemos que se ha modificado el filtro de selección de archivos para que sólo se muestren los archivos con extensión *.tgf*.

Ahora, si tenemos en cuenta la nueva estructura definida para estos archivos, nos encontramos con el problema de cómo podemos identificar y visualizar los atributos que componen las transacciones de la base de datos importada ya que no contamos con la información cabecera (tag *@attribute*) que determine los atributos y el tipo de los items involucrados en las transacciones, y lo que es más problemático aún, la cantidad de items presentes en las transacciones nos es la misma.
De lo anterior tenemos que la única manera de conocer el nombre de todos los items que están presentes en la base de datos, es recorrer todas las transacciones que la componen, y construir una lista con los diferentes items a medida que son encontrados.
Como esta funcionalidad escapa a los objetivos del desarrollo propuesto, las modificaciones introducidas no permiten la visualización del nombre de los atributos que componen las transacciones de la base de datos.
En su lugar, se muestra una lista de los atributos, utilizando un nombre genérico para referirnos a ellos, encontrados en la transacción que tiene mayor cantidad de items.

Una vez finalizada la importación del archivo *.tgf* de base de datos; en el formulario se muestra una lista con los atributos encontrados en la base de datos similar a la que vemos a continuación:

$$\text{TG-ATTRIBUTE}_1,$$
$$\text{TG-ATTRIBUTE}_2,$$
$$\ldots,$$
$$\text{TG-ATTRIBUTE}_n$$

donde **n** es la cantidad máxima de atributos encontrados en una misma transacción dentro de la base de datos seleccionada.

Para ello, cada vez que se importa una base de datos a explorar, se guarda la cantidad máxima de items presentes en una transacción, y se arma la lista resultado con el nombre de items genérico TG-ATTRIBUTE_i.

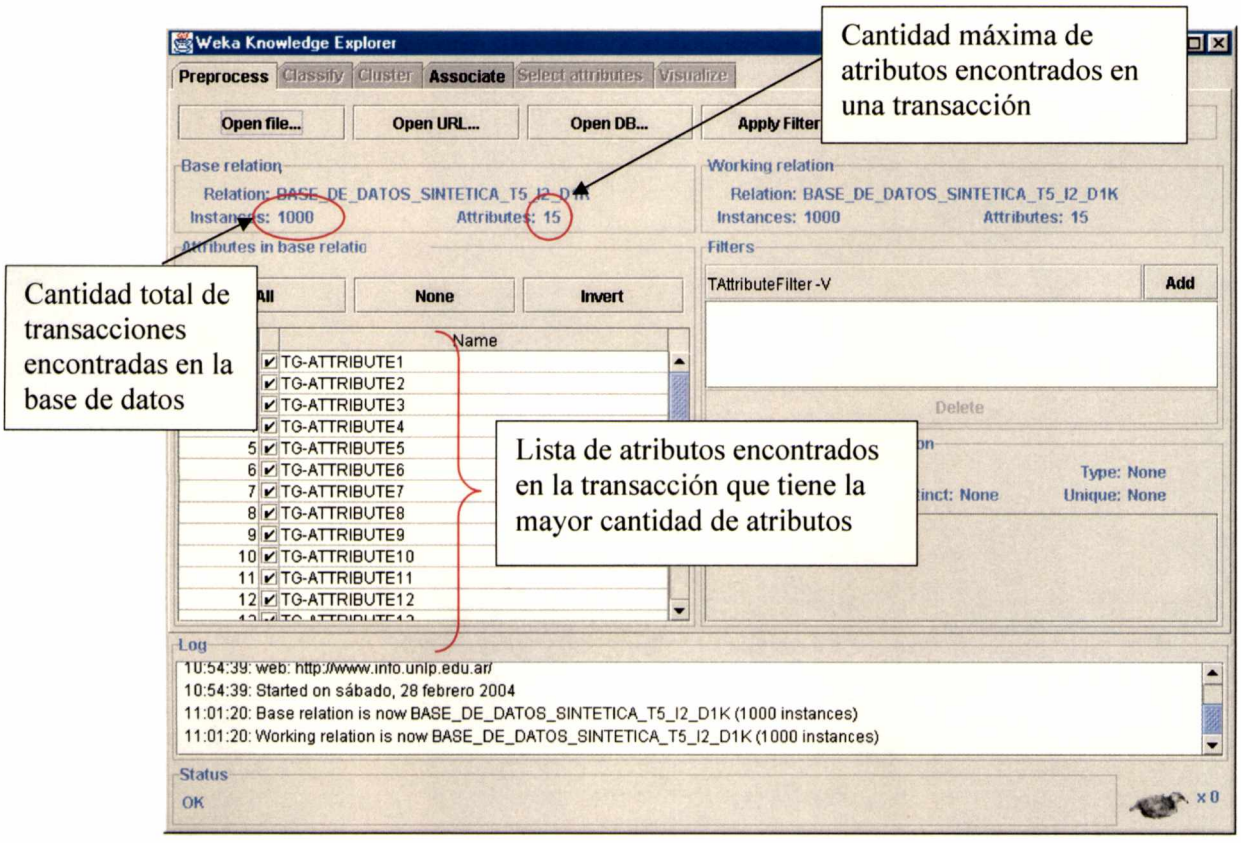

Figura 17
Formulario de selección e importación con los atributos de base de datos a explorar.

Selección y ejecución del algoritmo de asociación

Luego, para explorar el archivo de base de datos seleccionado e importado en busca de los itemsets frecuentes y las reglas de asociación temporal, vamos a la pestaña Associate (Ver **Figura 18**).

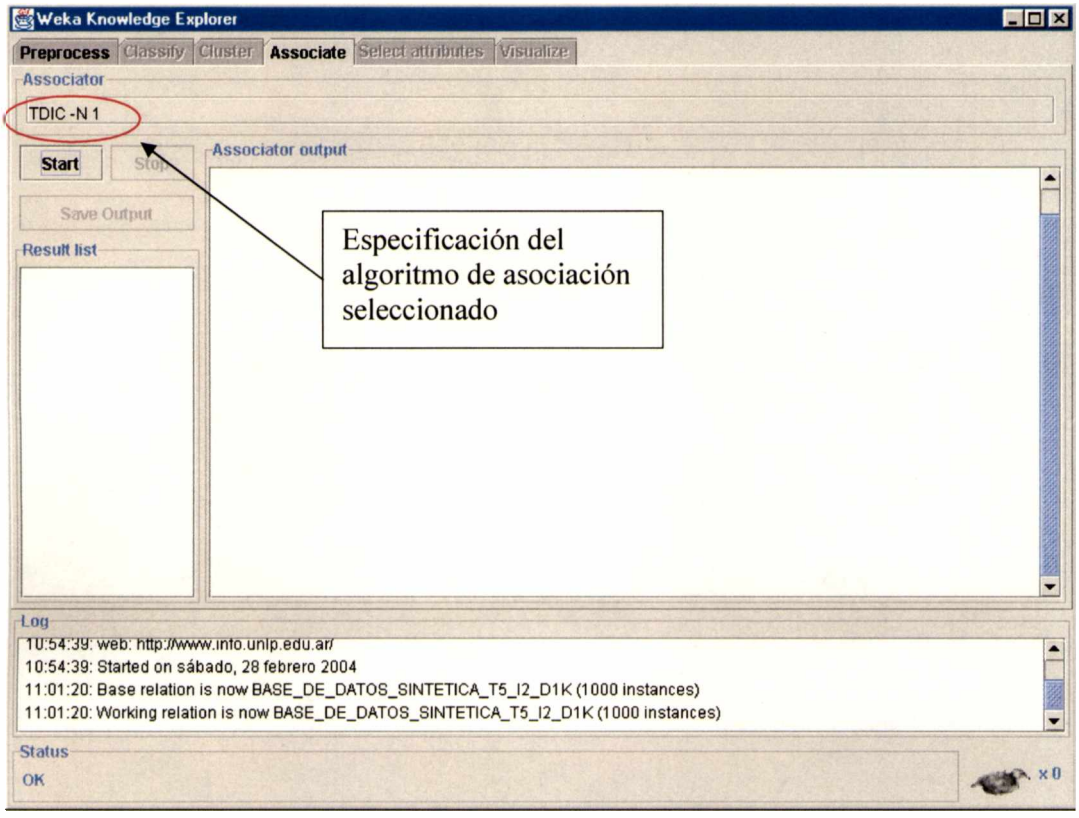

Figura 18
Vista del formulario, previa a la ejecución, del algoritmo de asociación seleccionado.

Esta ventana nos permite seleccionar el algoritmo de búsqueda de reglas de asociación temporal de interés, especificarle los parámetros para la corrida particular, ejecutar el algoritmo de búsqueda seleccionado, y guardar los resultados obtenidos una vez finalizada la exploración.

Una vez seleccionado el algoritmo de asociación, haciendo click sobre el botón [Start] comenzamos la exploración de la base de datos utilizando dicho algoritmo.
A partir de este momento, y hasta que finalice la ejecución del algoritmo en cuestión, la interfase nos va indicando el progreso de dicho proceso, adoptando una forma similar a la que se puede ver en la **Figura 19**.

Como podemos apreciar, en el formulario se indica el momento en el cuál comenzó la ejecución de algoritmo seleccionado, y se muestran los datos iniciales de la ejecución:

- esquema/algoritmo de asociación actualmente en ejecución,
- nombre de la base de datos sobre la cuál se está realzando la exploración,

- cantidad total de transacciones de la base de datos, y
- cantidad máxima de atributos (items) encontrados en una transacción de la base de datos.

En la parte inferior del formulario, vemos el *frame* "*Status*" donde se indica el estado de la exploración; y en el vértice inferior derecho, tenemos el símbolo de la aplicación *Weka*, que está en constante movimiento, de derecha a izquierda y viceversa, indicando que el proceso está activo. El contador que está asociado a la imagen indica la cantidad de procesos concurrentes (threads) que se están ejecutando.

Como vimos en la **Figura 18** y **Figura 19**, cuando comienza el proceso de exploración de la base de datos, utilizando el algoritmo de asociación, vemos que el botón [Start] es deshabilitado, al mismo tiempo que el botón [Stop] es habilitado.
De esta manera, en cualquier momento de la ejecución del algoritmo de asociación, es posible detener el proceso haciendo un click sobre el botón [Stop]. En este caso, en el *frame* "*Status*" se mostrará el mensaje "*See error log*", indicando que la ejecución del algoritmo de asociación ha sido detenida abruptamente por el usuario.
Simultáneamente, en el *frame* "*Log*", se indicará que la ejecución del algoritmo ha sido interrumpida de la misma manera. Este *frame* se utiliza para mostrarle al usuario todos los mensajes que se producen antes, durante y después de procesar la base de datos seleccionada.

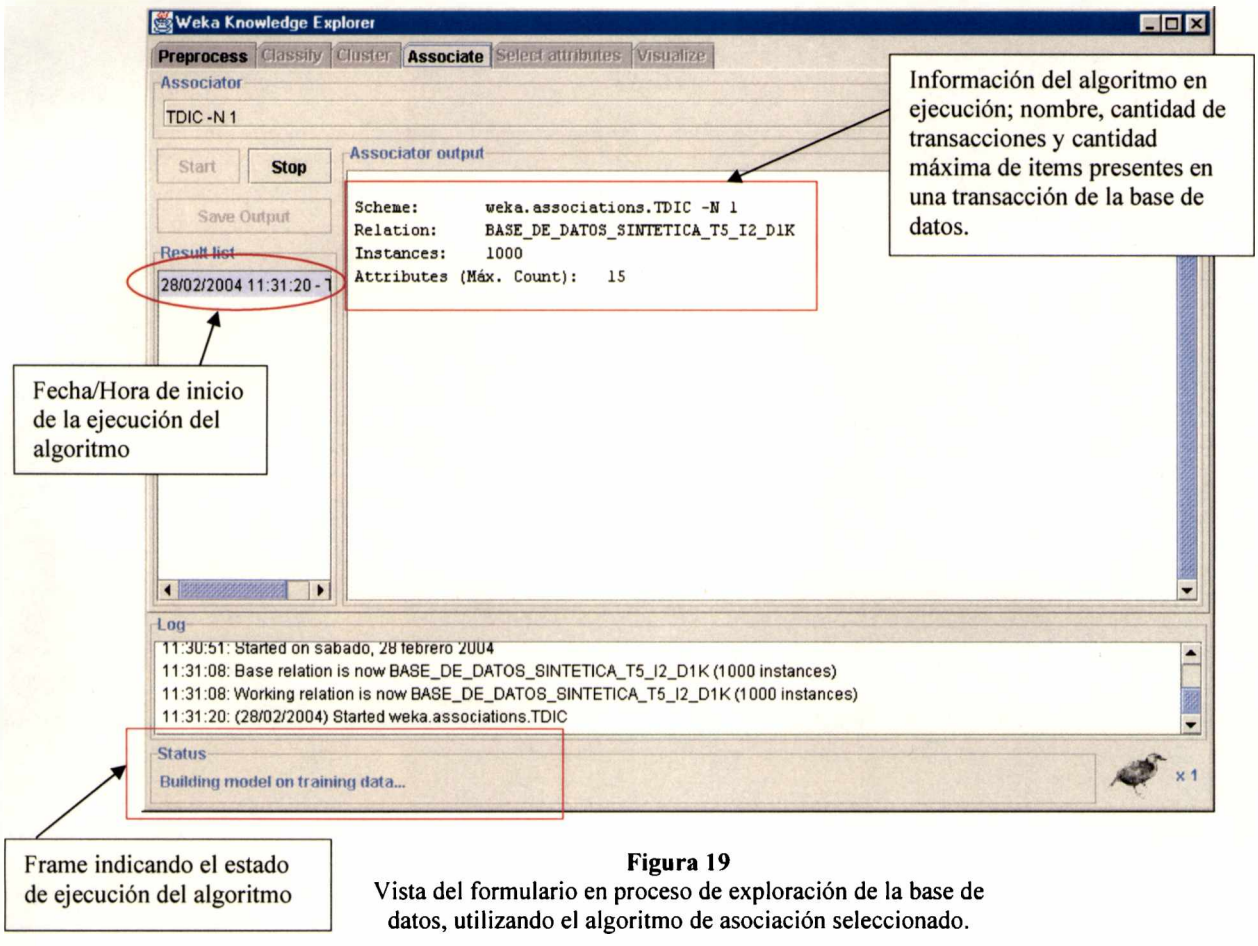

Figura 19
Vista del formulario en proceso de exploración de la base de datos, utilizando el algoritmo de asociación seleccionado.

Finalización de la ejecución del algoritmo de asociación

Una vez finaliza la ejecución de algoritmo de asociación, en el *frame* de salida "*Associator output*" se visualizan los resultados obtenidos en el proceso.

En este momento, en la versión original, los datos que se visualizan constan de la cantidad de itemsets frecuentes encontrados por cada uno de los tamaños, y el detalle de las reglas de asociación encontradas.
En este sentido, con las modificaciones introducidas, el prototipo desarrollado visualiza la cantidad de itemsets frecuentes encontrados, y el detalle (valor del itemset frecuente, intervalo de vida e intervalo de frecuencia, valor de soporte y valor de soporte temporal) de los mismos, por cada uno de los tamaños, junto con el detalle de las reglas de asociación encontradas.

La manera en la cual se visualizan las reglas de asociación ha sido modificada dándole una visión más relacionada con la estructura general de la regla de asociación temporal encontrada.
En la versión temporal, por cada una de las reglas de asociación encontradas se muestra la premisa y la consecuencia de la regla, con el símbolo "=>" como separador, junto con el intervalo en el cual la regla tiene validez. Además, se muestra el soporte y el soporte temporal de la premisa de la regla, y la colección de confianzas de la regla.

Por último, también se indica la fecha / hora de inicio y finalización del proceso realizado.
En la **Figura 20** podemos ver el estado del formulario una vez finalizado el proceso de generación de las reglas de asociación temporal.

Otra de las modificaciones introducidas en la aplicación *Weka* original, está relacionadas con la generación de una copia del archivo de salida en el directorio especificado en el archivo de configuración de la aplicación.

De esta manera, cuando finaliza el proceso de generación de las reglas de asociación temporal, automáticamente se genera un archivo de texto con el mismo conteniendo que el usuario está visualizando en el *frame* "*Associator output*".
El archivo de salida generado tiene por nombre la fecha y hora (con el formato *ddmmyyyyhhmiss*) en que finalizó el proceso y la extensión *.txt* y es copiado a la ruta especificada en el archivo de configuración.

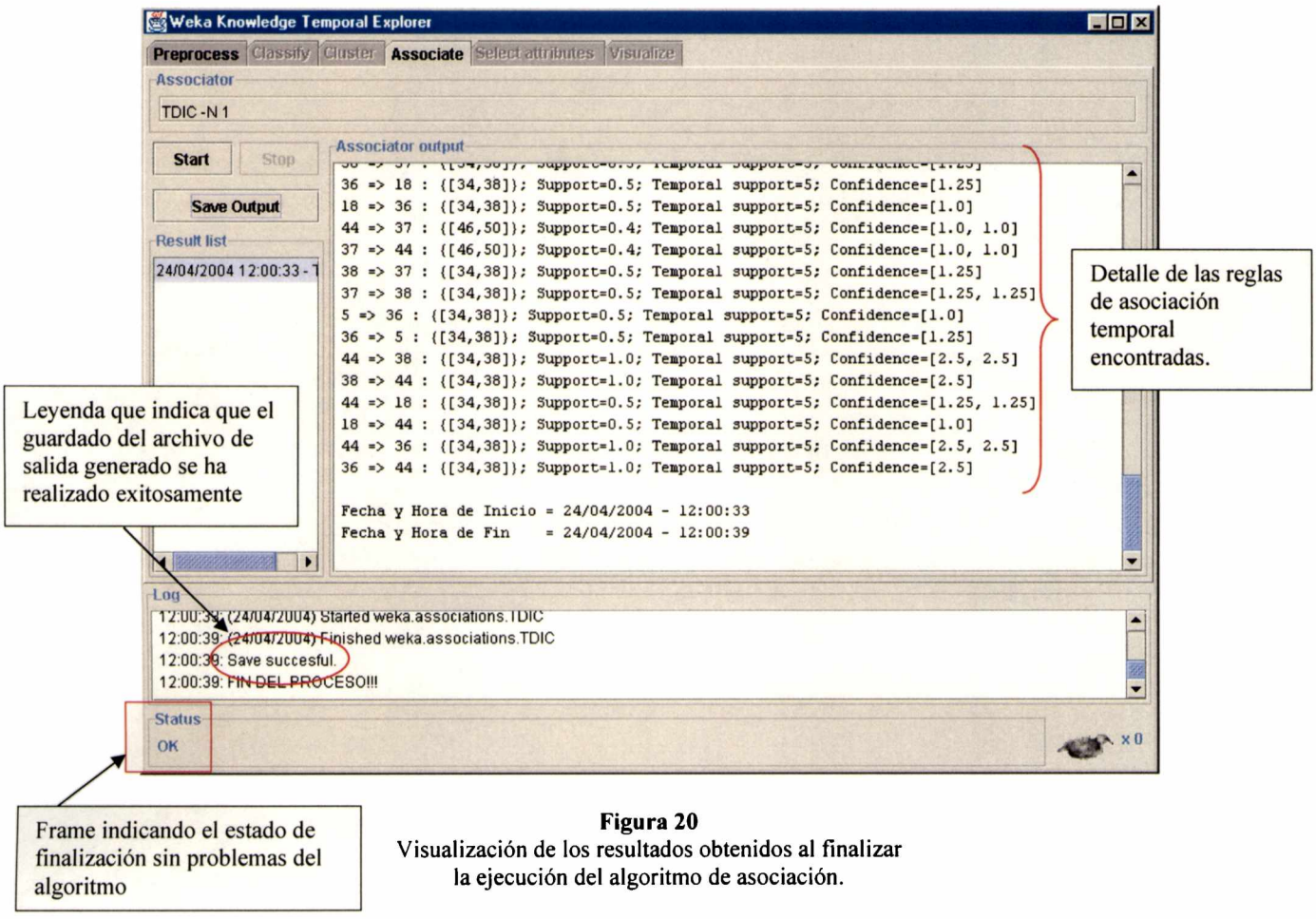

Figura 20
Visualización de los resultados obtenidos al finalizar la ejecución del algoritmo de asociación.

Especificación del prototipo desarrollado

Básicamente, el prototipo (algoritmo) desarrollado recibe la base de datos de transacciones, donde cada registro representa una transacción que está identificada en el tiempo, es decir que tiene un *timestamp* asociado; el umbral para el mínimo soporte y el mínimo soporte temporal; un valor numérico que indica la cantidad de transacciones que son leídas juntas cada vez que se lee de la base de datos, llamado *M*; un valor que determina la amplitud de los intervalos que forman el histograma, llamado Δ_t; el umbral para la mínima confianza; un valor booleano indicando si se debe recalcular el soporte de la premisa para el cálculo de la confianza de la regla; y el mínimo número de reglas a ser generadas.

También se define un histograma general, llamado *histograma R*, para contener todas las transacciones de la base de datos.

Respetando las convenciones de *Weka*, para implementar el algoritmo se definió una nueva clase llamada TDIC, subclase de Associator, que ejecuta el algoritmo cuando es invocado el método buildAssociations, el cual recibe como argumento la base de datos de transacciones.
Dicho método, construye las reglas de asociación temporal siguiendo estos pasos:
- inicializa el estado interno del objeto, setea la base de datos de transacciones recibida y guarda la fecha/hora de comienzo del proceso,
- encuentra todos los itemsets frecuentes, invocando al método findLargeItemSets (primera fase); y,
- utilizando los itemsets frecuentes encontrados en el paso anterior, genera las reglas de asociación temporal, invocando al método findRulesQuickly (segunda fase), y
- almacena la fecha/hora de finalización del proceso.

Para encontrar todos los itemsets frecuentes, el método findLargeItemSets mantiene un conjunto L_k que representa el conjunto de todos los k-itemsets frecuentes, es decir los itemsets frecuentes de tamaño k; un conjunto C_k con los k-itemsets candidatos, es decir los itemsets de tamaño k potencialmente frecuentes, y un conjunto TRAIN que contiene todos los itemsets candidatos que todavía estamos contando, o sea, los itemsets cuyo soporte se está contando.
Inicialmente, la variable *k* es seteada a 1, las colecciones C_k, L_k, y TRAIN, son inicializadas en vacío.

Por simplicidad, el algoritmo primero encuentra todos los itemsets frecuentes de tamaño 1 invocando el método singletons.
De esta manera tenemos en una sola pasada todos los 1-itemsets frecuentes lo cual nos da una proyección de los distintos k-itemsets posibles que vamos a encontrar en las pasadas subsiguientes.
Este método recorre toda la base de datos de transacciones identificando los items individuales, generando por cada uno de ellos un itemset con el valor del item, calculando su soporte y guardando el identificador temporal de las transacciones en las que el item apareció por primera y última vez para definir su lifespan.

Para llevar a cabo el proceso de identificación de los itemsets frecuentes de tamaño 1 se recorre toda la base de datos, y por cada transacción, obtenemos los items individuales que la forman y verificamos:

- ✓ Si es la primer ocurrencia del item en una transacción de la base de datos, entonces definimos un nuevo *1-itemset* con el valor del item. Además, le seteamos el contador de soporte en cero, los límites inferior y superior del lifespan y el histograma vacíos; y lo insertamos en la colección de salida respetando el orden lexicográfico.
- ✓ En caso contrario, buscamos en la colección de salida el itemset que referencia el item en cuestión.

En ambos casos, incrementamos el contador de soporte del *1-itemset* en 1, actualizamos el límite inferior y superior del lifespan, según corresponda, y el histograma de acuerdo al *timestamp* (identificador temporal) de la transacción que estamos analizando.
Cuando terminamos de procesar la transacción, actualizamos el *histograma R* con su identificador temporal.

De lo expuesto en [2] [5], el *histograma R* es de vital importancia a la hora de calcular el soporte de los itemsets en el intervalo, ya que contiene, por cada uno de sus subintervalos, la cantidad total de transacciones de la base de datos que están contenidas en dicho subintervalo.
Si no contásemos con la información almacenada en el *histograma R*, cada vez que necesitásemos calcular el soporte de un itemset, deberíamos recorrer toda la base de datos para calcular la cantidad de transacciones que están incluidas en el intervalo de vida del itemset, lo cual degradaría en gran medida la performance del algoritmo desarrollado.

Una vez que fueron analizadas y procesadas todas las transacciones, recorremos la colección de itemsets de tamaño 1 encontrados en la base de datos, y por cada uno de ellos, invocamos el método updateLargeIntervalsCounterFTr, pasándole como argumentos el *histograma R* y el número total de transacciones encontradas en la base de datos.
La funcionalidad del método consiste en dejar en estado consistente los 1-itemsets encontrados. Para lo cual:

- ✓ inicializa el intervalo de frecuencia con el valor del lifespan del itemset,
- ✓ actualiza el contador que contiene la cantidad total de transacciones de la base de datos, utilizando el *histograma R*, tal que su identificador temporal (*timestamp*) está incluido en el intervalo de frecuencia del itemset, y
- ✓ actualiza la variable que determina que el itemset ha recorrido todas las transacciones de la base de datos y ha completado el proceso de búsqueda.

Una vez finalizada la ejecución del método singletons, obtenemos una colección con todos los *1-itemsets* encontrados en la base de datos.
Cada uno de estos *1-itemsets* tiene definido el intervalo de vida (*lifespan*) y el intervalo de frecuencia, que va desde la primera transacción hasta la última en la cual el itemset ha sido encontrado; y su histograma conteniendo la colección de subintervalos de amplitud Δ_t, donde por cada uno de ellos se especifica la cantidad de transacciones, que están definidas en el lifespan del itemset, en las cuales el itemset está presente.

A los miembros de esta colección los llamamos *itemsets candidatos* porque todavía no podemos asegurar que sean frecuentes, pero son *candidatos a serlo.*

Luego, invocando el método deleteItemSets, analizamos cada itemset de la colección de *1-itemsets candidatos* retornada por el método singletons, para determinar:
- si el itemset candidato es frecuente en su período de vida, ó
- si el itemset candidato no es frecuente en su período de vida, pero si es frecuente en algún subintervalo maximal contenido en su período de vida.

Para ello, por cada itemset candidato obtenido, invocamos al método isFrecuent, pasándole como argumentos el umbral para el mínimo soporte y el umbral para el mínimo soporte temporal, definidos por el usuario.
El método, primero calcula el soporte y el soporte temporal del itemset y luego los compara con los valores recibidos como argumento.
Si los valores del soporte y soporte temporal calculados para el itemset son mayores o iguales a los valores mínimos definidos por el usuario, entonces retorna *verdadero* indicando que el itemset es frecuente en todo su período de vida.
Retorna *falso* en caso contrario.

Si el itemset candidato es frecuente en todo su período de vida, lo guardamos en la colección de salida; sino almacenamos el itemset candidato en la colección de itemsets a verificar si son frecuentes en algún subintervalo maximal contenido en su período de vida.

El cálculo de soporte del itemset se realiza de la siguiente manera: se divide el contador de soporte (indica la cantidad de transacciones que están incluidas en el lifespan del itemset y lo contienen) por la cantidad total de transacciones que están definidas en el lifespan del itemset (este valor es calculado utilizando el *histograma R*).
El valor obtenido del cálculo anterior es el que determina cuán frecuente es el itemset en la base de datos de transacciones, puesto que indica la cantidad de transacciones que contienen al itemset, sobre el total de transacciones que están definidas en su período de vida.

De manera similar, el valor del soporte temporal del itemset se computa como el valor del módulo del intervalo de frecuencia. Para ello, debemos tener en cuenta la *diferencia temporal* definida por el usuario en el archivo de configuración, que determina la diferencia real que existe entre dos transacciones consecutivas de la base de datos.
Para clarificar aún más el concepto del cálculo del soporte temporal utilizando la *diferencia temporal*, veamos un ejemplo.
Supongamos que tenemos una base de datos como la definida en la **Figura 9**. Aquí, los identificadores de las transacciones están representados como múltiplos de 100.
Como vimos anteriormente, aplicando la función singletons obtenemos el 1-itemset candidato {A}, entre otros, el cuál está definido en el intervalo [100,1200].
Luego, si calculamos el módulo del intervalo [100,1200], nos da como resultado 1100. Pero esto no es verdad ya que nos existen 1100 instantes de tiempo entre la transacción 1 y la transacción 12 de la base de datos, puesto que los identificadores temporales de las transacciones están representados como múltiplos de 100.
Por lo tanto, si asumimos que la diferencia temporal entre dos transacciones consecutivas de la base de datos es 100, el módulo nos da el valor 11.

Luego, le sumamos el valor 1 para considerar ambos límites del intervalo, y obtenemos que el valor del módulo del intervalo de frecuencia del itemset {A} es igual a 12.

En este punto es donde encontramos las mayores diferencias entre el algoritmo TDIC y los demás algoritmos.
Esto se debe principalmente a que en el algoritmo TDIC para verificar la frecuencia de un itemset se computa su soporte (de la misma manera en todos los algoritmos) y se lo divide por la cantidad total de transacciones que están definidas en su lifespan, teniendo en cuenta sólo el conjunto de transacciones comprendido entre la primera y última transacción en que el itemset está presente en la base de datos.
El resto de las transacciones no son tenidas en cuenta porque su identificador temporal no está incluido en el lifespan del itemset, y por consiguiente, no son de interés para el cálculo del soporte.
Por otro lado, el soporte temporal es un concepto que ha sido incorporado en [2] [5], y por consiguiente, no está presente en los algoritmos precedentes.

Una vez procesados todos los itemsets candidatos, tenemos la colección de *1-itemsets* que son *frecuentes* en todo su período de vida.
Luego, invocando el método a_posteriori, con la colección de *1-itemsets candidatos* que no son frecuentes en todo su período de vida como argumento, verificamos si existe algún intervalo maximal, contenido en el período de vida del itemset, en el cual el *itemset candidato* es frecuente.

No son de interés todos los posibles subintervalos derivados del *lifespan* del itemset, sino que nuestro interés está focalizado en aquellos subintervalos contenidos en el *lifespan* del itemset que son maximales (de amplitud máxima) y que cumplen con el mínimo soporte y el mínimo soporte temporal especificado por el usuario.

Para ello, el método a_posteriori verifica si cada itemset de la colección recibida como argumento contiene algún intervalo maximal en donde el itemset en cuestión es frecuente, utilizando el umbral para el soporte, el umbral para el soporte temporal y el *histograma R* para realizar las comparaciones.
Por cada itemset de la colección, recorremos su histograma de atrás hacia delante y en cada pasada verificamos si es frecuente tomando dicho subintervalo para realizar los cálculos.
Inicialmente, comenzamos definiendo un subintervalo con los mismos valores que el *lifespan*, pero en cada pasada decrementamos el límite superior del subintervalo en Δ_t unidades, verificando si el itemset es frecuente en cada pasada tomando ese subintervalo como *lifespan*.
Si el itemset no es frecuente en dicho subintervalo, se decrementa el límite superior del subintervalo, y se repite el proceso.

Por el contrario, si el itemset es frecuente en dicho subintervalo, entonces se genera una copia del itemset frecuente con los valores del intervalo maximal como intervalo de frecuencia y se agrega en la colección de salida.
En este caso se repite el proceso tomando como nuevo límite inferior del subintervalo el límite superior del intervalo maximal, recientemente encontrado, incrementado en Δ_t

unidades, y como nuevo límite superior el límite superior del lifespan (original) del itemset.

Este proceso se repite tantas veces como posibles intervalos maximales podamos calcular para el itemset en cuestión, teniendo en cuenta que, para todos los posibles subintervalos maximales, el límite inferior nunca puede ser mayor que el límite superior; y que la amplitud (módulo) de los intervalos maximales generados debe ser mayor o igual que el umbral para el soporte temporal recibido como argumento.

En la **Figura 21** tenemos un esquema del algoritmo A_posteriori.

```
int lowTop = lifespan().lowerLimit();
int uppTop = lifespan().upperLimit();
int pos  = uppTop;
int modulo = ClsGeneral.modulo(pos, lowTop);
int lowBottom, uppBottom;

if ((this.m_counterFr > 0) && (modulo > minTempSup)) {

  while ((lowTop < pos) && (modulo >= minTempSup)) {
    lowBottom = lowTop;
    uppBottom = uppTop;
    pos  = uppBottom;

    modulo = ClsGeneral.modulo(pos, lowBottom);

    while ((lowBottom < pos) && (modulo >= minTempSup)) {
      tInt = new TInterval(lowBottom, pos);

      if (this.isFrecuent(minSup, minTempSup, tInt, histR)){
        aux = this.copy();
        aux.largeInterval(new TInterval(lowBottom, pos));
        result.put(aux);

        lowBottom = pos + TDIC.deltaT();
        pos = uppBottom;
      } else
        pos = pos - TDIC.deltaT();

      modulo = ClsGeneral.modulo(pos, lowBottom);
    }

    lowTop = lowBottom + TDIC.deltaT();
    pos = uppTop;
    modulo = ClsGeneral.modulo(pos, lowTop);
  }
}

return result;
```

Figura 21
Esquema del algoritmo A_posteriori

Una vez calculados los itemsets que son frecuentes en todo su lifespan, ó en uno o más subintervalos maximales contenidos en su lifespan, tenemos L_1, la colección de *1-itemsets frecuentes*.

Si el tamaño de la colección L_1 es mayor a cero, es decir que ha sido encontrado algún *1-itemset frecuente*, entonces:
- se incrementa el valor de *k* (indicador del tamaño máximo de los itemsets frecuentes que se están calculando),
- se incrementa el contador de pasadas, previamente inicializado en 1, y
- se combinan todos los itemsets de longitud 1, para formar los itemsets candidatos de longitud 2 (*2-itemsets candidatos*).

Para realizar esta tarea, se utiliza la función apriori_gen [5], la cual toma como argumento la variable L_{k-1}' que contiene el conjunto de todos los *(k-1)-itemsets frecuentes* encontrados en las pasadas anteriores con sus lifespan asociados, y la variable L_{k-1} que contiene el conjunto de todos los *(k-1)-itemsets frecuentes* encontrados en la última pasada.
Una vez finalizada su ejecución, la función retorna el super conjunto con los *k-itemsets candidatos*, C_k, cada uno con su lifespan asociado.

Para llevar a cabo la generación de los itemsets candidatos se aplica el criterio definido en [5], a saber:

sean **v** y **w**, dos *(k-1)-itemsets* con lifespan $[\mathbf{v}.t_1, \mathbf{v}.t_2]$ y $[\mathbf{w}.t_1, \mathbf{w}.t_2]$, respectivamente, y

sea **u** el *k-itemset* generado a partir de **v** y **w** con la función apriori_gen, entonces el lifespan de **u** es $[\mathbf{u}.t_1, \mathbf{u}.t_2]$, donde

$$\mathbf{u}.t_1 = \max \{ \mathbf{v}.t_1, \mathbf{w}.t_1 \} \text{ y}$$
$$\mathbf{u}.t_2 = \min \{ \mathbf{v}.t_2, \mathbf{w}.t_2 \}$$

La función apriori_gen está organizada en dos pasos: el paso de *Join*, y el paso de *Prune*.
El paso de *Join*, se ejecuta a través del método joinItemSets, el cual recibe los dos conjuntos L_{k-1}' y L_{k-1}, y genera el super conjunto C_k.
Para ello, toma cada uno de los *(k-1)-itemsets frecuentes* del conjunto L_{k-1}, y por cada de los *(k-1)-itemsets frecuentes* restantes verifica que:
- ✓ los items de ambos *(k-1)-itemsets* que se encuentran desde la posición 1 hasta la posición (k-2), inclusive, sean iguales; y
- ✓ el item de la posición (k-1) del primer itemset sea menor (de acuerdo al orden lexicográfico) que el item de la posición (k-1) del segundo itemset que estamos analizando.

Si lo anterior se cumple, formamos un nuevo *k-itemset candidato* a partir de dos *itemsets frecuentes* de tamaño (k-1), con la siguiente estructura:
- desde la posición 1 hasta la posición (k-2) contiene los mismos items que los *(k-1)- itemsets originarios*; y
- el item de la posición (k-1) es igual al item de la posición (k-1) del primer *itemset frecuente* de tamaño (k-1); y
- el item de la posición (k) es igual al item de la posición (k-1) del segundo *itemset frecuente* de tamaño (k-1).

Teniendo en cuenta la forma en la cual se construye el nuevo *itemset candidato* de tamaño k, podemos asegurar que los items del nuevo *k-itemset* están ordenados según el orden lexicográfico.

Una vez terminado el proceso de *join* con los *(k-1)-itemsets frecuentes* de L_{k-1}, se repite el proceso con los *itemsets frecuentes* del conjunto $L_{k-1}' \cup L_{k-1}$.
Ahora, para calcular los *k-itemsets candidatos* se toma un *(k-1)-itemset frecuente* del conjunto L_{k-1}' y un *(k-1)-itemset frecuente* del conjunto L_{k-1} y se construye un nuevo *k-itemset candidato* a partir de ellos.

El paso de *Prune (poda)* se lleva a cabo invocando al método pruneItemSets.
Cuando se ejecuta este método, todos los *k-itemsets* candidatos generados en el paso anterior, que tienen algún subconjunto propio que no está incluido en L_{k-1}' ni en L_{k-1}, son eliminados.
Asimismo, los *k-itemsets* candidatos generados, tal que la amplitud de su lifespan es menor que el umbral para el mínimo soporte temporal definido por el usuario, son eliminados.

En resumen, una vez encontrados y generados los itemsets frecuentes de tamaño 1, utilizando el método singletons, tenemos el conjunto L_1.
Luego, invocando la función apriori_gen con L_1, y el conjunto vacío (no existen itemsets frecuentes de tamaño 1 encontrados en las pasadas anteriores ya que ésta es la primer pasada que realizamos) tenemos el conjunto C_2 conteniendo los itemsets candidatos de tamaño 2.
Los *itemsets candidatos* de C_2 son agregados en TRAIN para calcular su soporte, y determinar su frecuencia.
A partir de este momento, se leen *M* transacciones de la base de datos.

El prototipo recorre la colección de itemsets candidatos de TRAIN para computar su soporte utilizando el método upDateCounters. Este método es similar al procedimiento subset definido en [1].
Dicho método recibe como argumentos al conjunto de itemsets candidatos TRAIN, y las M transacciones recientemente leídas de la base de datos; y actualiza el soporte de los itemsets candidatos de TRAIN que están incluidos en dichas transacciones.
Para ello, toma cada una de las transacciones recibidas y recorre el conjunto de itemsets candidatos invocando al método upDateCounter, pasándole la transacción como argumento.

Cuando un itemset (candidato) ejecuta dicho método, intenta actualizar su soporte y soporte temporal, verificando:

1) Si el identificador temporal de la transacción está contenido en el lifespan del itemset, entonces incrementa el contador que almacena el *total de transacciones procesadas* por el itemset, el cual será utilizado para determinar si el itemset ha procesado todas las transacciones de la base de datos (ha completado una pasada en la base de datos).
2) Si el paso anterior se cumple, verifica si el itemset está contenido en la transacción, en cuyo caso:
   a) incrementa el contador de soporte del itemset,

b) actualiza los límites del intervalo de frecuencia del itemset con el identificador temporal de la transacción que se está procesando, e
c) incrementa en 1 (uno) el contador de transacciones del subintervalo del histograma del itemset que incluye el identificador temporal de la transacción que se está procesando.

Como se dijo anteriormente, el histograma está formado por una colección de subintervalos.
Veamos ahora un poco más en detalle cómo es la estructura del histograma ya que es sumamente importante en el proceso de generación de los itemsets frecuentes.
En lo que respecta a la implementación del prototipo desarrollado, tenemos que cada uno de los histogramas es un objeto de la clase Histogram.
Además, cada histograma está formado por una colección de objetos de la clase HistogramElement.

Supongamos que tenemos un itemset X el cuál está definido en el intervalo $[t_1, t_2]$.
Luego, podemos imaginarnos al histograma del itemset X como el conjunto de tuplas:

$(v_1, cont_1)$,
$(v_2, cont_2)$,
...,
$(v_i, cont_i)$,
...,
$(v_n, cont_n)$,

donde $(v_i, cont_i)$ es un objeto de la clase HistogramElement.
Con esta estructura, tenemos que cada una de los elementos del histograma $(v_i, cont_i)$ de la colección debe responder a las siguientes características:

- ✓ $v_i = [t_i, t_i+\Delta_t]$ representa el subintervalo contenido en $[t_1, t_2]$,
- ✓ $cont_i$ indica el número de transacciones en las que el itemset está presente, y cuyo *timestamp* está contenido en v_i.

De lo anterior, tenemos que el intervalo del histograma está comprendido entre el mínimo y el máximo valor de v_i, es decir $[v_1, v_n]$.
Por otro lado, si tenemos en cuenta que, por definición, el itemset X está definido en el intervalo $[t_1, t_2]$, entonces llegamos a la conclusión que $v_1 = t_1$ y $v_n = t_2$.

Inicialmente, el conjunto de tuplas del histograma está vacío. Cuando se agrega una transacción en el histograma, tomamos su identificador temporal y verificamos en que subintervalo $[t_i, t_i+\Delta_t]$ está incluido.
Debido a que la colección de elementos del histograma se crea vacía, entonces puede darse el caso de que no exista algún subintervalo (HistogramElement) del histograma en el cuál el identificador temporal de la transacción pueda estar contenido.
En este caso, debemos crear una nueva instancia de la clase HistogramElement, con el contador de transacciones seteado a cero, y definirle como intervalo de interés del itemset a $[v_{n+1}, v_{n+1}+\Delta_t]$, donde v_n es el máximo valor de índice de los subintervalos definidos hasta el momento en el histograma del itemset.

Una vez que tenemos identificado el subintervalo que contiene unívocamente a la transacción procesada, incrementamos en 1 el contador de transacciones ($cont_i = cont_i + 1$).
Por último, actualizamos el intervalo del histograma para indicar que va desde v_1 (el menor de los límites inferiores) hasta v_{n+1} (el mayor de los límites superiores).

Cabe recordar que, de acuerdo a la forma en la cual el prototipo desarrollado va construyendo el histograma, siempre tomamos transacciones que contienen al itemset, y cuyos identificadores temporales están contenidos en el período de vida (*lifespan*).
Por esta razón, bajo ninguna circunstancia, el límite inferior del histograma podrá ser menor que el límite inferior del lifespan del itemset, ó el límite superior del histograma podrá ser mayor que el límite superior del lifespan del itemset.

Dado que TRAIN contiene todos los itemsets candidatos que se están procesando actualmente, es posible encontrar en TRAIN itemsets candidatos de diferente tamaño (2-itemsets candidatos, 3-itemsets candidatos, ..., k-itemsets candidatos, siendo *k* el tamaño máximo de los itemsets frecuentes que estamos procesando) que no han recorrido todas las transacciones de la base de datos [3].

Una vez finalizada la ejecución del método upDateCounters, tenemos actualizados los contadores de soporte y de soporte temporal de los itemsets candidatos de TRAIN con las *M* transacciones leídas.
Luego, por cada valor *j*, desde el *contador de pasada* hasta *k*, inclusive; recuperamos de TRAIN todos los *j-itemsets candidatos* que cumplen con el mínimo soporte y el mínimo soporte temporal, es decir que son frecuentes en su lifespan o en algún intervalo maximal contenido en su lifespan.

Para ello, utilizamos el método deleteItemSets, el cuál toma que argumento, además del conjunto TRAIN, el tamaño de los itemsets candidatos que debe analizar (valor *j*).
El método recupera todos los *j-itemset candidato c* de TRAIN, y por cada uno de ellos:
- actualiza el contador de soporte temporal, invocando el método calculateCounterFTr() con el *histograma R* como argumento; y
- verifica si es frecuente, pasándole el umbral para el mínimo soporte y el umbral para el mínimo soporte temporal.

Si lo anterior es correcto, agrega el itemset candidato *c* (que a partir de este momento llamaremos *j-itemset frecuente c*) en la colección de itemsets de salida.

Una vez finalizada la ejecución del método, agrego en L_j los *j-itemsets frecuentes* encontrados.
Luego verifico si el valor *j* es igual a *k*, en cuyo caso guardo en L_k' los *j-itemsets frecuentes* encontrados en la última pasada.

Cuando termino de procesar todos los itemsets candidatos de TRAIN, verifico si en la última pasada encontré algún itemset frecuente de tamaño *k*, para ello, verifico si el conjunto L_k' no está vacío.
Si lo anterior es correcto, incremento en 1 el valor de *k*, para incluir en el siguiente proceso el cálculo de los itemsets candidatos de siguiente tamaño.

Luego, por cada valor *w*, desde el *contador de pasada*+1 hasta *k*, inclusive, tal que encontré hasta el momento algún itemset frecuente, recupero:
- el conjunto con los *w-itemsets frecuentes* encontrados en la última pasada, Q_w, y
- el conjunto con los *w-itemsets frecuentes* encontrados en las pasadas anteriores, Q_w';

y genero los nuevos (*w+1*)*-itemset candidatos*, utilizando nuevamente el método apriori_gen, con los conjuntos Q_w y Q_w' y el valor actual de *w* como argumento.

Una vez finalizada la ejecución del método apriori_gen, agrego en TRAIN el super conjunto generado con los nuevos (*w+1*)*-itemsets candidatos*, para que sean procesados en la próxima pasada.
Paralelamente, se agregan los *j-itemsets frecuentes* encontrados en la última pasada en la colección de *j-itemsets frecuentes* encontrados en las pasadas anteriores, para que sean tenidos en cuenta en futuras comparaciones.

Cuando terminé de generar todos los nuevos *itemsets candidatos* (*w* tomó todos los valores posible de *k*) entonces elimino de TRAIN aquellos itemsets candidatos que han completado una pasada.
Para ello, invocamos al método deleteCompleteItemSets pasándole como argumento el conjunto TRAIN.
Dicho método elimina de TRAIN todos los itemsets candidatos que han recorrido y procesado todas las transacciones de la base de datos y no son frecuentes en su período de vida completo, como sigue: por cada *itemset candidato c* de TRAIN, tal que *c* completó una pasada (recorrió todas las transacciones de la base de datos) y no es frecuente en todo su lifespan; verifica si es frecuente, al menos, en algún subintervalo maximal contenido en su lifespan (invocando al método a_posteriori [5] para *c*) y lo agrega en la colección de salida.
Por el contrario, si el *itemset candidato c* no completó una pasada es incluido directamente en la colección de salida sin realizar ninguna verificación.

El método a_posteriori recibe los valores de mínimo soporte, y mínimo soporte temporal, y el histograma *R*; y por cada iteración, define un nuevo subintervalo $[t_i, t_j]$ contenido en el lifespan del itemset y verifica que el itemset sea frecuente en dicho subintervalo.

Veamos un poco más en detalle el funcionamiento de dicho método para entender cuales son las particularidades de la selección de los intervalos maximales.
Inicialmente, define un nuevo intervalo $[t_i, t_j]$, con los mismos valores que el lifespan $[t_1, t_2]$ del itemset.
Luego, por cada intervalo $[t_i, t_j]$ tal que $t_i < t_j$, y la amplitud del intervalo no sea menor que el mínimo soporte temporal, verifica lo siguiente:
- ✓ Si el itemset es frecuente en el subintervalo $[t_i, t_j]$, entonces define una "copia" del itemset, con el intervalo de frecuencia $[t_i, t_j]$, y lo agrega en la colección resultado. Además, asigna en t_i el valor de t_j incrementado en una unidad, ($t_i = t_j$ + *mínima diferencia temporal*); y en t_j el valor del limite superior del lifespan del itemset (t_2). Luego, repite el proceso con el nuevo intervalo, buscando nuevamente desde el final del intervalo (t_2) hasta t_i.
- ✓ En el otro caso, el itemset no es frecuente en el intervalo $[t_i, t_j]$, incrementa en una unidad el valor de t_i ($t_i = t_i$ + *mínima diferencia temporal*), y repite el proceso con el mismo valor de t_j.

Como ejemplo, supongamos que tenemos el itemset *X*, el cual está definido en el intervalo $[t_1, t_2]$.
Inicialmente, tenemos que $[t_i, t_j] = [t_1, t_2]$. Luego, comenzamos a decrementar (quitamos el valor que equivale a la *mínima diferencia temporal*) el valor de t_j y por cada subintervalo generado, calculamos si el itemset *X* es frecuente en dicho subintervalo.
Supongamos que el itemset *X* es frecuente en $[t_i^1, t_j^1]$ y que la amplitud de $[t_i^1, t_j^1]$ es mayor que el umbral para el mínimo soporte temporal, entonces definimos una copia del itemset frecuente *X* con el intervalo $[t_i^1, t_j^1]$ como intervalo de frecuencia y lo agregamos en la colección de salida del método.
Luego, repetimos el proceso de búsqueda tomando el nuevo subintervalo $[t_i, t_j] = [t_j^1, t_2]$.
De la misma manera que en el caso anterior, decrementamos el valor de t_j y por cada nuevo subintervalo generado (de amplitud mayor que el umbral para el mínimo soporte temporal), verificamos que el itemset *X* sea frecuente en dicho subintervalo.
Supongamos ahora que X es frecuente en $[t_i^2, t_j^2]$, entonces definimos una nueva copia del itemset frecuente *X* con el intervalo $[t_i^2, t_j^2]$ como intervalo de frecuencia y lo agregamos en la colección de salida del método.
Continuando con el proceso de búsqueda de los intervalos maximales, tenemos el nuevo subintervalo $[t_i, t_j] = [t_j^2, t_2]$.
Supongamos ahora que el itemset *X* es frecuente en el subintervalo $[t_i^3, t_j^3]$. Luego, generamos una nueva copia de *X* con $[t_i^3, t_j^3]$ como intervalo de frecuencia y lo agregamos en la colección de salida.
Por último, vamos a verificar si *X* es frecuente en el subintervalo $[t_i, t_j] = [t_j^3, t_2]$.
Siguiendo con la metodología antes aplicada, vamos decrementando el valor de t_j, verificando en cada caso que la amplitud del subintervalo generado no sea menor que el umbral para el mínimo soporte temporal definido, y que el itemset *X* sea frecuente en dicho subintervalo.
Supongamos ahora que el itemset *X* no es frecuente en ninguno de los posibles subintervalos generados en $[t_i, t_j] \subseteq [t_j^3, t_2]$ que cumplen con las restricciones de soporte temporal, entonces incrementamos el valor de t_i, y repetimos el proceso con los subintervalos contenidos en $[t_j^3 + \textit{mínima diferencia temporal}, t_2]$.

Este proceso se repite mientras haya algún subintervalo, generado siguiendo el proceso antes descripto, contenido en $[t_1, t_2]$ en donde el itemset X es potencialmente frecuente.

Cuando finaliza la ejecución del método, la colección con los nuevos itemsets frecuentes (copias del itemset *X*, pero con los intervalos de frecuencia actualizados en cada caso) es retornada.

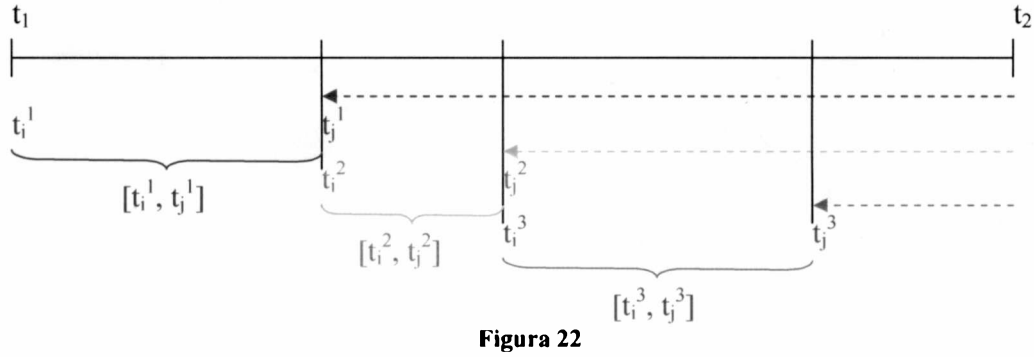

Figura 22
Ejemplo de la ejecución del método a_posteriori

De lo anterior, tenemos que un itemset que no es frecuente en todo su período de vida, puede serlo en uno o más de los intervalos maximales contenidos en su período de vida.

En estos casos, se crea una copia del *itemset original* por cada intervalo de frecuencia maximal con los mismos valores, salvo el intervalo en dónde el itemset es frecuente, el histograma actualizado con las transacciones del nuevo intervalo, y el valor de soporte del itemset en dicho intervalo.

Por lo tanto, tenemos que dicho itemset aparecerá *n* veces (*n* representa la cantidad de subintervalos maximales en donde el itemset es frecuente) en la colección de salida, con los mismos items pero con distinto valor de soporte y soporte temporal de acuerdo al nuevo intervalo de frecuencia.

La colección de valores retornada por el método deleteCompleteItemSets es asignada a TRAIN.
Si el conjunto TRAIN no es vacío, y no llegué al final de la base de datos de transacciones, es decir, quedan transacciones por procesar; entonces se leen otras *M* transacciones de la base de datos y se repite el proceso para computar el soporte de los itemsets candidatos de TRAIN, utilizando las transacciones recientemente leídas.

Cuando llego al final de la base de datos de transacciones, decimos que se ha completado una pasada.
Luego, si el conjunto TRAIN tiene itemsets candidatos, incrementamos el contador de pasada, y repetimos todo el proceso desde el principio de la base de datos de transacciones.
Por el contrario, si el conjunto TRAIN es vacío, finalizo el proceso de búsqueda de los itemsets frecuentes y retorno el conjunto resultado.

En este punto, damos por concluida la primera fase del proceso de búsqueda de las reglas de asociación temporal que consistía en encontrar todos los itemsets frecuentes de la base de datos de transacciones.

Utilizando los itemsets frecuentes encontrados en el proceso anterior, comenzamos con la segunda fase para determinar las reglas de asociación temporal.

De lo expresado en [5], para generar las reglas de asociación temporal, es necesario encontrar, para cada itemset frecuente de tamaño *j*, todos los subconjuntos propios de tamaño j-1 tal que los elementos que lo forman respetan el orden lexicográfico.
De lo anterior tenemos que, dado un itemset frecuente *Z*, debemos encontrar, por cada subconjunto propio *X* de *Z*, las reglas

$$X \Rightarrow (Z\text{-}X)\text{: } fl_z \text{ tal que } s(Z, fl_z) / s(X, fl_z) \geq \theta,$$

donde el conjunto (*Z-X*) tiene sólo un elemento.

En otras palabras tenemos que para todo *j-itemset frecuente* **Z** en $[t_1, t_2]$, con **X** siendo un subconjunto propio de **Z** de tamaño *j*-1, **Y** = (**Z** - **X**) el subconjunto con el item restante, la confianza de la regla **X** => **Y** $[t_1, t_2]$, denotada por *conf*(**X** => **Y**, $[t_1, t_2]$, **d**), es igual a:

$$conf(\boldsymbol{X}=>\boldsymbol{Y}: [t_1, t_2]) = s(\boldsymbol{X} \cup \boldsymbol{Y}, [t_1, t_2]) / s(\boldsymbol{X}, [t_1, t_2])$$

o lo que es lo mismo:

$$conf(\boldsymbol{X}=>\boldsymbol{Y}: [t_1, t_2]) = s(\boldsymbol{Z}, [t_1, t_2]) / s(\boldsymbol{X}, [t_1, t_2]).$$

De la definición anterior, si tenemos la regla de asociación $\boldsymbol{X}=>\boldsymbol{Y}: [t_1, t_2]$, decimos que *X* es la premisa de la regla, y que *Y* es la consecuencia de la regla.

Para llevar a cabo la *fase 2*, utilizamos el método findRulesQuickly, el cual encuentra y retorna todas las reglas de asociación, con las características antes mencionadas, teniendo en cuenta el tiempo como factor para determinar el período de vida en el cual la regla tiene validez.
Tomando la colección de salida retornada en la *fase 1*; por cada valor de *j* mayor a 1, tal que existen *itemsets frecuentes* en la colección, dicho método procesa los *j-itemsets frecuentes* determinando las reglas de asociación a partir de sus items y del intervalo en donde el itemset es frecuente.
Así, por cada *itemset frecuente u* de tamaño *j* invocamos el método generateRules con los siguientes argumentos: el umbral de la mínima confianza, la colección de *itemsets frecuentes* de tamaño (*j*-1) encontrados en la fase 1, un valor booleano indicando si se debe calcular el soporte de la premisa de la regla, y el *histograma R*.

Cuando el *itemset frecuente u* de tamaño *j* recibe la invocación este método, recorre la colección de items que lo forman y calcula las reglas de asociación en el intervalo de frecuencia, tal que por cada item *w* de *u*, se genera :

- un nuevo itemset de tamaño 1, para representar la *consecuencia de la regla*, con el valor del item *w*; y
- un nuevo itemset de tamaño *j*-1, para representar la *premisa de la regla*, con todos los items restantes del itemset *u*.

Una vez creados los itemsets que representan la *premisa* (*premise*) y la *consecuencia* (*consequence*) de la regla; se crea un nuevo objeto de la clase TemporalAssociationRule con el nombre *tar*, con los siguientes valores:

- el *itemset u* a partir del cual se origina la regla,
- la *premisa* y la *consecuencia* de la regla,
- el intervalo de frecuencia del *itemset u* sobre el cual se va a realizar el cálculo,
- la colección de (*j*-1)-*itemsets frecuentes* encontrados en la fase 1, y
- el indicador del cálculo del soporte de la premisa.

Cuando el constructor de la clase TemporalAssociationRule recibe la invocación, internamente almacena el valor de los parámetros recibidos en sus variables de instancia, y setea el valor de la variable *calculatePremiseSupport* con el valor del parámetro que indica si se debe calcular el soporte de la premisa.

Luego se invoca el método confidenceForRule del objeto *tar*, pasándole el *histograma R* como argumento, para calcular la confianza de la regla.
En este punto, dependiendo del valor de la variable *calculatePremiseSupport*, el objeto *tar* calcula la confianza de la regla de la siguiente manera:

✓ Si es verdadero, antes de calcular la confianza de la regla debe calcular el valor de soporte de la premisa en el intervalo de frecuencia de la regla.
Para ello, invoca al método confidenceForRuleCalculateSupportPremise.

- ✓ Si es falso, asume que los itemsets tienen una distribución uniforme en la base de datos, entonces el soporte de un itemset particular no debería variar significativamente de un intervalo a otro.
  Con esta idea, utiliza el mismo valor de soporte que tiene calculado el *itemset premisa* para calcular la confianza de la regla, sin importar el intervalo de frecuencia de la regla.
  En este caso, para calcular la confianza de la regla, invoca al método confidenceForRuleSameSupportPremise.

El método confidenceForRuleSameSupportPremise calcula la confianza de la regla asumiendo una distribución uniforme de los items en la base de datos de transacciones. Para ello, invoca al método setConfidence con el valor de soporte del *itemset origen* de la regla y la colección de valores de soporte de los *itemsets idénticos a la premisa*, como argumentos.

Como vimos en la generación de los itemsets frecuentes, puede darse el caso de que un itemset que no es frecuente en todo su período de vida, lo sea en uno o más de los intervalos maximales que están contenidos en su lifespan.
Vimos también que por cada uno de estos itemsets frecuentes en más de un intervalo maximal, se generaba una copia del *itemset origen* con los valores actualizados de soporte y soporte temporal, la cual es agregada en la colección retornada por el método.

De lo anterior, tenemos que en caso de que el *itemset premisa* de la regla sea frecuente en más de un intervalo maximal, la colección recibida como argumento tendrá los valores de soporte del itemset en dichos intervalos maximales.
En caso contrario, la colección tendrá sólo el valor de soporte del itemset premisa en su período de vida completo (ó en el único intervalo maximal en dónde el itemset es frecuente).

Con estos antecedentes, para calcular la/s confianza/s de la regla, el método setConfidence debe recorrer la colección de valores de soporte del *itemset premisa* invocando al método getConfidence con cada uno de los valores de la colección y el valor de soporte del *itemset origen* de la regla.
Dicho método, en cada invocación toma el valor de soporte del *itemset origen* de la regla y lo divide por el valor de soporte de la *premisa*, y retorna el valor resultado.
Por último, este valor computado es almacenado en la colección de confianzas de la regla.

De manera similar actúa el método confidenceForRuleCalculateSupportPremise para calcular la confianza de la regla, pero con la salvedad que en este caso no se asume una distribución uniforme de los items en las transacciones de la base de datos. Por esta razón, el soporte de los itemsets premisa en el intervalo de validez de la regla debe ser calculado antes de computar la confianza de la regla.

Al igual que en el caso anterior, para calcular la/s confianza/a de la regla, se invoca al método setConfidence pasándole el valor de soporte del *itemset origen* de la regla junto con la colección de valores de soporte del itemset premisa en el intervalo de validez de la regla.

Para ello, invoca al método calculateSupportValue, con el itemset premisa de la regla y el *histograma R* como argumentos.
Este método calcula y retorna la colección de valores de soporte de los itemsets premisa (recordar que pueden existir varios itemsets "idénticos", pero con distintos valores de soporte y soporte temporal, los cuales han sido generados en la fase 1), invocando al método getSupport, en el intervalo de validez de la regla.

Cada uno de los itemsets que coinciden con la premisa, recibe la invocación del método getSupport con el intervalo de validez de la regla y el *histograma R* como argumentos y retorna el valor de soporte del itemset en dicho intervalo.
Para ello, invoca al método updateTransactionCounterInsideInterval el cual calcula la cantidad de transacciones contenidas en el histograma del itemset premisa actual en el intervalo de validez de la regla, a partir de ahora *counterFr*, y la cantidad de transacciones contenidas en el *histograma R* para el intervalo de validez de la regla, a partir de ahora *counterFTr*.
Luego, invoca al método getSupport el cual devuelve el valor de soporte del itemset calculando el cociente entre los valores *counterFr* y *counterFTr*.

Una vez calculado el valor de soporte del *itemset premisa actual* en dicho intervalo, es agregado en la colección de salida del método calculateSupportValue.
Cuando dicho método finaliza su ejecución, la colección con los soportes de las premisas en el intervalo de validez de la regla es retornada.
De la misma manera que ocurre en el caso anterior, el método setConfidence se utiliza para calcular la confianza de la regla con cada uno de los valores de soporte calculados.

Independientemente del método utilizado para calcular la/s confianza/s de la regla en el intervalo de validez, se agrega la regla de asociación temporal, con todos sus valores de confianza calculados, en la colección de reglas del itemset y se procede a computar la confianza de la siguiente regla generada a partir de sus items.

Cuando el itemset termina de calcular la confianza de todas las reglas de asociación temporal generadas a partir de sus items, invoca el método pruneRules, pasándole la colección de reglas encontradas y el umbral para la mínima confianza, para *"podar"* las reglas de asociación temporal cuya confianza está por debajo del umbral definido por el usuario.
Dicho método, simplemente elimina los valores de confianza de las reglas de asociación temporal que están por debajo del mínimo valor especificado por el usuario.
Si todos los valores de confianza de la regla de asociación temporal son menores al umbral para la mínima confianza, entonces la regla de asociación no es tenida en cuenta.
Por el contrario, si uno o más valores de la regla de asociación temporal están por encima (o son iguales) al valor especificado por el usuario, entonces la regla es retornada por el proceso.

Cuando finaliza el proceso de poda de las reglas de asociación temporal generadas a partir del itemset, el método generateRules retorna la colección de reglas de asociación temporal cuya confianza no es menor que el umbral para la mínima confianza.

Las reglas de asociación temporal del itemset son incluidas al final de la colección de salida del método findRulesQuickly, y se repite el proceso con el siguiente itemset frecuente para generar sus reglas.

Una vez que han sido procesados todos los itemsets frecuentes generando sus reglas de asociación temporal, damos por finalizada la ejecución del método findRulesQuickly.

Luego, almacenamos la fecha y hora de finalización del proceso de búsqueda de los itemsets frecuentes y generación de las reglas de asociación temporal, y damos por finaliza la ejecución del método buildAssociations de la clase TDIC.
En este momento, damos por finalizado de análisis de la base de datos sintética, y los datos obtenidos del procesamiento son visualizados en pantalla.

Simultáneamente con la visualización en pantalla de los resultados obtenidos una vez finalizado el proceso, se genera de forma automática un archivo con los mismos datos de proceso. Dicho archivo se guarda en el path especificado por el usuario en el archivo de configuración.
Esta nueva funcionalidad ha sido incluida en la aplicación para permitir la ejecución múltiple de procesos asociados a la misma base de datos sintética, sin tener necesidad de detener el proceso para almacenar los resultados obtenidos cada vez que el mismo finalizaba, dando lugar a la ejecución de la próxima cadena de parámetros del archivo de configuración.
Para obtener más información sobre este tema, ver el apartado "**Ejecución múltiple de procesos**" en la página 92.

Como vimos, el prototipo desarrollado nos permite obtener itemset frecuentes que con los modelos de generación convencionales, no podrían haber sido generados, en su gran mayoría por las limitaciones de soporte, ya que al no tener en cuenta el tiempo de vida de cada itemset en particular, se están considerando transacciones en las cuáles el itemset no tiene validez, y por lo tanto, se están teniendo en cuenta intervalos de validez y de frecuencia que generalmente perjudican al itemset.

El descubrimiento del carácter temporal de los objetos es de vital importancia a la hora de encontrar los itemsets candidatos de la base de datos, determinar si los mismos son frecuentes y generar las reglas de asociación.
Utilizando el concepto temporal de los objetos podemos identificar el instante de tiempo en el que cada objeto tuvo lugar, permitiendo realizar el filtrado de los mismos tomando sólo los objetos que son de interés en un intervalo particular de estudio.

Como vimos, el factor tiempo está asociado a los componentes principales del prototipo, a saber: itemsets, histogramas, intervalos, y reglas de asociación.
La utilización del tiempo en el proceso de búsqueda y generación de los itemsets frecuentes y las reglas de asociación nos permite estar un escalón por encima de las implementaciones convencionales ampliando el abanico de posibilidades y focalizando nuestro interés en los intervalos que son de calidad superior ya que nos permiten conocer con mayor exactitud el ciclo de vida de los items en la base de datos.

Otro de los conceptos fundamentales en el descubrimiento de los itemsets frecuentes y la generación de las reglas de asociación temporal, es la definición, mantenimiento y consulta del histograma.
Este simple concepto permite conocer en cualquier momento del proceso de análisis, el comportamiento temporal del itemset propietario del histograma. Dicho de otra manera, un histograma es como una radiografía del itemset en la base de datos ya que almacena el contador con las transacciones, cuyos identificadores temporales están comprendidos en el período de vida (*lifespan*) del itemset.
Más aún, un histograma es el componente principal para determinar el soporte del itemset en la base de datos durante el proceso de generación de los itemsets frecuentes y la generación de las reglas de asociación.

Con las modificaciones introducidas en el archivo de parametrización del prototipo, es posible ejecutar el algoritmo seleccionado con diferentes parámetros, y comparar los resultados obtenidos.
Además, podemos "retomar" la ejecución del proceso en el punto en que habíamos concluido anteriormente, utilizando los archivos temporales generados en el *filesystem*, obteniendo resultados parciales de proceso, con la posibilidad de realizar *backups* de los datos generados por el proceso de análisis de la base de datos, utilizando el algoritmo de asociación más conveniente para el estudio.

Estas y otras nuevas funcionalidades hacen del prototipo desarrollado una herramienta muy eficiente y de suma utilidad a la hora de la selección del algoritmo de asociación más conveniente para el análisis de la base de datos de interés.

Modelos de datos

Con el objetivo de implementar los cambios en la aplicación *Weka* original, tratando de mantener la funcionalidad base, fue necesario modificar algunas de las clases existentes, teniendo como premisa que las modificaciones introducidas no interfieran con el funcionamiento original de las mismas; generar nuevas clases a partir de las clases existentes con el objetivo de representar el comportamiento temporal de los objetos durante el proceso de búsqueda y generación de las reglas de asociación temporal.

De lo anterior, podemos agrupar las nuevas clases incorporadas de la aplicación en dos conjuntos: *clases derivadas de clases existentes* y *clases no derivadas de clases existentes*.

Clases derivadas de clases existentes

El grupo de las *"clases derivadas"* está formado bien por clases que heredan su comportamiento de clases definidas en el código base de la aplicación *Weka*, ó bien por clases que tienen un funcionamiento similar a una clase del código base, pero que han sido generadas utilizando el comportamiento previamente definido, incluyéndole algunas modificaciones para representar el carácter temporal de las mismas.

En general, este tipo de clases conservan el mismo nombre que la clase de la cual "*copian*" su comportamiento, anteponiendo el prefijo "T" para hacer notar la funcionalidad temporal que poseen.

A continuación, se listan las clases que han sido incluidas en este grupo, junto con una breve descripción de su funcionalidad y componentes.

GUIChooser (GUI)
Definida en el paquete *weka.gui*. Esta clase es la GUI de inicio de la aplicación y selección del proceso a realizar.

En este caso, no se definió una nueva clase, sino que se modificó la clase original para incluirle la nueva funcionalidad deseada.
Estas modificaciones tienen que ver con la inclusión de un nuevo botón que permite la visualización de la GUI para realizar la *exploración temporal* y el procesamiento de la base de datos seleccionada, a través de la búsqueda de los itemsets frecuentes y la generación de las reglas de asociación temporal.

Por definición, esta clase puede ser ejecutada desde la línea de comando ya que define e implementa el método main.

TExplorer (GUI)
Definida en el paquete *weka.gui.explorer*. Esta clase es la versión temporal de la clase Explorer y es la clase principal del entorno de exploración de *Weka*.

Entre sus funciones más relevantes, tenemos que permite al usuario crear, abrir, guardar, configurar la base de datos (asociaciones) y desarrollar el análisis de performance de los datos con las características temporales introducidas.

El método más importante de esta clase es el constructor, el cual crea la GUI (ambiente) de exploración e inicializa el estado de cada una de las "pestañas" con los valores por defecto.

TPreprocessPanel (GUI)
Definida en el paquete *weka.gui.explorer*. Es la versión temporal de la clase PreprocessPanel.

Esta GUI controla el preprocesamiento simple de las instancias. En otras palabras, esta interfase permite la selección y preprocesamiento de las asociaciones (bases de datos) a analizar, posibilitando la inclusión/exclusión de los atributos de la base de datos para el proceso, aplicando los diferentes filtros disponibles de acuerdo al tipo de preprocesamiento a realizar sobre el conjunto de transacciones existentes.

Entre las modificaciones introducidas a esta clase, tenemos que en la selección de la base de datos a analizar (procesar) se filtra para que se visualicen sólo los archivos con extensión *.tgf*, asegurándonos de que el archivo seleccionado contará con las características temporales que son necesarias a la hora de realizar el análisis de sus datos.

Entre los métodos más importantes de la clase tenemos:
- los constructores que crean, configuran y conectan todos los componentes del panel y definen las características de visualización del mismo en el momento de la instanciación del objeto;
- los métodos que permiten setear y recuperar el conjunto de instancias sobre el cual se está trabajando;
- el método filterInstances que filtra el conjunto de instancias que recibe como argumento de acuerdo al filtro previamente definido;
- los métodos que permiten la carga del conjunto de instancias base o de trabajo de diferentes medios (query, url a base de datos, url a query, archivo *.tgf*); y
- el método getPropertiesProccessFile que se ejecuta al final del constructor de la clase y recupera del archivo de propiedades *trabajoDeGrado.props* todas las variables de entorno definidas por el usuario, que serán utilizadas como parámetros de configuración en todos los procesos.

TAssociationsPanel (GUI)
Definida en el paquete *weka.gui.explorer*. Es la versión temporal de la clase AssociationsPanel.

Esta GUI permite al usuario seleccionar, configurar y ejecutar un esquema que obtenga conocimiento interpretando asociaciones (bases de datos) cuyas transacciones poseen características temporales.

La modificaciones introducidas están relacionadas con la selección de los algoritmos de procesamiento de las asociaciones y la forma en la cual se muestran los resultados obtenidos.
Además, esta clase es la responsable de llevar a cabo los múltiples procesos de análisis sobre la base de datos seleccionada, dependiendo de los parámetros del archivo de

configuración; y de realizar el almacenamiento de los archivos que contienen la información obtenida del proceso, una vez finalizado la ejecución del algoritmo seleccionado.

TAttributeSelectionPanel (GUI)
Esta clase ha sido definida en el paquete *weka.gui* y es la versión temporal de la clase AttributeSelectionPanel.

La función principal de esta clase es crear un panel que muestra los atributos contenidos en un conjunto de instancias (transacciones), permitiendo al usuario indicar si cada atributo es seleccionado o no para ser incluido en el análisis.

Los métodos principales de la clase son los que están relacionados con la selección del conjunto de instancias a procesar y la identificación de sus atributos. A saber:
- el método setInstances el cual recibe un objeto de la clase TInstances y setea las transacciones (instancias) cuyos nombres de los atributos serán visualizados; y
- el método getSelectedAttributes, el cual retorna una lista conteniendo los atributos de la base de datos que han sido seleccionados para ser incluidos en el proceso de análisis.

TAttributeSummaryPanel (GUI)
Definida en el paquete *weka.gui*. Esta clase es la versión temporal de la clase AttributeSummaryPanel.

Esta GUI muestra el resumen de las estadísticas de un atributo (nombre, tipo, cantidad de valores erróneos, cantidad de valores distintos, etc).
En el caso de los atributos numéricos da alguna otra estadística como puede ser el significado o la desviación estándar; mientras que para los atributos nominales retorna la cantidad de cada valor que toma el atributo.

Los métodos más importantes son:
- el constructor de la clase que inicializa todos los objetos internos de la intancia y define la estética del panel;
- el método setInstances que setea la base de datos sobre la cual se van a realizar las estadísticas; y
- los métodos setAttribute, setDerived, setTable y setHeader que permiten setear, definir y manipular el estado de los atributos de la instancia recibida a fin de realizarle las estadísticas dependiendo del tipo de valores que cada uno de ellos puede almacenar.

TGenericArrayEditor (GUI)
Definida en el paquete *weka.gui*. Es la versión temporal de la clase GenericArrayEditor.

Esta clase es un editor de propiedades para colecciones de objetos que a su vez contienen editores de propiedad.
La función principal de los objetos de esta clase está relacionada con el manejo de las propiedades de objetos de una colección.

Esta GUI tiene definido dos botones que permiten agregar o eliminar las propiedades de un objeto. Luego, dependiendo del tipo de objeto seleccionado será la acción que se tome en cada caso.

TGenericObjectEditor (GUI)
Definida en el paquete *weka.gui*. Esta clase es la versión temporal de GenericObjectEditor.

Esta clase es un editor de propiedad para objetos que han sido definidos como "*editables*" en el archivo de configuración *TGenericObjectEditor* el cual lista los posibles valores que pueden ser seleccionados y configurados por el propio objeto.

El archivo de configuración es llamado *TGenericObjectEditor.props* y puede residir en cualquier lugar determinado por la variable "*user.home*" o el directorio actual, y un archivo de propiedades por defecto es leído desde el código base provisto por *Weka* en su distribución original.

Para aumentar la velocidad, el archivo de configuración es leído sólo una vez cuando la clase es inicialmente cargada.
Cuando un objeto de la clase es instanciado, se carga la configuración del archivo de propiedades provista por el archivo ubicado en la dirección definida por la constante PROPERTY_FILE.
La información contenida en este archivo es utilizada para cargar las diferentes listas de selección de acuerdo a los valores requeridos en cada caso.

La estructura general del archivo de configuración *TGenericObjectEditor.props* es la siguiente:
- Lista de *ResultProducers*
- Lista de *ResultListeners*
- Lista de *Classifiers*
- Lista de *DistributionClassifiers*
- Lista de *Attribute Selection Evaluators*
- Lista de métodos *Attribute Selection Search*
- Lista de *Clusterers*
- Lista de *SplitEvaluators*:

```
weka.experiment.SplitEvaluator=\
      weka.experiment.ClassifierSplitEvaluator, \
      weka.experiment.CostSensitiveClassifierSplitEvaluator, \
      weka.experiment.RegressionSplitEvaluator, \
      weka.experiment.TInstanceQuery
```

- Lista de *Filters*:

```
weka.filters.TFilter = \
      weka.filters.TAttributeFilter
```

- Lista de *Associators*:

```
weka.associations.TAssociator=\
      weka.associations.TDIC, \
      weka.associations.AprioriTG
```

- Lista de *Loaders*:

```
weka.core.converters.TLoader =\
      weka.core.converters.TGLoader
```

TInstancesSummaryPanel (GUI)
Esta clase ha sido definida en el paquete *weka.gui*. Es la versión temporal de la clase InstancesSummaryPanel.

Esta GUI se utiliza para visualizar el resumen de la información de una instancia (base de datos) particular.
La información mostrada en el resumen de la instancia consiste en el nombre, el número de instancias, y el número de atributos de la base de datos importada.

Los métodos más importantes de esta clase son:
- el constructor de la clase, el cual inicializa todo el contenido del panel y sus componentes de visualización, y
- el método setInstances, el cual recibe la base de datos en cuestión y recupera la información a visualizar.

TLoader
Esta interfase ha sido definida en el paquete *weka.core.converters* y es la versión temporal de la clase Loader.
Al igual que su predecesor, se utiliza para cargar instancias, en este caso con características temporales, desde un origen o entrada en algún formato.

Los métodos más importantes son:
- el método setSource que inicializa el objeto y setea el origen del *dataset* que es recibido como argumento;
- el método getStructure el cual determina y retorna (si es posible) la estructura (obtenido a través del *header*) del *dataset* como un conjunto vacío de instancias;
- el método getDataSet el cual retorna el *dataset* completo; y
- el método getNextInstance el cual lee el *dataset* incrementalmente retornando la siguiente instancia o retornando *null* si no hay más instancias para retornar.

TAbstractLoader
Definida en el paquete *weka.core.converters*, es una clase abstracta para los *Loaders* que contienen las implementaciones por defecto de los métodos *setSource*.
Esta clase abstracta implementa la interfase TLoader, por esta razón, provee implementaciones para los métodos, definidos como abstractos, de dicha interfase.

TGLoader
Definida en el paquete *weka.core.converters* y es una subclase de TAbstractLoader.
Los objetos de esta clase se utilizan para leer un archivo de texto con extensión *.tgf*, es decir, una base de datos de transacciones con características temporales.

Los métodos más importantes de esta clase son:
- el método setSource el cual recibe un objeto File, restaura el objeto y setea el origen del *dataset* proporcionado;
- el método getStructure el cual determina y retorna (si es posible) el *dataset* como un conjunto vacío de instancias;
- el método readStructure el cual define la estructura del *dataset* leyendo su cabecera;

- el método getDataSet el cual retorna el *dataset* completo;
- el método getInstance el cual intenta parsear una línea del *dataset*; y
- el método checkStructure que chequea la estructura de la colección de objetos que recibe.

TInstance

Esta clase está definida en el paquete *weka.core.* y puede verse como la versión temporal de la clase Instance.
Al igual que la clase Instance, esta clase permite manipular una instancia, es decir, una transacción de la base de datos.

Cada uno de los objetos de esta clase almacena la colección de atributos de tipo string que representan los items (productos) incluidos en la transacción, un identificador unívoco, y un identificador temporal que representa el momento en el cual la transacción ocurrió en la base de datos.
Además, cada una referencia a la instancia de la base de datos (objeto de la clase TInstances) a la que pertenece.

Como su nombre lo indica, el *identificador unívoco* de la instancia permite identificar unívocamente la transacción en la base de datos. Por esta razón, este identificador no puede repetirse dentro del mismo conjunto de instancias.
Por otro lado, tenemos el *identificador temporal* de la instancia, el cual representa el momento en el cual la instancia tuvo lugar.

Analizando el problema conocido como *canasta de mercado*, supongamos que el *identificador temporal de la transacción* se define como el momento en el cual el cliente abona su compra.
Luego, tenemos que es altamente probable que, si existen varias (al menos tres) cajas registradoras encargadas de almacenar la información referente a los productos comprados por cada cliente, puede darse el caso de que dos consumidores pueden estar pagando la mercadería comprada (no necesariamente la misma) en el mismo instante de tiempo, entonces ambas transacciones tendrán el mismo identificador temporal.
Por esta razón, decimos que el identificador temporal de la transacción no es unívoco.

Por definición, cada instancia de la base de datos tendrá una estructura como se indica a continuación:

- en la primera posición de la transacción se indica el identificador unívoco.
- en la segunda posición de la transacción se indica el identificador temporal.
- a partir de la tercera posición de la transacción (de izquierda a derecha) se indican los valores de los atributos, separados por coma, ordenados según el orden lexicográfico.

Entre los métodos más importantes de esta clase tenemos:

- los constructores de la clase que crean el objeto e inicializan el estado interno del mismo,
- el método classIndex el cual retorna el índice (posición) del identificador unívoco dentro de la instancia,
- el método classTemporalIndex el cual retorna el índice (posición) del identificador temporal dentro de la instancia,

- los métodos classValue y classTemporalValue que retornan el valor del identificador unívoco y el valor del identificador temporal de la transacción,
- los métodos para insertar, modificar y eliminar el valor de un atributo de la transacción,
- el método numValues el cual retorna la cantidad de atributos de la transacción, y
- el método replaceMissingValues el cual se utiliza para reemplazar todos los valores inválidos de la transacción por los valores del vector que recibe como argumento.

De acuerdo a la estructura de los archivos *.tgf* de base de datos, se ha modificado la clase original para que la colección de atributos de la instancia (transacción) sean de tipo *string*, en lugar de valores en punto flotante como había sido definido inicialmente.

Además, teniendo en cuenta que, en general, las transacciones tendrán una cantidad diferente de atributos, la colección de valores es de longitud variable por cada transacción, y la posición que ocupa un valor particular en la colección dependerá de los valores "menores" (precedentes de acuerdo al orden lexicográfico) que estén definidos en la transacción de la base de datos.

TInstances

Esta clase ha sido definida en el paquete *weka.core* y puede ser vista como la versión temporal de la clase Instances.

El objetivo principal de esta clase es almacenar un conjunto ordenado de instancias identificadas en el tiempo. En otras palabras, esta clase representa la base de datos de transacciones, donde cada una de las transacciones tiene definido, además de los atributos que la componen, un identificador temporal que representa el instante de tiempo en el cual la transacción se produjo.

De acuerdo a la forma en la cual se construye la base de datos de transacciones, tenemos que la colección de instancias (objetos de la clase Instance) de la base de datos está ordenada en primer lugar por el identificar unívoco, en orden ascendente, y luego, por el identificador temporal de cada instancia dentro de la base de datos.
Con esta estructura, tenemos que todas las transacciones que tienen el mismo identificador temporal aparecerán consecutivas en la base de datos.

Además, los atributos de cada instancia estarán ordenados de acuerdo al orden lexicográfico, lo cual servirá para la búsqueda de los atributos dentro de cada instancia.
Como vimos, el conjunto de transacciones de la base de datos es almacenado dentro del objeto como una colección de instancias.
Para almacenar las transacciones de la base de datos en la colección debemos procesar un archivo con formato *tgf* que contiene todas las transacciones de interés.
Leyendo el *header* de dicho archivo podemos determinar el nombre lógico (*tag @relation*) de la base de datos y el conjunto de transacciones que la constituyen la base de datos propiamente dicha (*tag @data*), las cuales están identificadas en el tiempo.

Por definición, tenemos que cada transacción de la base de datos está definida en una línea del archivo, donde los caracteres RETORNO_DE_CARRO o FIN_DE_LINEA determinan el final de la transacción.

De esta manera, si procesamos cada una de las líneas del archivo, podemos ir generando y almacenando las instancias (transacciones) que forman la base de datos.
Una vez que el carácter FIN_DE_ARCHIVO es alcanzado, tenemos la certeza de que ha finalizado el archivo y podemos dar por finalizado el proceso de importación.

Los objetos de esta clase contienen, además del conjunto de las instancias de la base de datos; la colección de atributos de la base de datos con nombres genéricos (recordar que no existe el *tag @attribute* porque las transacciones de estos archivos tienen diferente cantidad de items); un indicador del índice (posición) del identificador unívoco; y un indicador del índice (posición) del identificador temporal de la transacción.

Podemos clasificar los métodos más importantes de la clase en tres grupos, de acuerdo a la funcionalidad para la cual han sido desarrollados, a saber:

- ✓ **métodos de propósito general:** los métodos pertenecientes a este grupo son utilizados para la creación y la comparación de los objetos de la clase, y para manipular la colección de instancias de la base de datos. A saber:
  - los constructores que permiten crear un objeto de esta clase a partir de un archivo de base de datos (con extensión *.tgf*);
  - los métodos que permiten setear y retornar el índice (posición) del identificador y el índice (posición) del identificador temporal de las transacciones;
  - el método enumerateInstances que devuelve una lista con las instancias de la base de datos;
  - los métodos para agregar, eliminar y modificar una instancia en particular;
  - los métodos para setear y recuperar los atributos de una instancia particular;
  - el método equalHeaders para chequear si dos instancias tienen el mismo *header*;
  - el método readHeader que se utiliza para leer la información cabecera del archivo a importar y genera la colección de atributos encontrados en la base de datos;
  - el método instance que devuelve la instancia de la colección que se encuentra en la posición que recibe como argumento;
  - los métodos firstInstance y lastInstance que retornan la primera y última instancia de la base de datos, respectivamente;
  - el método sort que permite ordenar las instancias de acuerdo al atributo pasado como argumento.

- ✓ **métodos de generación y manipulación del *dataset*:**
  - el método getInstance que lee una única instancia utilizando el *tokenizer*, y la agrega a la base de datos invocando al método getInstanceFull el cual lee el resto de las transacciones de la base de datos y las agrega en el conjunto de instancias a procesar; y
  - los métodos getFirstToken, getLastToken y getNextToken que recuperan el primer, último y siguiente *token* (marca) de la instancia (transacción) que estamos procesando;
  - el método getNextInstance el cual lee una transacción (línea) del archivo, actualiza el contador de transacciones leídas, la guarda en el dataset y devuelve un valor de verdad indicando si hay más transacciones en el archivo para procesar.

- ✓ **métodos utilizados en el proceso de generación de los itemsets frecuentes y la búsqueda de las reglas de asociación temporal:**
  - el método initializeProcess que inicializa el estado interno del objeto, carga nuevamente el archivo de transacciones y lee la información contenida en el *header*;
  - el método numInstances el cual retorna la cantidad (actual) de instancias del dataset;
  - el método numTotalInstances que devuelve la cantidad total de instancias recuperadas del archivo importado inicialmente;
  - el método existsNextInstance que verifica la existencia de transacciones en el archivo que aún no han sido procesadas, comparando el contador de transacciones leídas con el valor devuelto por el método numTotalInstances;
  - el método instance que devuelve la instancia del *dataset* ubicada en la posición que recibe como argumento; y
  - el método elements que retorna el *dataset* completo.

La utilización de los métodos numInstances y numTotalInstances es de suma importancia en el proceso de generación de los itemsets frecuentes.
Por esta razón y para evitar todo tipo de dudas respecto de la funcionalidad de los mismos, veamos un ejemplo.

Supongamos que estamos procesando un archivo de base de datos con 1000 transacciones, las cuales son procesadas en grupos de 100 (definido por el usuario), entonces el método numTotalInstances siempre retornará 1000 (este valor fue inicialmente calculado, cuando se importo el archivo completo para recuperar la información de los atributos, en el método getInstanceFull); mientras que el método numInstances retornará la cantidad actual de transacciones leídas (100 o menos si estamos en la última lectura y la base de datos tuviera un número de transacciones que no es múltiplo de 100).

TAssociator
Esta clase está definida en el paquete *weka.associations* y es la versión temporal de la clase Associator.

Al igual que la clase de la cual "*copia*" su comportamiento, esta clase es un esquema abstracto para algoritmos de aprendizaje. Todos los esquemas de algoritmos temporales de aprendizaje implementan esta clase.

El método más importante de esta clase es buildAssociations el cual recibe un objeto de la clase TInstances (la base de datos de transacciones) y genera un asociador.

Por definición, las implementaciones de este método deben inicializar todas las variables del objeto que están siendo seteadas a través de opciones.
También por definición, este método no debe alterar y/o cambiar la base de datos que recibe.
Subclases de la clase TAssociator son las clases ***AprioriTG*** y ***TDIC***.

AprioriTG
Definida en el paquete *weka.associations*, esta clase hereda su comportamiento de la clase TAssociator.
Por definición, esta clase implementa el algoritmo Apriori [1], el cual iterativamente reduce el mínimo soporte hasta que encuentra el número requerido de reglas con la mínima confianza definida por el usuario, tomando como entrada un archivo *.tgf*.

Las modificaciones introducidas en esta clase, con respecto a la clase Apriori original, están relacionadas con la forma en la cual se recuperan los items de cada transacción y la manera en la que se cuenta el soporte de los itemsets.
Además, con las modificaciones introducidas, es posible realizar el procesamiento de los archivos *.tgf* con sus características particulares, ya que en este caso, no tenemos la definición en la cabecera del archivo de los items presentes en las transacciones, más aún, no conocemos de antemano los items que están presentes en la base de datos de transacciones como se asumía en la versión original.
Otra de las modificaciones realizadas permite el procesamiento de bases de datos con atributos de tipo *string*, que en la versión inicial era una restricción fuerte a la hora de procesar los datos del archivo.

En cuanto al funcionamiento general de archivo, el mismo ha sido mínimamente alterado permitiendo realizar la comparación del procesamiento de una base de datos seleccionada, utilizando el *algoritmo Apriori con características temporales* (AprioriTG) y el algoritmo desarrollado en el prototipo (TDIC).

ItemSetTG
Esta clase ha sido definida en el paquete *weka.associations*.
Tal cual ocurre con las clases Apriori y AprioriTG; esta clase es la versión temporal de la clase ItemSet que es utilizada cuando se ejecuta el método buildAssociations de la clase AprioriTG.

Comparando ambas clases, vemos que el funcionamiento es similar, salvo que se le han realizado algunas modificaciones para que el algoritmo Apriori pueda ser comparado con el algoritmo TDIC.

La función principal de los objetos de esta clase es almacenar un conjunto de items (cadena de caracteres). Generalmente, el conjunto de items está ordenado por el orden lexicográfico, el cual está determinado por la información cabecera del conjunto de instancias usadas en la generación.

Los métodos más importantes de esta clase son:
- los métodos para manipular la colección de items del itemset,
- el método containedBy utilizado para verificar si el itemset está contenido en una instancia (transacción) particular,
- los métodos deleteItemSets y pruneItemSets que permiten filtrar y manipular un conjunto de itemsets,
- los métodos singletons, upDateCounter y upDateCounters utilizados durante el proceso de generación de los itemsets frecuentes,

- el método generateRules utilizado para generar las reglas de asociación a partir del itemset y el método pruneRules utilizado para eliminar las reglas de asociación que no cumplen con las restricciones de mínima confianza, y
- los métodos confidenceForRule, liftForRule, leverageForRule, y convictionForRule utilizados para analizar las reglas de asociación obtenidas a partir del itemset.

En este caso, las modificaciones introducidas están orientadas al manejo de bases de datos con transacciones que, en general, no tienen la misma cantidad de atributos y los mismos son de tipo *string*.

TItemSet

Es la versión temporal de la clase ItemSet. Al igual que su predecesor, está definida en el paquete *weka.associations*.
Esta clase es una de las clases principales del prototipo y se utiliza para almacenar el contenido de una transacción con características temporales.

Internamente, los objetos de esta clase están formados por:

i. una colección de atributos (de tipo *string*), ordenados según el orden lexicográfico.
ii. el intervalo de tiempo en el cual está definido (*timestamp*), es decir, el intervalo cuyo límite inferior es el identificador temporal de la primera transacción en la que el itemset ocurrió, y como límite superior el identificador temporal de la última transacción en la que el itemset ocurrió;
iii. la cantidad de ocurrencias del itemset en la base de datos, dentro del *timestamp* en el cual está definido;
iv. el intervalo de frecuencia del itemset;
v. el *histograma* del itemset; y
vi. los valores que indican el *soporte* y el *soporte temporal* del itemset en la base de datos, luego de ser procesado.

Los métodos más importantes de la clase son:

- los constructores que inicializan el estado por defecto del itemset;
- los métodos getSupport, getSupport(TInterval, Histogram), getSupportValue, updateTransactionCounterInsideInterval, calculateSupport, calculateSupport(TInterval, Histogram) relacionados con el cálculo del soporte;
- los métodos getTemporalSupport, calculateTemporalSupport, calculateTemporalSupport(TInterval), getTemporalSupportValue relacionados con el cálculo del soporte temporal;
- los métodos para setear y retornar los valores de los atributos;
- los métodos upDateCounter(TInstance), increaseCounterFr, decreaseCounterFr, lifespan, updateUpperLimit, updateLowerLimit, isMenor, counterFr, counterFTr, complete, setComplete, calculateCounterFTr, itemsCounter, largeInterval, a_posteriori, isFrecuent, updateSingleton, updateLargeIntervalsCounterFTr, addElementHistogram, key, generateRules, pruneRules utilizados en la ejecución de los algoritmos de búsqueda de itemsets frecuentes y generación de reglas de asociación temporal; y
- los métodos hashCode y equals utilizados para almacenar el itemset en las *tablas de hash*, que retornan la clave (colección de los items que lo forman, separados por coma) del itemset y verifican la igualdad entre dos *itemsets*, respectivamente.

Debido a la importancia que tiene la utilización de esta clase en todo el proceso de búsqueda de los itemsets frecuentes, y generación de reglas de asociación temporal, ha sido necesario introducirle varias modificaciones para permitir, en primer lugar el almacenamiento y manejo de items como cadena de caracteres; en segundo lugar, la asociación de las variables y comportamiento necesario para representar el carácter temporal del itemset.
Por último, todas las modificaciones necesarias para utilizar el itemset como contenedor de la información relevante cuando se ejecuta el algoritmo TDIC [3].

TAttributeFilter
Definida en el paquete *weka.filters*. Esta clase es la versión temporal de la clase AttributeFilter y extiende la clase TFilter.
Los objetos de esta clase son (instancias de) filtros que borran un rango de atributos de la base de datos.

Entre los métodos más importantes de la clase podemos listar los siguientes:
- el método setInputFormat, que setea el formato de entrada de las instancias que recibe como parámetro;
- el método input, que ingresa una instancia para filtrar. Generalmente, la instancia es procesada y transformada a disponible para la salida inmediatamente. Algunos filtros requieren que todas las instancias (transacciones) de la base de datos sean leídas antes de producir la salida;
- el método setAttributeIndices que setea que atributos serán eliminados; y
- el método getAttributeIndices que retorna el rango de selección actual.

TFilter
Esta clase está definida en el paquete *weka.filters* y puede verse como la versión temporal de la clase Filter.
Es una clase abstracta para filtrar instancias (transacciones), es decir, objetos que toman instancias como entrada, realizan alguna transformación sobre la instancia y retornan la instancia.

Las implementaciones de los métodos en esta clase, asumen que la mayoría del trabajo será realizado en los métodos redefinidos e implementados por las subclases.

Los métodos más importantes son:
- el método setOutputFormat el cual recibe las instancias a filtrar y les setea el formato de salida (la clase derivada debe utilizar este método una vez determinando el formato de salida);
- el método setInputFormat el cual recibe un conjunto de instancias y les setea el formato de entrada;
- el método getOutputFormat que devuelve el formato de salida de las instancias;
- el método getInputFormat que retorna el conjunto actual de instancias, las cuales están siendo filtradas por el formato de entrada; y
- el método input que recibe una instancia (transacción) para filtrar; y
- el método output que devuelve la instancia después de realizar el filtrado;
- el método filterFile utilizado para testear filtros desde la línea de comandos; y

- el método batchFilterFile utilizado para testear filtros disponibles para procesar múltiples archivos batch.

TSparseInstance
Definida en el paquete *weka.core*, extiende la clase TInstance y puede verse como la versión temporal de la clase SparseInstance.
Esta clase almacena una instancia como un vector desordenado.
Una instancia esparcida (no ordenada) sólo requiere almacenamiento para sus atributos que son distintos de nulo.

Dado que el objetivo principal de esta clase es reducir el almacenamiento requerido para los *datasets* que tienen un gran número de valores por defecto, también incluye los atributos nominales.

Entre los métodos más importantes de la clase podemos notar:
- los constructores de la clase que generan una instancia esparcida a partir de la instancia o de un conjunto de atributos que recibe como argumento; y
- los métodos para manipular los atributos de la instancia.

Clases no derivadas de clases existentes

ClsInicio (GUI)
Definida en el paquete *weka.gui*. Este panel es utilizado como interfase de presentación del prototipo desarrollado.

El método más importante es el constructor de la clase el cual inicializa los componentes a ser visualizados y muestra automáticamente la ventana que permite seleccionar el tipo de análisis (*exploración o experimentación*) que el usuario desea llevar a cabo.

ClsGeneral
Definida en el paquete *weka.core*. Esta clase es utilizada para almacenar variables y métodos de propósito general, los cuales son utilizados durante el proceso de generación de las reglas de asociación temporal.

Todos sus componentes son definidos como estáticos, por lo tanto, son accedidos directamente utilizando la *notación punto* utilizando el nombre de la clase para referenciarlos. En otras palabras, son *variables y métodos de clase*.
Por esta razón, no es necesario crear objetos de esta clase ya que la información reside en la clase misma.

En general, sus métodos están relacionados directamente con el proceso de búsqueda de los itemsets frecuentes y generación de las reglas de asociación temporal.

Los métodos más significativos de esta clase utilizados para setear y retornar los valores recuperados del archivo de propiedades *trabajoDeGrado.props* cuando se invoca el constructor de la clase TPreprocessPanel, a saber:

- fileSeparator
- pathSeparator
- pathSeparator
- lineSeparator
- userAccountName
- userHomeDirectory
- userCurrentWorkingDirectory
- userCurrentDataDirectory
- processFiles almacena la cadena de caracteres (*string*) que define las variables de proceso, tal cual es leída del archivo de configuración, a ser utilizada en el proceso de generación múltiple;
- resultFiles almacena la cadena de caracteres (*string*) que define los destinos de los archivos que contienen los resultados de proceso, a ser utilizada en el proceso de generación múltiple;
- getProcessFilesList almacena la cadena de caracteres (*string*) que define las variables de proceso de cada corrida múltiple a realizar como una entrada en un vector;
- getResultFilesList contiene los diferentes destinos de los archivos que contienen los resultados de cada proceso múltiple como una entrada en un vector;
- getNextProcessFiles recupera el siguiente conjunto de variables de proceso múltiple;
- getNextResultFiles recupera el siguiente path destino del archivo que contiene los resultados del proceso múltiple ejecutado;
- getProcessFilesValues recibe la cadena de caracteres con las variables de proceso, separadas por "|", y setea cada variable de proceso para ser identificada individualmente durante el proceso múltiple;
- getResultFilesValues setea el path destino del archivo resultado generado en el proceso múltiple;
- existsNextProcessInfo verifica la existencia de datos para el proceso múltiple;
- cantMomentosPorDia
- processType que define el tipo de proceso a realizar; y
- frecuentItemsetCounter que indica el máximo identificador de archivos temporales a importar cuando el valor de processType es igual a 2 o 3.

Los métodos más significativos de esta clase utilizados para setear y devolver el umbral para el soporte, el soporte temporal y la mínima confianza, definidos por el usuario, a ser utilizados en el algoritmo de búsqueda de los itemsets frecuentes y la generación de las reglas de asociación temporal, a saber:

- minSupport que indica el mínimo soporte de un itemset;
- minTemporalSupport que indica el mínimo soporte temporal de un itemset;
- minMetric que indica la mínima confianza de una regla;
- numRules que establece el mínimo número de reglas que deben ser generadas para ser tenidas en cuenta;
- readCounter que indica la cantidad máxima de transacciones (instancias) que son leídas cada vez que se accede a la base de datos;
- deltaType que determina el desplazamiento mínimo que tendrán los intervalos de un *histograma*;
- removeMissingCols que indica si se deben eliminar las columnas (atributos de las transacciones) erróneas;

- calculatePremiseSupport que indica si se debe calcular el soporte de la premisa en el intervalo de validez de la regla o si se asume una distribución constante (uniforme) en toda la base de datos y se toma el valor que ya tiene calculado el itemset premisa en el intervalo de frecuencia.

Los métodos más significativos de esta clase utilizados para generar y recuperar los archivos auxiliares generados durante la ejecución del proceso de búsqueda, a saber:
- verifyTargetPath que verifica la existencia del path origen/destino de los archivos auxiliares de proceso;
- getTargetPath que construye la cadena que representa el path destino de los archivos auxiliares de proceso;
- getHashtable que recupera la *tabla de hash* del path destino de los archivos auxiliares con los itemsets generados durante el proceso, de acuerdo al nombre y tamaño recibidos como argumento;
- saveHashtable que guarda la *tabla de hash* en el path destino como un archivo auxiliar con los itemsets generados durante el proceso, de acuerdo al nombre y tamaño recibidos como argumento;
- getHistogram que recupera el *histograma* del path destino de los archivos auxiliares, de acuerdo al nombre y tamaño recibidos como argumento;
- saveHistogram que guarda el *histograma* en el path destino como un archivo auxiliar, de acuerdo al nombre y tamaño recibidos como argumento;
- appendElements que retorna una nueva *tabla de hash* conteniendo los itemsets de las dos *tablas de hash* recibidas como argumento;
- addElement que agrega un itemset en la tabla de hash recuperada del path destino de los archivos auxiliares;
- addElements que concatena los itemsets de la *tabla de hash* que recibe como argumento en la *tabla de hash* que recupera del path destino de los archivos auxiliares;
- itemSetPositionValues que retorna la posición del itemset en el vector o *tabla de hash*, según corresponda, recibidos como argumento;
- removeTemporalContainers que elimina del path destino todos los archivos temporales generados.

Y los métodos más significativos de propósito general, a saber:
- el método modulo que calcula y retorna el módulo entre estos valores enteros que recibe como argumento, teniendo en cuenta la *diferencia temporal* (mínima diferencia encontrada entre dos transacciones consecutivas en el tiempo, de la base de datos) de las transacciones;
- el método mod que calcula y retorna el resto de la división de dos valores enteros que recibe como argumento;
- el método div que retorna la parte entera del cociente entre dos valores enteros que recibe como argumento.

TInterval

Definida en el paquete *weka.associations*. Esta clase es una de las más importantes ya que permite especificar el límite inferior y superior, como valores enteros, de un intervalo de tiempo. Esta información es de vital importancia para representar el carácter temporal de los objetos de la aplicación.

Algunos de los métodos más importantes son:
- los constructores de la clase;
- el método modulo que permite calcular el módulo, de acuerdo a la diferencia temporal especificada por el usuario, del intervalo;
- los métodos para recuperar y setear los límites inferior y superior; y
- el método intersection que permite calcular la intersección de dos intervalos, es decir la porción que ambos intervalos tienen en común.

Histogram

Esta clase representa el histograma definido en [2] [5] y ha sido definida en el paquete *weka.associations*.

Como se describe anteriormente, teniendo en cuenta el período de vida (*lifespan*) asociamos con cada itemset un histograma el cual se va construyendo durante el proceso de búsqueda de los itemsets frecuentes.
En un sentido más general, tenemos que los histogramas describen el comportamiento temporal de los itemsets. Por esta razón, podemos considerar a cada histograma como una "serie de tiempos" en los que el itemset está presente, los cuales son patrones utilizados para minar.

En el histograma acumulamos el número (cantidad) de transacciones, pertenecientes al período de vida del itemset, en las cuales el itemset está presente.
El ancho de cada uno de los intervalos del histograma, llamado *time unit*, es definido por el usuario y es el mismo para todos los histogramas del proceso.
Por definición, tenemos que cada histograma está formado por una secuencia de subintervalos cuya amplitud está determinada por el valor Δ_t, definido por el usuario.
De esta manera, si tomamos un itemset X, el cual está definido en el intervalo del tiempo $[t_1, t_2]$, tenemos el histograma definido como sigue:

$$\{(v_1,f_1), (v_2,f_2),..,(v_p,f_p)\}$$

donde

$$v_1 = t1,$$
$$v_2 = v_1 + \Delta_t,$$
$$\ldots,$$
$$vp + \Delta_t = t_2$$

y fi es el número de transacciones, pertenecientes al período comprendido entre v_i y $v_i + \Delta_t$, que contienen al itemset X.
El valor Δ_t, definido por el usuario, siempre será igual o mayor que la diferencia mínima que existe entre dos transacciones consecutivas de la base de datos.

De lo anterior tenemos que una transacción con un identificador temporal asociado podrá estar incluida en uno y sólo uno de los subintervalos $[v_i, v_i + \Delta_t]$ que forman el histograma de un itemset particular.

Internamente, vemos al histograma como una colección de objetos de la clase HistogramElement, los cuales representan los pares ordenados (v_i, f_i) y contienen el intervalo sobre el cual están definidos, y el contador de transacciones que contienen al itemset en dicho intervalo.

Luego, a medida que vamos recorriendo la base de datos de transacciones en busca de los itemsets frecuentes, tomamos cada transacción y verificamos que el *timestamp* de la transacción está incluido en el período de vida (*lifespan*) del itemset.
Si esto ocurre, verificamos que la transacción que estamos analizando contenga al itemset en cuestión; y, en caso afirmativo, tomamos el subintervalo del histograma que incluye el identificador temporal de la transacción que estamos analizando, e incrementamos el contador de transacciones en una unidad.

Entre lo métodos más importantes de esta clase podemos citar:
- los métodos que permiten setear y retornar el límite superior e inferior del intervalo del histograma;
- el método elementsCount que retorna la cantidad de transacciones que contienen al itemset en todo el intervalo de frecuencia y en un intervalo que recibe como argumento;
- los métodos para insertar, eliminar y retornar un HistogramElement en el histograma respetando el ordenamiento determinado por cada subintervalo; y
- los métodos para procesar una transacción particular.

HistogramElement
Esta clase ha sido definida en el paquete *weka.associations* y representa el subintervalo $[v_i, v_i + \Delta_t]$ del histograma utilizado para contener el contador de transacciones, tal que su identificador temporal está definido en $[v_i, v_i + \Delta_t]$, que contienen el itemset.

Los métodos más importantes de esta clase son:
- los constructores que definen el subintervalo del histograma sobre el cual el elemento tendrá validez e inicializan el contador de transacciones en cero;
- los métodos para setear y recuperar los límites inferior y superior del intervalo.
- el método add que actualiza el contador de las transacciones contenidas en esta porción del histograma si el identificador temporal de la transacción, que recibe como argumento, está incluido en el intervalo definido para el elemento del histograma;
- el método elementsCount que retona la cantidad total de transacciones que contienen al itemset en el intervalo de verificación ó la cantidad de transacciones contenidas en el intervalo que recibe como argumento.

En este caso, no es necesario que los objetos de esta clase almacenen los datos de las instancias (transacciones) que contiene ya que no se hace referencia al contenido de la misma una vez procesada.

Como vimos, la información contenida en el histograma, en particular, en los elementos del histograma, es utilizada para calcular el soporte del itemset principalmente cuando el mismo no sea frecuente en todo el período de vida completo y deba aplicarse el algoritmo a_posteriori definido en [5], ya que en este caso se va ajustando el intervalo de frecuencia del itemset con su período de vida, tratando de obtener aquellos subintervalos maximales en los que el itemset es frecuente, posibilitando recalcular el soporte del itemset en cada subintervalo maximal para verificar su frecuencia.

TemporalAssociationRule
Definida en el paquete *weka.associations*. Los objetos de esta clase se utilizan para calcular las reglas de asociación temporal durante el proceso de generación.

Por definición, tenemos que cada uno de los objetos de esta clase tiene definido el intervalo de tiempo sobre el cual la regla de asociación temporal es válida.
Además, cada regla de asociación es generada tomando como base un itemset frecuente sobre el cual se aplica la regla; y a partir de él, tenemos la premisa y la consecuencia de la regla de asociación.

Los métodos más importantes de la clase son:
- el constructor de la clase que permite inicializar el itemset frecuente origen, la premisa y la consecuencia de la regla y el intervalo de validez de la regla de asociación temporal;
- el método confidenceForRule el cual recibe el umbral para la mínima confianza definido por el usuario y el histograma general (*histograma R*) de la base de datos, y que se utiliza para calcular la confianza de la regla. El cálculo de la confianza de la regla de asociación temporal se lleva a cabo dependiendo del parámetro *calculatePremiseSupport* (especificado por el usuario).
  Si el valor es verdadero, entonces antes de calcular la confianza de la regla, se debe calcular el soporte de la premisa en el intervalo de validez de la regla (invocando el método confidenceForRuleCalculateSupportPremise).
  Si el valor es falso, entonces se asume que todos los itemsets tienen una *distribución temporal uniforme* en la base de datos, entonces la posibilidad de que aparezca en algún subconjunto del *lifespan* del itemset es la misma, en particular, en el subconjunto en el cual está definida la regla de asociación temporal (invocando el método confidenceForRuleSameSupportPremise);
- el método setConfidence que actualiza la colección de confianzas de la regla de asociación con los valores obtenidos del cálculo de la misma;
- el método getSupportValue que retorna el valor del soporte que la premisa tiene calculado (no lo calcula nuevamente);
- el método calculateSupportValue que calcula el soporte de la premisa en el intervalo de validez de la regla;
- el método getConfidence que recibe dos valores decimales y calcula la confianza; y
- el método pruneRules que elimina todas las reglas de asociación temporal generadas cuya confianza está por debajo del umbral definido por el usuario.

TItemSetContainer
Esta clase ha sido definida en el paquete *weka.core* y su función principal es la de ser utilizada como contenedor de itemsets en la *tabla de hash*.

Cada objeto de esta clase, contiene una cadena de valores, separados por coma, que representa la *clave* de identificación del contenedor, y una *colección de itemsets* que se corresponden con la clave del contenedor al que pertenecen.

Entre los métodos más importantes de la clase tenemos:
- el método addElement que permite agregar un itemset al contenedor;
- los métodos para insertar, recuperar y eliminar el itemset ubicado en una posición específica del contenedor;

- el método includes que se utiliza para verificar si el itemset pasado como argumento está incluido en el contenedor, es decir, verifica si el itemset está en la colección de itemsets del contenedor;
- el método key que retorna la clave (cadena de valores) del contenedor; y
- el método keyLength que calcula y retorna la cantidad de elementos de la cadena clave, o sea, la cantidad de valores separados por coma.

THashtable

Definida en el paquete *weka.core*. Esta clase implementa una *tabla de hash*, que mapea claves a valores.
Cualquier objeto no nulo puede ser utilizado como una clave o un valor.
Por definición, las claves en la *tabla de hash* deben ser unívocas para permitir la identificación individual de los valores asociados a la clave.

En nuestro caso particular, la *tabla de hash* puede ser vista como una colección de pares *clave / valor* (objetos de la clase Entry) donde las claves son *strings* y los valores son objetos de la clase TItemSetContainer.

Cada vez que se agrega un itemset en la *tabla de hash*, verificamos si su clave (hashCode) está contenida entre las claves de la tabla de hash. Si lo anterior es correcto, se incluye directamente el itemset en el contenedor (objeto de la clase TItemSetContainer) que mapea con la clave del itemset.
Por el contrario, si no existe una clave en la *tabla de hash* que mapee con la clave del itemset que estamos intentando agregar, entonces se genera una nueva entrada (*entry*) en la *tabla de hash*, con clave igual a la del itemset en cuestión y un contenedor vacío, y se incluye el itemset en dicho contenedor.

Para almacenar y retornar objetos exitosamente de una *tabla de hash*, los objetos utilizados como claves, *strings* en nuestro caso particular, deben implementar los métodos hashCode y equals.
De lo anterior, tenemos que existe un *polimorfismo* entre los objetos utilizados como claves ya que, sin importar la clase a la cual ellos pertenecen, sólo basta con que puedan ser identificados por su *código de hash* y comparados por sus valores internos.

Todos los métodos que necesitan comparar las claves (*strings*) o los valores (objetos de la clase TItemSetContainer) asociados a las claves, utilizan el método hashCode (*hey*) de los objetos que están siendo comparados.

Los métodos más importantes de la clase son:
- los constructores de la clase, los cuales definen y setean, entre otras cosas, la capacidad inicial y el factor de carga de la *tabla de hash*;
- el método elementsList que retorna una colección conteniendo todos los itemsets almacenados en los contenedores de la *tabla de hash*;
- el método contains que retorna verdadero si el objeto que recibe como argumento (itemset) está contenido en la *tabla de hash*. Retorna falso en caso contrario;
- los métodos containsKey y containsValue que verifican si el objeto recibido como argumento está contenido en el conjunto de claves o valores, respectivamente;

- el método get que retorna los valores de la *tabla de hash* que mapean con la clave recibida como argumento. En nuestro caso particular, este método retorna la colección (*contenedor*) de itemsets especificados en la entrada que mapea con la clave del itemset que recibe como argumento;
- el método put que agrega un objeto (itemset) en la *tabla de hash* respetando el siguiente procedimiento: si existe alguna entrada (entry) en la *tabla de hash* que coincide con la clave (*hashcode*) del objeto (itemset) que recibe como argumento para agregar en la *tabla de hash*, entonces agrega el itemset en el contenedor de dicha entrada; sino genera una nueva entrada en la *tabla de hash* con clave igual al *hashcode* del itemset y contenedor vacío, y agrega el itemset en dicho contenedor.

Además, para realizar el volcado de la *tabla de hash* a filesystem tenemos:
- el método writeObject el cual guarda el estado de la *tabla de hash* como un *stream* de datos que posteriormente es guardado en filesystem, y
- el método readObject que reconstruye la *tabla de hash* a partir de un *stream* de datos recuperado del filesystem.

Ambos métodos actúan de manera sincronizada para realizar su función.

TDIC

Esta clase ha sido definida en el paquete *weka.associations* y es la clase principal utilizada para implementar el algoritmo TDIC (*Temporal Dynamic Itemset Counting*) definido en [2].

Como todos los algoritmos de asociación temporal, esta clase extiende la clase TAssociator y por lo tanto implementa el método buildAssociations para ejecutar el algoritmo de asociación definido sobre el conjunto de instancias recibido como argumento.

Cuando el algoritmo comienza su ejecución, a través de la invocación del método buildAssociations, se inicializa el estado interno del objeto. En este momento se setean todas las variables de proceso recuperadas del archivo de configuración y almacenadas en la clase ClsGeneral.

Luego, dependiendo del tipo de proceso especificado por el usuario en el archivo de configuración, tenemos los siguientes escenarios de ejecución del algoritmo:

a) Si el tipo de proceso es 1, entonces se ejecuta el proceso completo. En este caso, primero se encuentran todos los itemsets frecuentes y luego se generan todas las reglas de asociación temporal a partir de los itemsets frecuentes encontrados en el paso anterior.
b) Si el tipo de proceso es 2, no se lleva a cabo ningún proceso sobre los datos. En este caso, sólo se recuperan los itemsets frecuentes encontrados en alguna ejecución anterior, leyéndolos del filesystem, y se finaliza el proceso. Este tipo de proceso resulta útil cuando solamente se intenta visualizar en pantalla, y generar el archivo de salida con, los itemsets frecuentes encontrados durante la ejecución del proceso normal.
c) Si el tipo de proceso es 3, entonces se recuperan los itemsets frecuentes encontrados en alguna ejecución anterior, leyéndolos del filesystem, y se generan las reglas de asociación temporal a partir de ellos. Este tipo de proceso resulta útil cuando se

quieren volver a generar las reglas de asociación temporal, utilizando un conjunto de itemsets frecuentes ya generado.

Como se dijo anteriormente, el algoritmo TDIC se centra en base al algoritmo DIC (*Dynamic Itemset Counting*) propuesto y desarrollado en [3], que reduce el número de pasadas realizadas sobre la base de datos ya que, a partir de leer *M* transacciones consecutivas y de calcular su soporte, va incorporando los nuevos itemsets candidatos en el mismo momento en que son descubiertos, sin esperar a leer todas las transacciones de la base de datos y completar así una pasada.
De esta manera, tenemos un mayor aprovechamiento del tiempo invertido en recorrer la base de datos de transacciones y una reducción del coste necesario para procesar las mismas.

Utilizando las ventajas antes descriptas, el potencial del algoritmo TDIC radica en la utilización del tiempo como herramienta para identificar las transacciones que realmente son de interés a la hora de evaluar el soporte y soporte temporal de cada uno de los itemsets que están siendo analizados para determinar si son frecuentes, o no, en el intervalo de frecuencia.

Volviendo a la ejecución del método buildAssociations; y asumiendo un tipo de ejecución normal (tipo de proceso 1), el algoritmo después de inicializar el estado interno del objeto utilizado para llevar a cabo el proceso (invocando al método initialize), almacena la hora de inicio e invoca al método findLargeItemSets para encontrar los itemsets frecuentes e iniciar la fase 1.

Inicialmente, se generan todos los itemsets frecuentes de tamaño 1 a través de la invocación del método singletons.
Luego, invocando al método deleteItemSets, se eliminan los itemsets candidatos encontrados en el paso anterior que no cumplen con las restricciones de soporte o de soporte temporal, según el criterio del usuario, generando de esta manera los itemsets frecuentes de tamaño 1.

A continuación, se generan los itemsets candidatos de tamaño 2, utilizando los 1-itemsets frecuentes encontrados recientemente, invocando al método apriori_gen.
Los 2-itemsets candidatos devueltos por este método son asignados a TRAIN para ser procesados a continuación.

A partir de este momento comienza la búsqueda recursiva de los $(n+1)$ itemsets frecuentes, donde n es mayor a 1.
Para ello, por cada pasada (lectura de todas las transacciones) que se realiza sobre la base de datos, se leen *M* transacciones de la base de datos, utilizando el método readInstances.
Con las *M* transacciones leídas recientemente, se actualizan los contadores de soporte, soporte temporal y el histograma de los itemsets candidatos que están contenidos en las transacciones leídas, invocando al método updateCounters.

Luego, por cada uno de los valores de k (tamaño de los itemsets candidatos encontrados hasta el momento), invocando al método deleteItemSets, se determinan los itemsets frecuentes encontrados en la pasada actual.

A continuación, se generan los "*nuevos*" itemsets candidatos a partir de los itemsets frecuentes encontrados en la pasada actual, invocando al método apriori_gen, pasándole como parámetro la colección de itemsets frecuentes encontrados en las pasadas anteriores y los recientemente encontrados (los itemsets frecuentes de ambas colecciones tienen igual tamaño), y el tamaño de los itemsets candidatos (igual al tamaño de los itemsets frecuentes pertenecientes a las colecciones incrementado en 1) a generar utilizando las colecciones antes mencionadas.
Los nuevos itemsets candidatos generados, son agregados en TRAIN para ser procesados en las pasadas sucesivas.

Una vez generados todos los nuevos itemsets candidatos utilizando los itemsets frecuentes recientemente encontrados, se procede a eliminar de TRAIN los itemsets candidatos que han completado una pasada, es decir, que han recorrido todas las transacciones de la base de datos tratando de encontrar los intervalos en los cuales son frecuentes.
Para ello, se utiliza el método deleteCompleteItemSets el cual toma cada uno de los itemsets candidatos de TRAIN y verifica que haya recorrido todas las transacciones de la base de datos. Si lo anterior es correcto, entonces invoca el método a_posteriori para determinar los intervalos maximales, pertenecientes a su *lifespan*, en los que el itemset es frecuente.
En caso de encontrar uno o más subintervalos maximales en los que el itemset es frecuente, se genera una copia del itemset por cada subintervalo maximal como intervalo de frecuencia.
Con los itemsets frecuentes que tienen subintervalos maximales, se invoca nuevamente el método apriori_gen para generar los itemsets candidatos, los cuales son agregados en TRAIN para ser incluidos en el proceso.

La ejecución del método findLargeItemSets finaliza cuando la colección TRAIN no tiene elementos (itemsets candidatos) para procesar.

A continuación, se invoca al método findRulesQuickly para generar las reglas de asociación temporal, dando comienzo a la fase 2 del algoritmo.
Para ello, se recuperan las colecciones de itemsets frecuentes encontrados en la fase 1, y por cada uno de ellos, se generan sus reglas de asociación temporal.
La colección de reglas de asociación generada a partir de cada itemset frecuente es almacena en la colección resultado devuelta por el método.

Otros métodos importantes de esta clase son:
- el método resetOptions, a través del cual los objetos de la clase pueden setear el estado interno de las variables de proceso, definidas en el archivo de configuración *trabajoDeGrado.props* y almacenadas en la clase ClsGeneral, antes de comenzar la ejecución del mismo;
- el método add el cual crea un itemset con un único item (singleton) recuperando el i-ésimo atributo de la transacción que recibe como argumento; y lo agrega en la colección resultado, respetando el orden lexicográfico;
- el método elementPosition que recupera la posición actual del itemset de la colección, cuya clave coincide con la cadena de caracteres recibida como argumento;
- el método join que genera un itemset candidato de tamaño k+1, a partir de dos itemsets de tamaño k que recibe como argumento, uniendo los items de ambos

itemsets hasta la posicion k-1, siempre que los mismos coincidan y que los items de la posición k sean diferentes entre si;
- el método joinItemSets, invocado por el método apriori_gen para representar el *paso de mezcla*, que combina todos los itemsets de tamaño k-1 de la colección recibida como argumento, para crear los itemsets de tamaño k y actualizar sus contadores;
- el método pruneItemSets, invocado por el método apriori_gen para representar el *paso de poda*, que elimina del conjunto de itemsets de tamaño k todos los itemsets tal que cualquier subconjunto propio de tamaño k-1 no está definido en el conjunto de (k-1) itemsets recibido como argumento;
- el método calculateDeltaT, el cual se utiliza para calcular el valor de desplazamiento dentro del histograma (determina la amplitud de cada uno de los intervalos del histograma).

Como dijimos, esta clase es el componente principal del prototipo desarrollado ya que implementa el algoritmo propuesto, utilizando la información temporal que las transacciones tienen dentro de la base de datos analizada.
Esta idea permite identificar a los itemsets en el tiempo, calcular su soporte con las transacciones que son de interés y desechar aquellas que no lo son.

Diagrama del modelo de datos

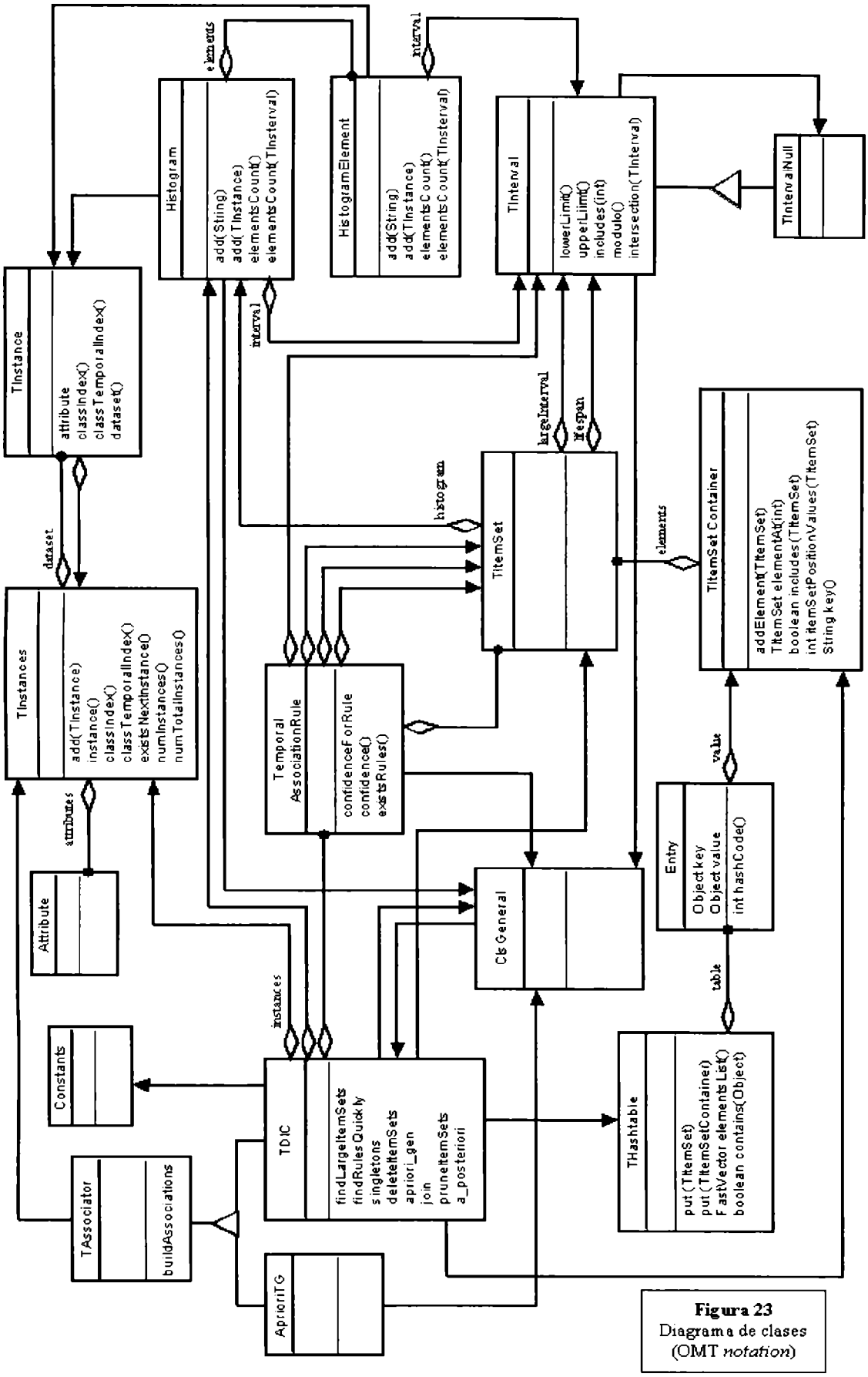

Figura 23
Diagrama de clases
(OMT *notation*)

Archivo de Configuración

Características generales

En la versión original de *Weka*, la única manera de asignar valores a las variables de proceso de los algoritmos de asociación es en el momento en que el algoritmo ha sido seleccionado.
En ese momento, y dependiendo del algoritmo a procesar, se pueden especificar los valores de las variables a través de la *interface genérica para editar las propiedades*, la cual se personaliza de acuerdo a las variables que cada algoritmo permite especificar.

Teniendo en cuenta que en el 90% de las pruebas realizadas eran tendientes a comparar dos algoritmos de asociación, se volvía muy tedioso tener que especificar, cada vez que se ejecutaba un algoritmo, el mismo valor para las variables de proceso.
De esta manera se decidió crear un archivo de configuración, al que llamamos *trabajoDeGrado.props*, el cuál se cargase por única vez al inicio de la aplicación *Weka*, en dónde pudiésemos asignar valor a las variables de proceso de interés de manera centralizada y acotada para evitar inconsistencias entre las ejecuciones realizadas.

Las variables del archivo de configuración definen el tipo de proceso a realizar sobre la base de datos seleccionada, y los directorios utilizadas para el almacenamiento de los archivos temporales de proceso.
En nuestro caso particular, además de las variables comunes a cualquier algoritmo de asociación, existe un conjunto de variables que han sido definidas y son utilizadas únicamente en la ejecución del algoritmo *TDIC* ya que están íntimamente relacionadas con la temporalidad de los datos a procesar y la forma en que los mismos son almacenados en las clases definidas internamente para tal fin.

Las variables de proceso más importantes del archivo de configuración son:

PROC_DATA.PROCESS_FILES
Define los múltiples procesos a realizar sobre la base de datos seleccionada.
Cada lista de variables de proceso está separada por ";". Por cada proceso, especifica además los valores de las variables de proceso que se aplicarán al algoritmo seleccionado. Los valores de las variables de proceso están separados por "|".

Los valores de las variables que se especifican por cada proceso son los siguientes:

minsup=*ms* (umbral para el mínimo soporte)
mintempsup=*mts* (umbral para el mínimo soporte temporal)
minmetric=*mm* (umbral para la mínima confianza)
minrules=*nr* (umbral para la generación de la reglas de asociación temporal)
rcounter=*rc* (cantidad de transacciones a leer por vez)
deltat=*dt* (amplitud de los subintervalos del histograma)

Los posibles valores que determinan la amplitud de los subintervalos del histograma (variable **deltat**) son:

TIME	DAY	WEEK	MONTH	YEAR
0	1	2	3	4

PROC_DATA.RESULT_FILES
Define los directorios destino de los archivos de salida para cada uno de los procesos múltiples definidos en la variable **PROC_DATA.PROCESS_FILES**.
Cada uno de estos directorios está separado por ";".

PROC_DATA.PROCCESS_TYPE
Define el tipo de proceso a realizar cuando se ejecuta el *algoritmo TDIC*.
Los posibles valores que se pueden especificar para esta variable son:
- Si el valor es 1, entonces la ejecución del proceso se realiza en dos fases como ha sido definido en [2][5], encontrando primero los itemsets frecuentes (*fase 1*), y luego, utilizando los itemsets frecuentes encontrados en la fase anterior, generando las reglas de asociación temporal (*fase 2*).
- Si el valor es 2, entonces no se realiza ningún proceso sobre la base de datos. Por el contrario, sólo se cargan las colecciones de itemsets frecuentes generados en la última corrida y se muestran en pantalla, generando el archivo de salida en el directorio especificado en la variable **PROC_DATA.RESULT_FILES**.
- Si el valor es 3, entonces se cargan las colecciones de itemsets frecuentes generados en la última corrida, y se generan las reglas de asociación temporal a partir de estos itemsets frecuentes, mostrando los datos de proceso obtenidos en la pantalla y generando el archivo de salida en el directorio especificado en la variable **PROC_DATA.RESULT_FILES**.

PROC_DATA.FI_COUNTER
Define el máximo índice de las colecciones de itemsets frecuentes generados anteriormente que serán importados.Esta variable sólo es tenida en cuenta cuando el valor de la variable de proceso **GRAL_DATA.PROCCESS_TYPE** es igual a 2 ó 3.
En estos casos, el *algoritmo TDIC* simula la primera fase del proceso de descubrimiento de las reglas de asociación temporal, importando directamente del *filesystem* las colecciones de itemsets frecuentes generados en algún otro proceso.

PROC_DATA.CALCULATE_PREMISE_SUPPORT
Esta variable es utilizada en el cálculo de las reglas de asociación temporal.
Cuando el valor de esta variable es *true*, el soporte de la premisa de la regla de asociación debe ser recalculado en el intervalo de validez de la regla de asociación a la cual pertenece.
Por el contrario, si el valor de la variable es *false*, entonces no se calcula nuevamente el soporte de la premisa, asumiendo una distribución uniforme de los items en las transacciones de la base de datos. Por lo tanto, se toma el valor de soporte que tiene calculado el itemset premisa actualmente en su intervalo de frecuencia.

PROC_DATA.REMOVE_MISSING_COLS
Indica si se deben eliminar o no las columnas (atributos / items) inválidas de la base de datos cuando la misma es pasada como parámetro al algoritmo de asociación seleccionado para el proceso.
En nuestro caso, esta variable no ha sido utilizada ya que los items de las transacciones de las bases de datos sintéticos han sido generadas aleatoriamente por *jBasket* y por tal motivo es imposible determinar la validez de los mismos puesto que no hay forma de controlarlo ya que no existe una cabecera donde se indique el tipo y lista de valores posibles para cada uno de los items.

GRAL_DATA.CANT_MOMENTOS_X_DIA
Define la cantidad de momentos por día definidos en las transacciones de la base de datos.
Está variable de proceso está íntimamente relacionada con la forma en la cual ha sido construida la base de datos de transacciones y la forma en la cual se calcula el identificador temporal de cada transacción.
Este valor es utilizado para calcular la amplitud de los intervalos en el histograma cuando se procesan los itemsets a medida que avanza la ejecución del algoritmo *TDIC*.

GRAL_DATA.DIFERENCIA_TEMPORAL
Define la diferencia temporal que existe entre dos transacciones consecutivas (en cuanto a su identificador temporal) de la base de datos.
Este valor es utilizado para calcular la amplitud de los subintervalos en el histograma.

Veamos más en detalle como se realiza el cálculo de la amplitud de cada subintervalo del histograma cuando utilizamos el *algoritmo TDIC*.
Tomando como referencia el valor de la variable **deltat** especificada en la variable de proceso **CANT_MOMENTOS_X_DIA**, la diferencia temporal se calcula como sigue:

- ✓ Si el valor de la variable **deltat** es igual a 0 (**TIME**), entonces la amplitud de los subintervalos en el histograma es igual a
  GRAL_DATA.DIFERENCIA_TEMPORAL.
- ✓ Si el valor de la variable **deltat** es igual a 1 (**DAY**), entonces la amplitud de los subintervalos en el histograma es igual a
  CANT_MOMENTOS_X_DIA * DIFERENCIA_TEMPORAL.
- ✓ Si el valor de la variable **deltat** es igual a 2 (**WEEK**), entonces la amplitud de los subintervalos en el histograma es igual a
  CANT_MOMENTOS_X_DIA * DIFERENCIA_TEMPORAL * 7.
- ✓ Si el valor de la variable **deltat** es igual a 3 (**MONTH**), entonces la amplitud de los subintervalos en el histograma es igual a
  CANT_MOMENTOS_X_DIA * DIFERENCIA_TEMPORAL * 30.
- ✓ Si el valor de la variable **deltat** es igual a 4 (**YEAR**), entonces la amplitud de los subintervalos en el histograma es igual a
  CANT_MOMENTOS_X_DIA * DIFERENCIA_TEMPORAL * 365.

Además, existen variables en el archivo de configuración que no están relacionadas directamente con la manera en la cual se lleva a cabo el descubrimiento de los itemsets frecuentes y su utilización en la generación de las reglas de asociación sobre la base de datos seleccionada, pero que son utilizadas en el almacenamiento en (y recuperación de) *filesystem* de los archivos temporales generados durante el proceso.

A continuación se detallan las variables involucradas en estas operaciones:

file.separator
Define el carácter utilizado para separar carpetas. Por defecto se utiliza el carácter "/".

path.separator
Define el carácter utilizado para separar las unidades (discos físicos o lógicos) de las carpetas destino de los archivos. Por defecto se utiliza el carácter ":".

user.home
Define la unidad de disco a partir de la cual se ejecuta la aplicación.

user.dir
Define el directorio de trabajo actual de la aplicación.

user.data
Define la carpeta donde se guardarán los archivos temporales de proceso generados durante la ejecución del algoritmo de asociación.
Como se mencionó anteriormente, el almacenamiento de los archivos temporales de proceso se realiza teniendo en cuenta el valor de las variables de proceso especificadas en el archivo de configuración.
Supongamos que tenemos los siguientes valores de las variables de proceso:

```
user.home=D
user.dir=Trabajo de Grado
user.data=datos
file.separator=/
path.separator=:
```

Entonces, el path destino de los archivos temporales de proceso se construye de la siguiente manera:

```
targetPath = user.home +
             path.separator +
             file.separator +
             user.dir +
             file.separator +
             user.data +
             file.separator
```

lo cual nos da como resultado el path ***D:/Trabajo de Grado/datos/***.

De la misma manera, podríamos haber especificado otros valores para las variables de proceso, sin alterar la forma en la cual se combinan para definir el path destino de los archivos temporales de proceso.

En cualquiera de los casos, debemos tener en cuenta los valores de las variables que utilizamos como separador (**path.separator** y **file.separator**) ya que estamos ejecutando *código Java* y corremos el riesgo que los path generados no sean correctamente interpretados por la *máquina virtual* provocando un acceso a disco erróneo o un error en tiempo de ejecución.

Ejecución múltiple de procesos

Como vimos, dependiendo del valor de las variables de proceso especificadas en e archivo de configuración `PROC_DATA.PROCCESS_TYPE` y `PROC_DATA.PROCESS_FILES`, podemos realizar múltiples ejecuciones del *algoritmo de asociación TDIC*, sobre la base de datos seleccionada.

De esta manera, cuando se visualiza la pantalla *Weka Knowledge Temporal Explorer*, se cargan todos los valores especificados en el archivo de configuración.
En el caso particular de la variable de proceso `PROC_DATA.PROCESS_FILES`, se realiza un proceso complementario para identificar los diferentes grupos de variables de proceso. Para ello, se busca repetidas veces el carácter ";", agregando una entrada en la colección de salida por cada grupo de valores encontrado.

Supongamos que tenemos la variable de proceso `PROC_DATA.PROCESS_FILES` con el siguiente valor:

```
minsup=0.25|mintempsup=4|minmetric=0.25|numrules=5|rcounter=100|deltat=1;minsup=0.1|min
tempsup=10|minmetric=0.2|numrules=20|rcounter=50|deltat=0;minsup=0.15|mintempsup=5|min
metric=0.1|numrules=1|rcounter=75|deltat=2;
```

Figura 24
Ejemplo de valores de la variable
`PROC_DATA.PROCESS_FILES`

Una vez completado el proceso de recuperación de los grupos de variables de proceso, tenemos la colección de salida con la información que se muestra en la **Figura 25**.

Posteriormente, cuando el usuario comienza el análisis de la base de datos seleccionada a través de la ejecución del *algoritmo de asociación TDIC*, la aplicación recupera la información contenida en la colección definida anteriormente; y por cada uno de los valores contenidos en dicha colección, recupera y setea los valores individuales de las variables de proceso, los cuales están separados por "|", y realiza el análisis de la base de datos seleccionada ejecutando el algoritmo de asociación con los valores de proceso especificados por el usuario para esa corrida particular.

```
colVP (0)=minsup=0.25|mintempsup=4|minmetric=0.25|numrules=5|rcounter=100|deltat=1
colVP (1)=minsup=0.1|mintempsup=10|minmetric=0.2|numrules=20|rcounter=50|deltat=0
colVP (2)=minsup=0.15|mintempsup=5|minmetric=0.1|numrules=1|rcounter=75|deltat=2
```

Figura 25
Discretización de los valores de la variable
`PROC_DATA.PROCESS_FILES`

Lo mismo ocurre con la variable de proceso `PROC_DATA.RESULT_FILES`.
En este caso, se realiza un proceso complementario para identificar los diferentes paths destino de los archivos que se obtienen como resultado de la ejecución del algoritmo seleccionado.
Para ello, se busca repetidas veces el carácter ";" agregando una entrada en la colección de salida por cada path destino encontrado.

Supongamos que tenemos la variable de proceso `PROC_DATA.RESULT_FILES` con el siguiente valor:

```
D:/Tmp/1/;D:/Tmp/2/;C:/Temp/;
```

Una vez completado el proceso de recuperación, tenemos la colección de salida con la siguiente información:

```
colPD (0)=D:/Tmp/1/
colPD (1)=D:/Tmp/2/
colPD (2)=C:/Temp/
```

Figura 26
Discretización de los valores de la variable
`PROC_DATA.RESULT_FILES`

Luego, cuando comienza la ejecución de algoritmo de asociación seleccionado, la aplicación recupera la información contenida en la colección definida anteriormente. Esta información es utilizada para guardar en *filesystem* el archivo resultado (con la misma información visualizada en la pantalla) cuando finaliza la ejecución del algoritmo de asociación seleccionado.

EXPERIMENTACIÓN

Para medir la performance del prototipo desarrollado con los algoritmos de asociación existentes, se llevaron a cabo un conjunto de pruebas, utilizando bases de datos sintéticos, tendientes a comparar la cantidad de pasadas realizadas sobre los datos, el tamaño y la cantidad de itemsets frecuentes encontrados durante el proceso, la cantidad y calidad de las reglas de asociación generadas, etc.

Con este objetivo, se han generado diferentes bases de datos sintéticos modificando en cada caso algún parámetro particular dejando fijos los restantes, de acuerdo al tipo de prueba a realizar, variando la cantidad de items involucrados, el tamaño total de la base de datos para encontrar el modelo de generación más óptimo a la hora de realizar las pruebas.

Generación de Datos Sintéticos

El descubrimiento del conocimiento en las bases de datos muy grandes se transformó en un tema importante de investigación en la comunidad de data mining.

A pesar de que excelentes ideas sobre mecanismos de representación del conocimiento y algoritmos para descubrir el conocimiento están constantemente emergiendo, los investigadores de data mining muy a menudo se enfrentan a un problema, quizás torpe pero muy real: ellos no cuentan con conjuntos de datos convenientes para probar sus algoritmos.
Una importante razón es que las organizaciones que tienen una gran cantidad de datos recogidos, usualmente no tienen la voluntad para compartirlos; debido a que ello puede hacer descubrir accidentalmente sus negocios secretos, en lugar de protegerlos, dejando a la organización al descubierto.

Como resultado, los conjuntos de datos disponibles son usualmente insuficientes para examinar totalmente los algoritmos que están siendo estudiados [8].

Como una alternativa a los conjuntos de datos reales, muchos investigadores reordenan los conjuntos de datos sintéticos que comparten características similares a uno real. Aún cuando difieren de los conjuntos de datos reales, los conjuntos de datos sintéticos pueden ayudar a evaluar varios aspectos de los algoritmos de data mining, en especial la escalabilidad.

La generación del dato transaccional sintético (también conocido como *market basket data*) fue introducida para estudiar el descubrimiento de reglas de asociación en [11]. En [11] se propuso el primer generador de datos sintéticos donde las transacciones imitan las transacciones procesadas en un supermercado.
El problema de este generador de datos sintéticos es que el tiempo no es considerado como un factor durante la generación de datos, y las transacciones generadas no tienen un identificador temporal asociado.

Este generador podría ser perfectamente utilizado para evaluar el *algoritmo Apriori* en su versión original, ya que las transacciones de la base de datos a procesar no contaban con características temporales asociadas.

Según lo definido en [1], la visión del mundo real que los autores tienen es que, las personas tienden a comprar conjuntos de items juntos y este comportamiento se produce repetidas veces para el mismo cliente, incluso para clientes distintos que, aparentemente, no tienen nada en común.
Cada uno de estos conjuntos de items es potencialmente un itemset frecuente maximal. Sin embargo, algunas personas pueden comprar sólo alguno de los items del conjunto. Por otro lado, tenemos que una transacción puede contener más de un itemset frecuente agrupando sus atributos (items) de manera conveniente.
El tamaño de las transacciones es generalmente el mismo y pocas transacciones tienen muchos items.

Para crear un conjunto de datos (transacciones), el *framework* para generar los datos sintéticos definido en [1], utiliza los siguientes parámetros:

| D |	Número de transacciones
| T |	Tamaño promedio de las transacciones
| I |	Tamaño promedio de los itemsets frecuentes potencialmente maximales
| L |	Número de itemsets frecuentes potencialmente maximales
N	Número de items (diferentes) utilizados en la generación

Figura 27
Parámetros utilizados por *jBasket*

En primer lugar se determina el tamaño de la siguiente transacción. Este tamaño es escogido utilizando una *distribución de Poisson* que toma μ igual a |T|.
Luego se asignan los items a la transacción.

A cada transacción le es asignado una serie de itemsets frecuentes potenciales.
Si el itemset frecuente que tomamos no encaja en la transacción, el itemset es metido en la transacción de todas formas en la mitad de los casos. Por el contrario, el itemset es movido a la siguiente transacción en el resto de los casos.
Los itemsets frecuentes son escogidos de un conjunto *W* de itemsets. El número de itemsets en *W* es seteado a |L|.

Los items de la primera transacción son elegidos al azar.

Para modelar el fenómeno de que los itemsets frecuentes tienen a menudo items en común, alguna fracción de items de los itemsets subsecuentes son elegidos desde los itemsets previamente generados.
Los autores utilizan una variable exponencialmente distribuida al azar con valor igual al *nivel de correlación* para decidir esta fracción de cada itemset. Los items restantes son escogidos al azar.

De lo anterior, podemos concluir que por cada itemset, la mitad de los items son generados al azar, y la otra mitad son escogidos de los itemsets anteriores.
Esta forma de construir los itemsets captura el fenómeno de que los potenciales itemsets frecuentes, a menudo, tienen items en común.

Cada itemset en *W* tiene un *peso* asociado, el cual corresponde a la probabilidad de que el itemset sea escogido.

Los itemsets frecuentes potenciales son usados para generar las *D* transacciones.
Cada transacción es la unión de un conjunto de itemsets potencialmente frecuentes que son elegidos uno a uno de acuerdo a su peso.
Si un itemset no encaja en la transacción exactamente (al menos un item que forma el itemset no está incluido en la transacción) la parte restante del itemset (los items que forma el itemset que no están incluidos en la transacción) es movida a la siguiente transacción.

Para capturar el fenómeno de que los items en un itemset potencialmente frecuente no son siempre comprados juntos por los consumidores, un *nivel de ruido* es seteado desde una *distribución Normal* con significado = 0.5 y varianza = 0.1.
Mientras se agregan itemsets potencialmente frecuentes a una transacción, se guarda un número real, uniformemente distribuido entre 0 y 1, generado al azar y se borra un item del itemset con tal que el número escogido al azar sea menor que el *nivel de ruido*.

Para la generación de los datos sintéticos, se ha utilizado el *framework* presentado en [8] en donde cada una de las transacciones generadas está identificada en el tiempo.
Aquí, los autores presentan un generador de datos sintéticos que construye un conjunto de datos de transacciones identificadas en el tiempo, con patrones temporales embebidos, controlados por un conjunto de parámetros de entrada.
El *framework* presentado, denominado *Calendar Schema*, está determinado por una jerarquía de granularidades de tiempo de entrada, y es usado como un conjunto de posibles patrones temporales.

El generador de datos sintéticos fue pensado para proveer soporte a la comunidad de data mining en la evaluación de varios aspectos (especialmente los aspectos temporales y la escalabilidad) de los algoritmos de data mining.

Un *calendar schema* es un esquema relacional $\mathbf{R = (f_n: D_n, f_{n-1}: D_{n-1},..., f_1: D_1)}$ junto con una *restricción de validez*.
Cada atributo $\mathbf{f_i}$ es un nombre de granularidad de tiempo (por ejemplo: *year*, *month*, *week*, *day*, etc).
Cada dominio $\mathbf{D_i}$ es un subconjunto finito de números enteros positivos.
La *restricción de validez* es una función booleana sobre $\mathbf{D_n \times D_{n-1} \times ... \times D_1}$ en donde se especifica cuales de las combinaciones de valores en $\mathbf{D_n \times ... \times D_1}$ son válidas.

Una *restricción de validez* sirve a dos propósitos. El primero es excluir las combinaciones que no corresponden a algún intervalo de tiempo debido a la interacción de las granularidades del calendario. El segundo propósito es excluir los intervalos de tiempo que no son de interés.

Dado un *calendar schema* $R = (f_n: D_n, f_{n-1}: D_{n-1},..., f_1: D_1)$, un *calendar-based pattern* (o *calendar pattern*) es una tupla sobre R de la forma $\mathbf{<d_n, d_{n-1}, ..., d_1>}$ donde cada $\mathbf{d_i}$ está en $\mathbf{D_i}$ o es el símbolo asterisco (*).

El *calendar pattern* $<d_n, d_{n-1}, \ldots, d_1>$ representa el conjunto de intervalos de tiempo que son intuitivamente descriptos por "el d^{th}_1 f_1 del d^{th}_2 f_2, ..., del d^{th}_n f_n".

Como ejemplo, supongamos que tenemos el siguiente *calendar schema*: *(year, month, day)*. Con este esquema, podemos elegir un *calendar pattern* como "*todos los días de Enero de 1999*" o "*todos los 16th días de Enero de todos los años*"
Cada *calendar pattern* en efecto representa los intervalos de tiempo dados por el conjunto de tuplas en D_n x ... x D_1 que son válidas.

Para modelar el fenómeno que algunos itemsets pueden estar asociados con un patrón temporal pero otros no, se requiere un subconjunto (parte común) de los itemsets que son elegidos de un conjunto común único de itemsets y que la otra parte (parte independiente) de los itemsets son generados independientemente.
Los itemsets pertenecientes a la parte común son llamados *itemsets patrones*. Los *itemsets patrones* son compartidos por los itemsets por intervalo entre algunos intervalos de tiempo tomados como base.

Se aplica el mismo método propuesto en [11] para producir los itemsets frecuentes potenciales, para generar tanto los itemsets patrones, como la parte independiente de los itemsets por intervalo.
Se usa un parámetro *pattern-ratio* P_r, para determinar el porcentaje de los itemsets por intervalo por cada intervalo de tiempo base que se está eligiendo de los itemsets patrones.

Al igual que el generador de datos sintéticos propuesto en [11], este generador utiliza los siguientes parámetros para la generación de los datos sintéticos identificados en el tiempo:

- ✓ D: número promedio de transacciones por intervalo de tiempo básico.
- ✓ T: tamaño promedio de las transacciones.
- ✓ I : tamaño promedio de los itemsets frecuentes potencialmente maximales.
- ✓ L: número de itemsets por intervalo.
- ✓ N: número de items

Además de la lista descripta anteriormente, el *framework* presentado utiliza el parámetro P_r relacionado con la característica temporal. Este parámetro es utilizado para determinar el porcentaje de los itemsets por intervalo de tiempo base que son elegidos de los itemsets patrones.

El prototipo desarrollado para implementar el *framework* propuesto es *jBasket*.
El generador de datos *jBasket* toma como entrada un *calendar schema*, así como otros parámetros que controlan las características temporales y no temporales de los conjuntos de datos sintéticos, y devuelve un conjunto de transacciones sintéticas identificadas en el tiempo que tienen algún patrón temporal embebido definido por los parámetros de entrada.

Una vez iniciada su ejecución, *jBasket* asigna a cada transacción generada un timestamp del *calendar schema*, asegurando que el dataset entero posea ciertas características temporales en términos de *calendar patterns* que los usuarios esperan [8a].

El proyecto *jBasket* está implementado en Java 1.3 y consiste de 6 clases: TBasket, CalendarSchema, AprioriGenerator, PatternGenerator, Converter, Stat y un archivo de configuración llamado *ParametersBundle.properties* que define todos los parámetros utilizados en *jBasket*.

La clase TBasket es la clase principal del framework.

La clase CalendarSchema define el *esquema* a utilizar en la generación de la base de datos. Esta clase contiene la definición de las variables que determinan el esquema y el dominio de cada uno de los componentes que lo forman, utilizado en la generación de los archivos individuales de datos sintéticos, a saber:

- como un vector de valores enteros, la variable `D` define el tamaño del dominio de cada granularidad,
- como un vector de valores enteros, la variable `B` define el valor base para cada granularidad,
- como una matriz de valores enteros, la variable `pattern_out` definen los patrones inválidos para los cuales no se debe generar el archivo de datos transaccionales.

Cuando se ejecuta la clase TBasket, teniendo en cuenta los valores definidos para las variables de la clase CalendarSchema, junto con los valores especificados en el archivo de configuración *ParametersBundle.properties*, se generan los archivos de datos sintéticos conteniendo las transacciones por intervalo de tiempo.

Los archivos de datos resultantes son organizados por intervalo de tiempo base, esto es, todas las transacciones que tienen el mismo *timestamp* son agregadas dentro del mismo archivo, el cual es nombrado por el intervalo de tiempo base al cual hace referencia.
Por ejemplo, dado el *calendar schema*

<*year*: {1995, ..., 1999}, *month*: {1, .., 12}, *day*: {1, .., 31}>

el framework *jBasket* generará los siguientes archivos individuales de datos sdinteticos:

1995_1_1.dat,
1995_1_2.dat,
...,
1999_12_31.dat;

donde **1995_1_1.dat** contiene las transacciones pertenecientes al "1° de Enero de 1995". De la misma manera, **1999_12_31.dat** contiene las transacciones pertenecientes al "31° de Diciembre de 1999" y así sucesivamente.

Con un *calendar schema* como este, una vez finalizada la ejecución de *jBasket*, tendremos generados 1860 (1860=5 x 12 x 31) archivos individuales de datos sintéticos, cada uno de ellos conteniendo las transacciones generadas en dicho período de tiempo definido por el esquema.

Todos los archivos generados son depositados en el directorio indicado en el parámetro "`Dir`", definido en el archivo de configuración.
Este directorio, cuyo valor por defecto es *data*, es ubicado bajo el directorio actual en el cual todas las clases de *jBasket* residen.

Los archivos de datos están en formato de texto. La primer línea de cada archivo consiste de un número entero que indica el número de transacciones que contiene el archivo.
A partir de la segunda línea, cada línea representa una transacción. Cada transacción está formada por un conjunto de valores enteros (que representan los items de la transacción), los cuales están separados con un espacio en blanco uno de otro.
El primer valor entero de cada transacción indica el número de items que forman la transacción, y los demás valores enteros representan los items de la transacción, los cuales están organizados en orden ascendente.

A modo de ejemplo, supongamos que tenemos el archivo individual de datos sintéticos **1995_1_1.dat**, el cual está formado por las siguientes tres líneas:

```
2
3 1 15 17
2 45 56
```

Figura 28
Contenido de archivo individual de datos sintéticos **1995_1_1.dat** generado por *jBasket*

La primer línea indica que hay dos transacciones que "ocurrieron el 1 de Enero de 1995", la segunda línea representa la transacción 1 y la tercer línea muestra el contenido de la transacción 2.
Para la transacción 1 (segunda línea del archivo), el tamaño de la transacción es 3. Este valor está determinado por el primer valor entero de la línea en cuestión, comenzando desde la izquierda. Luego, vemos que el conjunto de items de la transacción está formado por 1, 15, y 17.
Algo similar ocurre con la transacción de la tercer línea del archivo. En este caso, el tamaño de la transacción es 2 y el conjunto de items que constituyen la transacción está formado por 45 y 46.

En nuestro caso particular, después de realizar varias pruebas con el generador de datos sintéticos *jBasket* tendientes a seleccionar el esquema que resulte más óptimo a la hora de realizar las pruebas de performance sobre el prototipo desarrollado, variando la granularidad del *calendar schema* y los parámetros del archivo de configuración utilizados en cada generación, hemos llegado a la conclusión de que el esquema más conveniente a utilizar es :

<*week*: {1, 2, ..., 9}, *day*: {1, 2, ..., 7}, *time*: {1, 2, 3}>

Ahora bien, si tenemos en cuenta la forma en la cual el framework genera los archivos individuales de datos sintéticos; sabemos que cada vez que se ejecuta *jBasket* utilizando este esquema calendario, se generan exactamente 189 (189 = 9*7*3) archivos individuales de datos sintéticos conteniendo transacciones generadas aleatoriamente siguiendo patrones de repetición e inclusión de los items que permiten que las transacciones generadas se asemejen a la realidad.

Los archivos generados por *jBasket* utilizando este esquema calendario, serán nombrados siguiendo las especificaciones definidas en el framework de la siguiente manera:

1_1_1.dat, **1_1_2.dat**, **1_1_3.dat**,
1_2_1.dat, **1_2_2.dat**, **1_2_3.dat**,
1_3_1.dat, **1_3_2.dat**, **1_3_3.dat**,
...,
9_6_1.dat, **9_6_2.dat**, **9_6_3.dat**,
9_7_1.dat, **9_7_2.dat**, **9_7_3.dat**.

Con este esquema, y teniendo en cuenta las limitaciones de memoria existentes a la hora de ejecutar el algoritmo de asociación *TDIC*, se generaron transacciones sintéticas para probar la performance de dicho algoritmo sobre un amplio rango de datos característicos, fijando el valor de las variables N (número de items diferentes utilizados en la generación) y L (número promedio de itemsets frecuentes por cada intervalo de tiempo) a 50 y 1.000, respectivamente.

Los valores asignados a las variables N y L son totalmente arbitrarios ya que su utilización en el proceso de generación no es relevante, y cualquier modificación puede variar el archivo de base de datos obtenido pero en una medida que no tiene una importancia significante.

Por otro lado, el valor asignado a D tiene que ver con el número promedio de transacciones por intervalo de tiempo; y por lo tanto, con la cantidad total de transacciones que queremos obtener al finalizar el proceso. Por esta razón, el valor que le otorguemos a esta variable es de vital importancia y debe ser asignado criteriosamente para que las transacciones generadas tengan un alto valor semántico a la hora de probar el prototipo desarrollado.

Por tal motivo, una vez analizadas las cuestiones que tienen ver estrictamente con las limitaciones de hardware encontradas a la hora de probar el prototipo desarrollado, se optó por trabajar con archivos de datos sintéticos que tuviesen alrededor de 1.000 transacciones (la cantidad total de transacciones puede variar de acuerdo a la forma en la cual el framework genera aleatoriamente los items de cada una de las transacciones y las agrupa en un intervalo de tiempo particular).
Así, el valor de la variable D se fijó a 6, ya que 1.000 / 189 = 5,29 y se prefirió tomar el siguiente valor entero para estar seguros de que cada archivo de datos sintéticos resultado (la suma de todos los archivos individuales) tuviese la cantidad de transacciones deseadas.

Con las consideraciones antes mencionadas, utilizando el framework *jBasket* se generaron conjuntos de datos respetando el esquema de pruebas desarrollado en [1], combinando los siguientes valores de las variables T e I del archivo de configuración:

- $|T|$: 5, 10, y 20
- $|I|$: 2, 4, 6

Unificador de archivos de datos sintéticos

Para probar la performance del prototipo desarrollado y su utilización en la aplicación *Weka*, necesitábamos generar un conjunto de datos sintéticos que, además de estar identificados en el tiempo, estuviesen contenidos en un único archivo que representase la base de datos de transacciones.
Para ello, se desarrolló una aplicación que toma los archivos individuales de datos sintéticos generados por el framework *jBasket* y retorna un único archivo conteniendo todas las transacciones generadas individualmente, identificadas unívocamente y ordenadas por su *timestamp* en orden descendiente, dando origen a la base de datos sintética.

Como vimos en la generación de datos sintéticos, cuando se ejecuta el framework *jBasket*, se genera un archivo individual de datos sintéticos por intervalo de tiempo válido comprendido en el esquema definido en la clase CalendarSchema, teniendo en cuenta la granularidad y el dominio de cada uno de los valores del esquema.
Estos archivos individuales de datos sintéticos son nombrados siguiendo una nomenclatura particular, de acuerdo al instante de tiempo al cual pertenecen todas las transacciones que se están generando actualmente.

Retomando el ejemplo del apartado anterior, en el archivo **1995_1_1.dat** se colocarán todas las transacciones generadas sintéticamente que corresponden al período definido como "1 de Enero de 1995", en el archivo **1995_1_2.dat** se colocarán todas las transacciones generadas sintéticamente que corresponden al período "2 de Enero de 1995" y así sucesivamente hasta completar todas las granularidades válidas del esquema calendario definido.

Luego, dependiendo del *calendar schema* utilizado en la generación de los datos sintéticos, tendremos que utilizar un convertidor de calendario particular, que tome el período "1 de Enero de 1995" y retorne un número entero largo que lo represente y que cumpla con ciertas restricciones.
Si tenemos en cuenta que para la generación de los períodos se utilizó un calendario gregoriano para identificar cada una de las fechas obtenidas en el esquema, vemos lo siguiente:

- "1 de Enero de 1995" es posterior a "31 de Diciembre de 1994", y
- "1 de Enero de 1995" es anterior a "2 de Enero de 1995" , y
- la diferencia que existe entre ambas comparaciones es de un día calendario.

Por lo tanto, si tenemos en cuenta estos datos, el convertidor deberá ser capaz de mantener esta correlación cronológica, asignándole a cada período (nombre del archivo) un número entero largo unívoco sin alterar la condición subyacente de que la diferencia entre un par de números consecutivos sea igual en todos los casos, independientemente de los valores (momentos) que se elijan para la comparación; y que debe mantenerse el valor de verdad de las comparaciones.

De lo anterior, si tomamos los siguientes nombres de archivos individuales consecutivos generados utilizando un esquema calendario (por ejemplo: **1994_12_31.dat**, **1995_1_1.dat**, **1995_1_2.dat**) deben cumplirse las siguientes reglas:

- el *timestamp* asociado al "31 de Diciembre de 1994" sea menor que el *timestamp* asociado al "1 de Enero de 1995", y

- el *timestamp* asociado al "1 de Enero de 1995" sea menor que el *timestamp* asociado al "2 de Enero de 1995".

Teniendo en cuenta las consideraciones mencionadas anteriormente, se ha desarrollado una aplicación que genera un único archivo de salida con extensión *".tgf"*, conteniendo todas las transacciones sintéticas generadas por el framework *jBasket*.
Básicamente, el *unificador de archivos individuales de datos sintéticos* recupera cada uno de los archivos generados con *jBasket*, de acuerdo al orden cronológico de los mismos, y construye un único archivo de salida concatenando todas las transacciones contenidas en cada archivo individual, anteponiéndoles el *timestamp* que se obtiene de convertir el nombre del archivo físico al cual pertenecen a un valor entero.

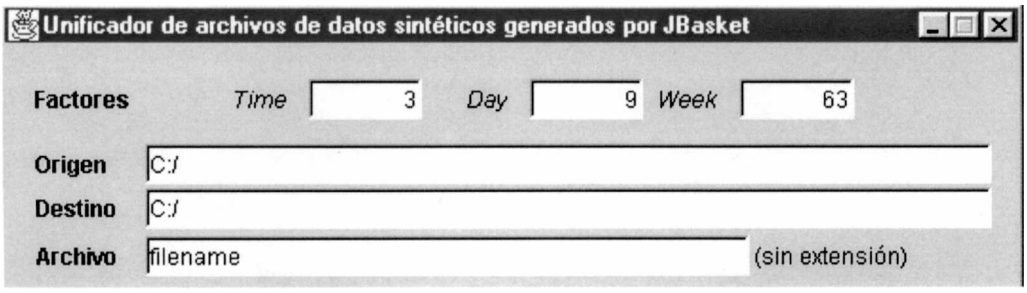

Figura 29
Parte superior de la GUI del unificador de archivos de datos sintéticos

Como vemos en la **Figura 29**, donde podemos apreciar parte de la interfase de esta aplicación y teniendo en cuenta el factor de generación WDT (*week*, *day*, *time*) utilizado para construir los archivos individuales de datos sintéticos con *jBasket*, al momento de generar la base de datos sintética única el usuario debe especificar los siguientes valores:

- ✓ los tres parámetros que definen los *factores de tiempo/momento*, *día* y *semana* utilizados para convertir el nombre de los archivos individuales de datos sintéticos a un valor numérico que represente su *timestamp*,
- ✓ el *directorio origen* donde se encuentran los archivos de datos sintéticos generados por *jBasket* a procesar,
- ✓ el *directorio destino* del archivo único de base de datos que será generado por la aplicación, y
- ✓ el *nombre del archivo a generar* (sin extensión).

El *unificador de archivos* ha sido desarrollado en Java y está formado por un conjunto de clases, las cuales se describen a continuación:

- Application: es la clase de entrada a la aplicación. Crea un objeto de clase FrameApp que se utiliza como interfase principal de la aplicación.
- FrameApp: es la GUI que define los componentes visuales de la aplicación.
- Convertor: es una clase abstracta que agrupa a todos los convertidores utilizados en la aplicación según el esquema calendario utilizado por *jBasket* para genera los archivos individuales de datos sintéticos.
  Esta clase define el método abstracto convert, el cual recibe como argumento el nombre del archivo individual de datos sintéticos a procesar y retorna un valor entero largo que representa el *timestamp* calculado. Dicho método debe ser

- el *timestamp* asociado al "1 de Enero de 1995" sea menor que el *timestamp* asociado al "2 de Enero de 1995".

Teniendo en cuenta las consideraciones mencionadas anteriormente, se ha desarrollado una aplicación que genera un único archivo de salida con extensión *".tgf"*, conteniendo todas las transacciones sintéticas generadas por el framework *jBasket*.
Básicamente, el *unificador de archivos individuales de datos sintéticos* recupera cada uno de los archivos generados con *jBasket*, de acuerdo al orden cronológico de los mismos, y construye un único archivo de salida concatenando todas las transacciones contenidas en cada archivo individual, anteponiéndoles el *timestamp* que se obtiene de convertir el nombre del archivo físico al cual pertenecen a un valor entero.

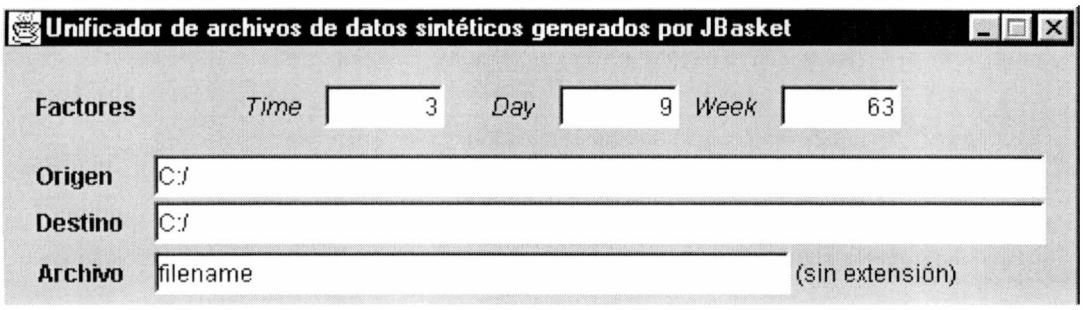

Figura 29
Parte superior de la GUI del unificador de archivos de datos sintéticos

Como vemos en la **Figura 29**, donde podemos apreciar parte de la interfase de esta aplicación y teniendo en cuenta el factor de generación WDT (*week*, *day*, *time*) utilizado para construir los archivos individuales de datos sintéticos con *jBasket*, al momento de generar la base de datos sintética única el usuario debe especificar los siguientes valores:

- ✓ los tres parámetros que definen los *factores de tiempo/momento*, *día* y *semana* utilizados para convertir el nombre de los archivos individuales de datos sintéticos a un valor numérico que represente su *timestamp*,
- ✓ el *directorio origen* donde se encuentran los archivos de datos sintéticos generados por *jBasket* a procesar,
- ✓ el *directorio destino* del archivo único de base de datos que será generado por la aplicación, y
- ✓ el *nombre del archivo a generar* (sin extensión).

El *unificador de archivos* ha sido desarrollado en Java y está formado por un conjunto de clases, las cuales se describen a continuación:

- Application: es la clase de entrada a la aplicación. Crea un objeto de clase FrameApp que se utiliza como interfase principal de la aplicación.
- FrameApp: es la GUI que define los componentes visuales de la aplicación.
- Convertor: es una clase abstracta que agrupa a todos los convertidores utilizados en la aplicación según el esquema calendario utilizado por *jBasket* para genera los archivos individuales de datos sintéticos.
  Esta clase define el método abstracto convert, el cual recibe como argumento el nombre del archivo individual de datos sintéticos a procesar y retorna un valor entero largo que representa el *timestamp* calculado. Dicho método debe ser

redefinido e implementado por cada una de las clases que heredan su comportamiento de la clase Convertor.
Cada subclase calculará el *timestamp* de acuerdo a la forma en la cual ha sido definida.

- DateConvertor: es una subclase de Convertor. Internamente, redefine el método convert de manera tal que cuando se ejecuta convierte el nombre del archivo individual de datos sintéticos recibido como argumento, a un valor entero largo que representa el *timestamp* de acuerdo a la fecha correspondiente en el *calendario gregoriano*.
  Este convertidor se aplica en el caso de que los nombres de los archivos individuales de datos sintéticos generados por *jBasket* tengan una estructura similar a **1995_1_1.dat**.
- WeekDateTimeConvertor: es una subclase de Convertor. Redefine el método convert de manera tal que, cuando se invoca, convierte el nombre del archivo individual de datos sintéticos recibido como argumento a un valor entero largo que representa el *timestamp* de acuerdo a los factores ingresados por el usuario. Este convertidor debe aplicarse cuando los nombres de los archivos individuales generados por *jBasket* han sido construidos utilizando un *calendar schema* WDT (*week*, *day*, *time*).
  Por ejemplo, si para generar los archivos de datos sintéticos se utilizó un *calendar schema* <9, 7, 3>, tomando como esquema base <1, 1, 1>; un archivo individual generado podría ser **7_5_2.dat** conteniendo las transacciones sintéticas del "segundo momento del día 5 de la semana 7".
- FileGenerator: es la clase que efectivamente se encarga de realizar la unificación de los archivos individuales de datos sintéticos. Para ello, recibe toda la información especificada por el usuario a través de la interfase (paths, factores, nombre de archivo de salida) y genera la base de datos sintéticos unificada concatenando todas las transacciones de los archivos individuales, identificando cada transacción con su *timestamp*.
- SortElement: es una clase que almacena un par de valores: *key*, *data*.
  Estos valores son utilizados para almacenar los nombres de los archivos individuales de datos sintéticos (*data*) con un valor entero largo (*key*) que representa su *timestamp*.
- Sorter: es utilizada para ordenar los elementos de un vector a partir del *método de la Burbuja*. Para llevar a cabo la ordenación, define el método sort que recibe un vector formado por objetos de la clase SortElement y los ordena de acuerdo al valor de su clave (*key*).

Como vimos, los archivos individuales de datos sintéticos han sido generados con *jBasket* utilizando un *calendar schema* WDT = {week: (1..9), day: (1..7), time: (1..3)}. Ahora bien, teniendo en cuenta este esquema calendario y que, por definición, la diferencia existente entre dos momentos consecutivos cualesquiera debe ser la misma, entonces para calcular el *timestamp* de las transacciones sintéticas de cada archivo individual utilizamos la clase WeekDateTimeConvertor pasándole como argumento los factores especificados en la **Figura 30**.

Antes de comenzar la generación de la base de datos sintéticos unívoca, el usuario debe indicar valores para los tres factores de cálculo de *timestamp*, el directorio origen de los

archivos generados por *jBasket*, y el directorio destino del archivo de base de datos unificado.
Luego, cuando el usuario presiona el botón *[Generar]*, se visualiza el inicio del proceso en la pantalla, mostrando *"Inicio de "* seguido por el nombre del archivo de salida especificado por el usuario.
A continuación, se crea una objeto de la clase FileGenerator, y se invoca el método generate de la misma pasándole como argumento los directorios origen y destino, el nombre de archivo de salida, y los factores especificados por el usuario.
Una vez finalizada la ejecución del método generate de la clase FileGenerator, se visualiza en la pantalla la finalización del proceso de unificación, mostrando *"Fin "* seguido por la cantidad total de transacciones, encerrado entre paréntesis, del archivo (unificado) de salida.

factor de *tiempo* o *momento*	3
factor de *día*	9
factor de *semana*	63

Figura 30
Factores utilizados en la unificación de los archivos individuales

Ahora bien, cuando un objeto de la clase FileGenerator recibe la invocación del método generate, con los parámetros antes mencionados, se construye el archivo de salida completando las siguientes tareas:

i. generar un vector con los nombres de los archivos individuales de datos sintéticos encontrados en el directorio origen,
ii. inicializar el contador de transacciones procesadas a 0 (cero),
iii. crear el archivo de salida con nombre igual al valor recibido como argumento, seguido de la extensión *".tgf"*,
iv. construir un vector ordenado por nombre, con los nombres de los archivos encontrados en el directorio origen (paso [i]) y los factores, y
v. generar un único archivo de datos sintéticos concatenando todas las transacciones de los archivos individuales, respetando el orden definido en el paso [iv]; asociándole a cada transacción un identificador unívoco (obtenido a partir del contador de transacciones) y el *timestamp* que se obtiene del nombre de archivo al cual pertenece.

Del conjunto de sentencias descriptas anteriormente, tenemos que los pasos [iv] y [v] son los más relevantes en el proceso de unificación. Por tal motivo, los mismos son descriptos a continuación con más detalle.

Para llevar a cabo el paso [iv], se invoca el método sortFiles pasándole como argumento un *vector de strings* con los nombres de los archivos (datos sintéticos) encontrados en el directorio origen y los factores para el tiempo, día y semana especificados por el usuario.
Este método crea un objeto de la clase WeekDateTimeConvertor, pasándole como parámetro los tres factores y, por cada nombre de archivo del vector, calcula su *timestamp*, invocando el método convert del objeto WeekDateTimeConvertor.
Luego, crea un objeto de la clase SortElement con el nombre de archivo (con extensión) y el valor del timestamp recientemente obtenido, que representan los valores de *data* y *key*, respectivamente.

Cuando termina de recorrer todos los archivos del vector, crea un objeto de la clase Sorter, e invoca el método sort del mismo para ordenar los archivos de acuerdo a su clave (*key*) aplicando el *método de ordenación de la Burbuja.*

Cuando se crea un objeto de la clase WeekDateTimeConvertor, se le pasan los factores para la semana, día y tiempo/momento, los cuales son almacenados internamente. Luego cuando el objeto recibe la invocación del método convert con un nombre del archivo individual, se procesa el nombre del archivo recibido a los efectos de identificar y recuperar los valores que representan la semana, el día y el tiempo o momento.

Veamos un ejemplo para entender mejor el funcionamiento de este método.
Supongamos que se invoca el método convert con el nombre de archivo **7_5_2.dat**.
El método, comenzando de izquierda a derecha, busca el carácter "_" (*underscore*) para identificar el valor que representa la semana.
Luego, asigna en la variable W la cadena de caracteres que se encuentran a la izquierda del *underscore*, en nuestro ejemplo W=7; y repite el proceso con la cadena de caracteres que se encuentra a la derecha *underscore*.
A continuación, busca la próxima ocurrencia del carácter *underscore* para identificar el valor del día y asigna en D la cadena de caracteres que se encuentra a la izquierda del *underscore*, en nuestro ejemplo D=5.
Por último, asigna en T la cadena de caracteres que se encuentra a la derecha de *underscore* para representar el tiempo/momento, son tener en cuenta la extensión del archivo (los caracteres que se encuentran a continuación del punto). En nuestro ejemplo T=2.

Una vez que han sido obtenidos los valores para W, D, y T; se calcula el *timestamp* de acuerdo a la siguiente fórmula, tomando los *factores de tiempo/momento*, *día* y *semana* recibidos como argumento:

$$timestamp = (((W - 1) * 63) + ((D - 1) * 9) + ((T - 1) * 3))$$

Figura 31
Fórmula utilizada para calcular el timestamp
de un archivo de datos sintéticos

Retomando el ejemplo propuesto, el *timestamp* del archivo **7_5_2.dat** se calcula aplicando la fórmula antes mencionada con los valores obtenidos del nombre del archivo, así:

$$\begin{aligned} timestamp\ (\textbf{7\_5\_2.dat}) &= (((7 - 1) * 63) + ((5 - 1) * 9) + ((2 - 1) * 3)) \\ &= ((6 * 63) + (4 * 9) + (1 * 3)) \\ &= (378 + 36 + 3) \\ &= 417 \end{aligned}$$

Figura 32
Cálculo del timestamp para el archivo de
datos sintéticos **7_5_2.dat**

Supongamos también que el archivo de datos sintéticos **7_5_2.dat** está formado por el siguiente conjunto de líneas:

```
8
1 3
3 6 34 41
1 1
4 18 23 34 49
3 6 23 47
2 6 49
3 6 25 42
7 5 12 16 22 43 45 46
```

Figura 33
Contenido del archivo de datos sintéticos **7_5_2.dat**

Como vimos en el apartado "**Generación de Datos Sintéticos**", este conjunto de sentencias indica, en la primera línea, que el archivo está formado por 8 transacciones, las cuales se detallan a partir de la segunda línea.
Cada transacción indica en la primera posición, empezando desde la izquierda, la cantidad de items que contiene. A continuación, se listan sus items separados por un espacio en blanco.

Supongamos ahora que, de acuerdo al orden establecido en el vector de nombres, corresponde unificar el archivo de datos sintéticos **7_5_2.dat**.
Para ello, se obtiene su *timestamp* tal cual se mostró en la **Figura 32**; y supongamos también que el contador de transacciones unificadas es igual a 797.
Entonces, para unificar las transacciones de este archivo, primero se "saltea" la primera línea que indica la cantidad de transacciones que contiene ya que no es tenida en cuenta para el proceso; y se repiten las lecturas mientras el archivo no llegue a su fin, es decir, no se lea el carácter EOF (*end-of-file*).
Por cada línea del archivo leída, se incrementa el contador de transacciones unificadas en una unidad y se genera una nueva línea en el archivo de salida con la siguiente estructura (de izquierda a derecha):

- el identificar unívoco (contador de transacciones), seguido por una coma y un espacio en blanco;
- el identificador temporal (*timestamp*), seguido por una coma y un espacio en blanco; y
- los items de la transacción original, separados por una coma y un espacio en blanco.

Una vez finalizado el procesamiento (*unificación*) del archivo individual **7_5_2.dat**, se continúa procesando el siguiente archivo del vector, de acuerdo al orden determinado por su *timestamp*.

La estructura de la base de datos sintética, incluyendo las transacciones del archivo individual **7_5_2.dat** queda como se muestra en la **Figura 34**.
Como puede verse, cuando las transacciones de un archivo individual son unificadas (concatenadas) en el archivo de base de datos sintéticos, no se incluye el valor numérico que indica la cantidad de items de cada transacción, ya que el mismo no es necesario

cuando se procesa el archivo a los efectos de encontrar los itemsets frecuentes y determinar, a partir de ellos, las reglas de asociación temporal.

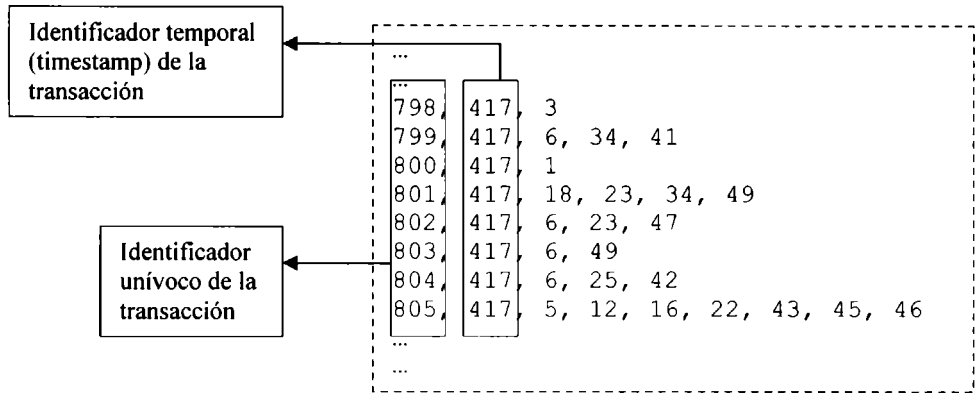

Figura 34
Contenido de la base de datos sintéticos una vez concatenadas las transacciones del archivo individual **7_5_2.dat**

Por otro lado, vemos que a cada transacción se le agrega un identificador unívoco (columna 1) y un identificador temporal (columna 2).
El *identificador unívoco* sólo se agrega a los efectos de poder identificar una transacción particular dentro de la base de datos. Además, el sentido común nos hace suponer que si tomamos una base de datos de transacciones real (por ejemplo: la base de transacciones de un supermercado) existirá alguna forma de identificar cada transacción independientemente del resto.
Esto no ocurre con el *identificador temporal* de las transacciones puesto que el identificador temporal de una transacción no es único porque, como podemos ver, todas las transacciones de intervalo de tiempo <7, 5, 2> tienen exactamente el mismo valor de *timestamp*, lo cual es lógico, ya que las mismas han sido recuperadas del archivo individual de datos sintéticos **7_5_2.dat**.

Pruebas de performance

Como se describió anteriormente, después de realizar varias pruebas con el generador de datos sintéticos *jBasket* tendientes a seleccionar el esquema que resulte más óptimo a la hora de realizar las pruebas de performance sobre el prototipo desarrollado, y teniendo en cuenta las limitaciones encontradas a la hora de probar el prototipo; se llegó a la conclusión de que las bases de datos a generar debían tener alrededor de 1000 transacciones cada una, ya que al momento de encontrar los itemsets frecuentes y generar a partir de ellos las reglas de asociación temporal, el tiempo requerido de proceso crecía exponencialmente a medida que el número de itemsets frecuentes encontrados iba aumentando.

Por tal motivo, se decidió que el esquema más conveniente a utilizar en la generación de las distintas bases de datos sintéticos es ***<week, day, time>***, donde *week* toma valores entre 1..9; *day* toma valores entre 1..7; y *time* toma valores entre 1..3.

Utilizando el framework *jBasket* y el *unificador de archivos sintéticos*, se generaron conjuntos de datos respetando el esquema de pruebas desarrollado en [1], asignándole los siguientes valores a las variables del archivo de configuración:

- N = 50
- L = 1.000
- | T |: 5, 10, y 20
- | I |: 2, 4, 6
- D = 6

Con estos valores, se han generado las siguientes bases de datos sintéticos:

Nombre	| T |	| I |	| D |
T5.I2.D1k	5	2	1k
T10.I2.D1k	10	2	1k
T10.I4.D1k	10	4	1k
T20.I2.D1k	20	2	1k
T20.I4.D1k	20	4	1k
T20.I6.D1k	20	6	1k

Figura 35
Bases de datos sintéticos generadas para testear el prototipo

Con estos datos sintéticos generados, se ha testeado el prototipo desarrollado a los efectos de determinar el comportamiento del mismo ante diferentes escenarios, manteniendo fija la cantidad de items utilizados en la generación de las bases de datos sintéticos y el promedio de transacciones incluidas por intervalo de tiempo.

Las pruebas han sido realizadas con el objetivo de comparar los algoritmos de asociación existentes *Apriori* y *TDIC*, poniendo especial énfasis en las características temporales introducidas y el impacto que las mismas tienen en la obtención de los resultados parciales (los resultados obtenidos en cada pasada) y los resultados finales una vez concluida la ejecución de los algoritmos.
Además de los ya mencionados, se tuvieron en cuenta otros factores que están relacionados principalmente con el momento (pasada) en que se generan los itemsets

candidatos; la cantidad de itemsets frecuentes encontrados al finalizar cada pasada; y el tiempo total consumido en procesar cada una de las bases de datos variando los valores asignados como umbral para el mínimo soporte, umbral para el mínimo soporte temporal y umbral para la mínima confianza.

En general, en lo que respecta a las reglas de asociación generadas, sólo se evaluó la cantidad total encontrada, sin tener en cuenta el valor semántico de las mismas, ya que las bases de datos están formadas por datos (transacciones) sintéticos y el valor de verdad de las reglas encontradas está sujeto a la aleatoriedad con que el framework *jBasket* selecciona los items individualmente y decide incluirlos o no en una transacción (para obtener más información sobre este tema, ver el apartado "**Generación de Datos Sintéticos**" de la página 109).

Las pruebas llevadas a cabo se realizaron sobre una PC con las siguientes características:

- Procesador **Intel Pentium IV**
- S.O. **Microsoft Windows 2000** (Service Pack 3)
- **512 Mb** de RAM
- Disco Rígido de **40 Gb**

Para realizar las pruebas fue necesario instalar el JDK 1.3 (Java Development Kit) y algunos paquetes específicos utilizados por la aplicación *Weka* desde su versión original, en la PC antes mencionada.

Básicamente, se llevaron a cabo tres tipos de pruebas:

- pruebas para comparar la cantidad de itemsets frecuentes encontrados,
- pruebas para comparar la cantidad de reglas de asociación generadas, y
- pruebas para comparar el tiempo de ejecución requerido para completar el proceso.

En todas las pruebas realizadas, se testeó el prototipo desarrollo incluyendo los algoritmos *TDIC* y *Apriori* (en su implementación original provista por *Weka* con las modificaciones necesarias para leer y procesar las bases de datos sintéticos con características temporales), con las bases de datos generadas utilizando *jBasket*, definidas en la tabla de la **Figura 35**, variando el valor del soporte, el valor de soporte temporal y la confianza de acuerdo a la siguiente tabla:

Mínimo Soporte	Mínimo Soporte Temporal	Mínima Confianza
0,25	5 10 20	0,25
0,5	20	0,25

Figura 36
Valores mínimos de soporte, soporte temporal y confianza utilizados en las pruebas de performance

Como vimos en el apartado "**Archivo de Configuración - Características generales**" (página 104), existen variables de proceso "comunes" a la ejecución de cualquier algoritmo de asociación y variables que están relacionadas específicamente con la ejecución del algoritmo *TDIC*.

En lo respecta a las "variables comunes" del archivo de configuración, se asignaron los valores mínimos de soporte, soporte temporal y confianza detallados en la **Figura 36**. Estos valores están relacionados con la construcción interna que define la variable **GRAL_DATA.PROCESS_FILES**, a saber:

```
[minsup=0.25, mintempsup=5, minmetric=0.25, numrules=1]
[minsup=0.25, mintempsup=10, minmetric=0.25, numrules=1]
[minsup=0.25, mintempsup=20, minmetric=0.25, numrules=1]
[minsup=0.5, mintempsup=20, minmetric=0.25, numrules=1]
```

De mismo modo, el valor de la variable de proceso **PROC_DATA.REMOVE_MISSING_COLS** se seteó a *false* ya que su utilización no era relevante a los efectos de las pruebas a realizar y el hecho de haberle asignado el valor *true* podría haber ocasionado un consumo de tiempo de proceso superior.

Por otro lado, para asignar valores a las variables que están relacionadas con la ejecución del algoritmo *TDIC* se tuvo en cuenta, principalmente, la manera en la cual habían sido creadas las bases de datos sintéticos con *jBasket*.
De esta manera, el valor asignado a las variables **GRAL_DATA.DIFERENCIA_TEMPORAL** y **GRAL_DATA.CANT_MOMENTOS_X_DIA**, es 1 y 3, respectivamente; ya que para crear los datos sintéticos se utilizó el esquema calendario ***<week: {1..9}, day: {1..7}, time: {1..3}>*** tomando tres momentos por día, con una diferencia temporal de 1.
Para calcular estos valores se tuvo en cuenta que la diferencia temporal entre dos momentos consecutivos cualesquiera es igual a 3. Como vimos anteriormente, este valor es utilizado en la fórmula (**Figura 31** de la página 121) para calcular el *timestamp* de un archivo de datos sintéticos.

Volviendo a la construcción interna de la variable **GRAL_DATA.PROCESS_FILES**, tenemos dos valores que han sido seleccionados de acuerdo a la conveniencia en la utilización de los recursos y el aprovechamiento de memoria. Estos valores son: **rcounter** y **deltat**.

En cuanto a la cantidad de transacciones leídas y procesadas juntas por cada algoritmo, podemos al algoritmo *Apriori* como un caso especial del algoritmo *TDIC*, en dónde la cantidad de transacciones leídas en cada proceso es igual a la cantidad total de transacciones de la base de datos.
Con esta configuración estaríamos desaprovechando una de las principales ventajas aportadas por el algoritmo *DIC* [3] que está relacionada con la generación de los itemsets frecuentes a medida que van siendo encontrados y su utilización en la generación de los itemsets candidatos para ser incluidos en el proceso en la siguiente lectura.
Por otro lado, si asignamos un número muy pequeño, por ejemplo 5, para procesar cada base de datos se deben realizar muchas lecturas de pocas transacciones cada una, reduciendo considerablemente la cantidad de itemsets frecuentes encontrados en cada

lectura y aumentando el esfuerzo requerido para generar los nuevos itemsets candidatos, a partir de los itemsets frecuentes encontrados en la lectura actual y los itemsets frecuentes encontrados en lecturas anteriores.

Por esta razón, y teniendo en cuenta que el tamaño promedio de las bases de datos sintéticos es de 1.000 transacciones, entonces se fijó el valor de la variable **rcounter** a 100. De esta manera, por cada corrida se realizarán 10 lecturas de 100 transacciones cada una sobre la base de datos siendo este un número *"ideal"* desde el punto de vista de la aplicación ya que se ve beneficiado el rendimiento del algoritmo al explotar sus características principales de la mejor manera.

De la misma forma, para definir el valor de la variable **deltat**, se tuvieron en cuenta otros factores ya que su valor, además de ser utilizado en la construcción de los histogramas, es de vital importancia cuando se ejecuta la función ***a-posteriori*** para encontrar los intervalos maximales en los cuales el itemset es frecuente.
Después de realizar varias pruebas, se determinó que el valor más óptimo para la variable **deltat** es 1.
Con este valor, los subintervalos del histograma tendrán una amplitud de 3 momentos, y contendrán todas las transacciones cuyos identificadores temporales estén contenidos en dicho subintervalo.
Por el lado de la función ***a-posteriori***, para cada itemset candidato que completó una pasada y no es frecuente en todo si lifespan, cada vez que se defina un nuevo subintervalo maximal se decrementará/incrementará, según sea el caso, el valor del mismo en un valor particular definido, entre otros valores, por la variable **deltat**.

Por último, tenemos las variables que están relacionadas con el *tipo de proceso* a realizar sobre la base de datos en cada caso, es decir, **GRAL_DATA.PROCCESS_TYPE** y **GRAL_DATA.FI_COUNTER**.
En la mayoría de los casos, a la variable **GRAL_DATA.PROCCESS_TYPE** se le ha asignado el valor 1. De esta manera, cada vez que se ejecuta el algoritmo *TDIC*, se completan ambas fases del proceso, buscando los itemsets frecuentes y generando a partir de ellos las reglas de asociación temporal.
Para estos casos, el valor de la variable **GRAL_DATA.FI_COUNTER** no es tenido en cuenta.
Sin embargo, hubo una pequeña cantidad de casos en donde el volumen de información manejada era muy grande y, por tal motivo era conveniente dividir el proceso en dos pasos bien diferenciados: por un lado el descubrimiento de los itemsets frecuentes y resguardo de los mismos para su posterior estudio y análisis; y por otro lado, la generación de las reglas de asociación tomando como entrada los conjuntos de itemsets previamente encontrados
En estos casos, a la variable **GRAL_DATA.FI_COUNTER** se le especificó el valor 3 permitiendo continuar con el proceso de generación de las reglas de asociación.

Como un caso excepcional, hubo unos pocos casos en donde la variable **GRAL_DATA.FI_COUNTER** tomó el valor 2, puesto que sólo se deseaba generar el reporte final con los itemsets frecuentes encontrados.

Con las configuraciones antes mencionadas, se realizaron las pruebas de performance sobre los algoritmos *Apriori* y *TDIC*.

En una primera tanda de pruebas, se estudió la performance de los algoritmos respecto de la cantidad de itemsets frecuentes encontrados durante la ejecución de la *fase 1* del proceso de generación de las reglas de asociación definido en [2][5] (para más información al respecto, ver apartado "**Descubrimiento de Reglas Temporales**", página 37).

En las **Figuras 37** y **38** podemos ver dos gráficos comparativos de la cantidad de itemsets frecuentes encontrados procesando el mismo conjunto de datos con los algoritmos *Apriori* y *TDIC*, respectivamente; fijando el valor para el mínimo soporte en 0,25 y el valor para el mínimo soporte temporal en 5.

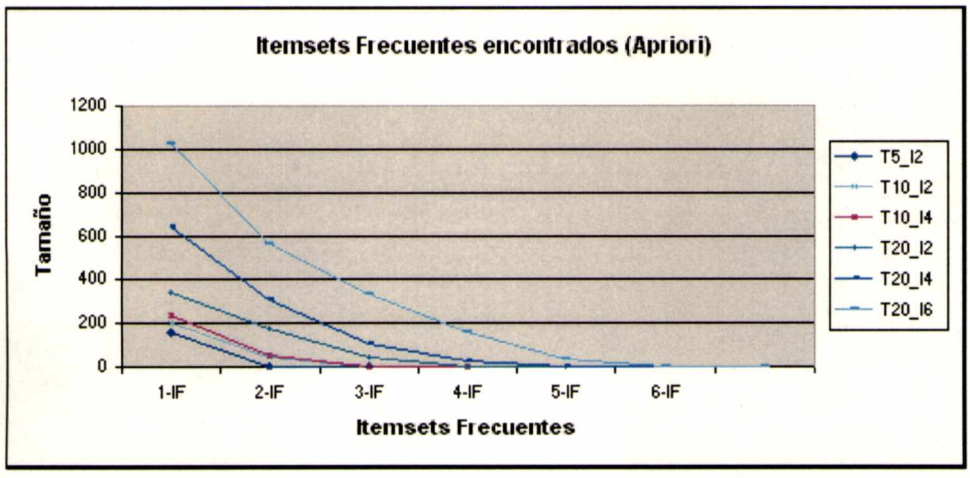

Figura 37
Itemsets Frecuentes descubiertos por *Apriori*.
Soporte=0.25 | Soporte Temp.=5

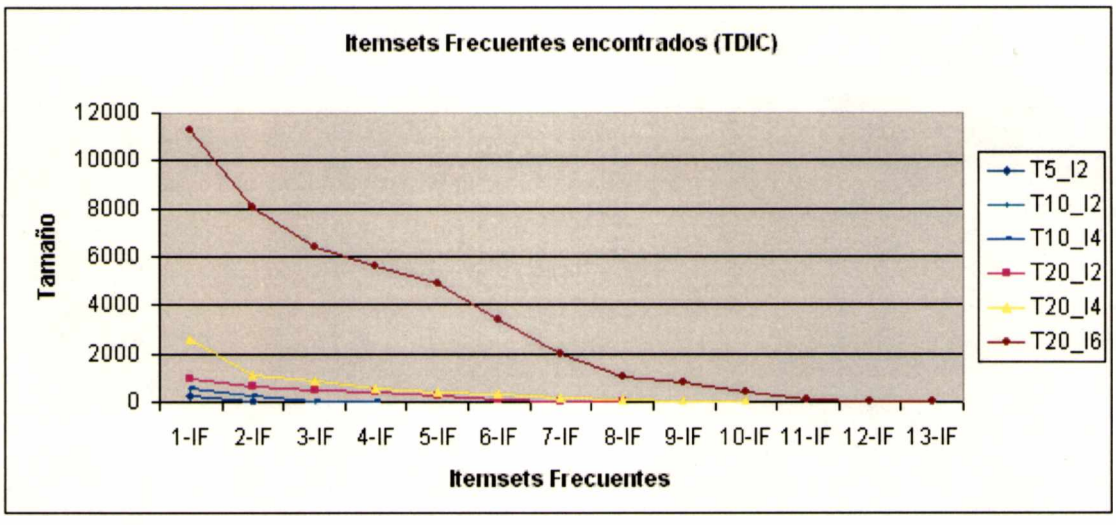

Figura 38
Itemsets Frecuentes descubiertos por *TDIC*.
Soporte=0.25 | Soporte Temp.=5

Allí podemos ver que en ambos casos se decrementa la cantidad de itemsets frecuentes encontrados a medida que aumenta el índice (valor k) de los itemsets frecuentes encontrados.

A primera vista podría parecer que los resultados obtenidos son incorrectos, pero si tenemos en cuenta la forma en la cual se generan los itemsets candidatos veremos que esta forma de pensar es errónea.
Para ello, es necesario revisar la definición de la función ***apriori-gen*** descripta en el apartado "**Generación de los candidatos Apriori**" de la página 21.
Internamente, para generar los itemsets candidatos de tamaño $k+1$, la función combina dos itemsets frecuentes de tamaño k tal que sus items cumplen ciertas restricciones de igualdad y la intersección de sus intervalos de frecuencia es mayor que el umbral para el mínimo soporte temporal definido por el usuario.
De esta manera, se generan algunos itemsets candidatos de tamaño $k+1$ que resulten de la combinación de dos itemsets frecuentes de tamaño k, pero que son desechados porque no cumplen con las restricciones antes mencionadas.

Por lo tanto, al tener menor cantidad de itemsets candidatos a procesar en la siguiente pasada, es lógico que los itemsets frecuentes derivados de estos itemsets candidatos existan en menor cantidad.

Por otro lado, vemos que la cantidad de itemsets frecuentes encontrados por *TDIC'* para los mismos valores de k (tamaño de los itemsets frecuentes) es siempre mayor.
Comparando ambos gráficos vemos que para las base de datos **T5_I2_D1k**, **T10_I2_D1k**, y **T10_I4_D1k** la cantidad de itemsets frecuentes encontrados es del doble de elementos cuando las mismas han sido procesadas con el algoritmo *TDIC'*; para la base de datos **T20_I2_D1k** la cantidad de itemsets frecuentes encontrados es tres veces mayor, para la base de datos **T20_I4_D1k** la cantidad de itemsets frecuentes encontrados es cuatro veces mayor; y para la base de datos **T20_I6_D1k** la cantidad de itemsets frecuentes encontrados es aproximadamente 10 veces más grande.
Esto se debe, en primer lugar, a la manera en que se construyen los itemsets candidatos; y en segundo lugar, a que el *factor tiempo* juega un papel muy importante a la hora de decidir si un itemset candidato que ha completado una pasada es frecuente o no.
Este es uno de los puntos en dónde se encuentran las mayores diferencias entre ambos algoritmos. En el caso del algoritmo *TDIC'*, cuando un itemset candidato ha recorrido y procesado todas las transacciones de la base de datos (decimos que ha completado una pasada), y se determina que no es frecuente en todo su período de vida, entonces se aplica la función ***a-posteriori***, para recuperar todos los intervalos frecuentes maximales, si es que existen, contenidos su lifespan en los cuales el itemset es frecuente.
Al invocar esta función, el algoritmo *TDIC'* le da una "última oportunidad" a los itemsets candidatos de ser frecuentes en al menos un subintervalo maximal que esté contenido en su lifespan.
Por esta razón, el algoritmo *TDIC* genera más itemsets frecuentes que el algoritmo *Apriori*.

Por último, podemos ver que la ejecución del algoritmo *TDIC'* ha generado una mayor variedad de itemsets frecuentes que el algoritmo *Apriori*. En particular, vemos que se

han generado más del doble de valores para *k* (13 contra 6) cuando se procesaron los datos sintéticos con *TDIC*.
El motivo de este suceso está fundamentado en los puntos antes mencionados, ya que, como es de suponer, al generar una mayor cantidad de itemsets candidatos, existe una alta probabilidad de que se generen más itemsets frecuentes; más aún teniendo en cuenta la aplicación de la función ***a-posteriori*** por parte del algoritmo *TDIC*.
Por lo tanto, se genera una relación cíclica que impulsa este hecho, ya que: a mayor cantidad de itemsets frecuentes de tamaño *k* encontrados se genera una mayor cantidad de itemsets candidatos de tamaño *k+1* para procesar; los cuales, eventualmente, se pueden convertir en itemsets frecuentes de tamaño *k+1*, y así sucesivamente.

Para finalizar con las comparaciones relacionadas sobre la cantidad de itemsets encontrados en cada proceso; vemos que en ambos algoritmos la cantidad de itemsets encontrados crece a medida que aumenta el valor de las variables T e I utilizadas en la generación de las bases de datos sintéticos por *jBasket* (ver **Figura 27**).
Esto fenómeno se debe principalmente a la forma en la cual *jBasket* construye las bases de datos en función de los valores asignados a las variables T e I.
En cualquier caso, a mayor valor de la variable T, nos encontramos con transacciones más grandes, es decir, con mayor cantidad de items. Por esta razón, crece la posibilidad de construir itemsets candidatos y frecuentes más grandes.
Por otro lado, a mayor valor de la variable I, existe una probabilidad más alta de que los itemsets candidatos que se forman a partir de las transacciones sean frecuentes.
Por tal motivo, es lógico que la cantidad de itemsets frecuentes encontrados en la base de datos **T5_I2_D1k** sea menor que la cantidad de itemsets frecuentes encontrados en la base de datos **T10_I2_D1k**, la cual a su vez es menor que la cantidad de itemsets frecuentes encontrados en la base de datos **T10_I4_D1k**, y así sucesivamente hasta ver que la cantidad de itemsets frecuentes encontrados en la base de datos **T20_I6_D1k** es mayor que todas las anteriores.

Por otro lado, en las **Figuras 39** y **40** podemos ver el gráfico comparativo de la cantidad de itemsets frecuentes encontrados procesando las mismas bases de datos sintéticas con los algoritmos *Apriori* y *TDIC*, respectivamente, pero asignando en este caso el valor 0,25 como mínimo soporte y el valor 10 como mínimo soporte temporal.

Como primera observación, vemos que la cantidad total de itemsets frecuentes encontrados en cada caso es menor, para el mismo valor de *k*, que la obtenida cuando el mínimo soporte temporal era igual a 5.
Con estos resultados comprobamos la importancia que tienen los umbrales (valores mínimos) a la hora de procesar una base de datos, ya que en este caso, estamos requiriendo que los intervalos de frecuencia de los itemsets frecuentes tengan el doble de amplitud que en el caso anterior.
Por lo tanto, existe una gran cantidad de itemsets candidatos que no son frecuentes ya que sus intervalos de frecuencia no cumplen con las nuevas restricciones de soporte temporal.

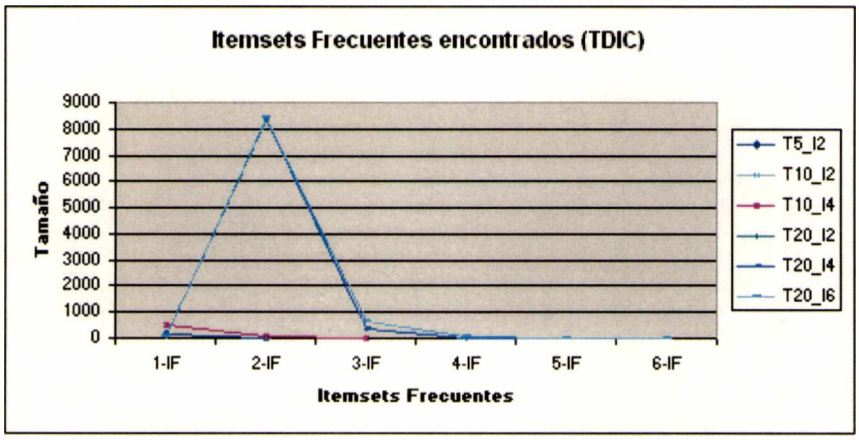

Figura 39
Itemsets Frecuentes descubiertos por *Apriori*.
Soporte=0.25 | Soporte Temp.=10

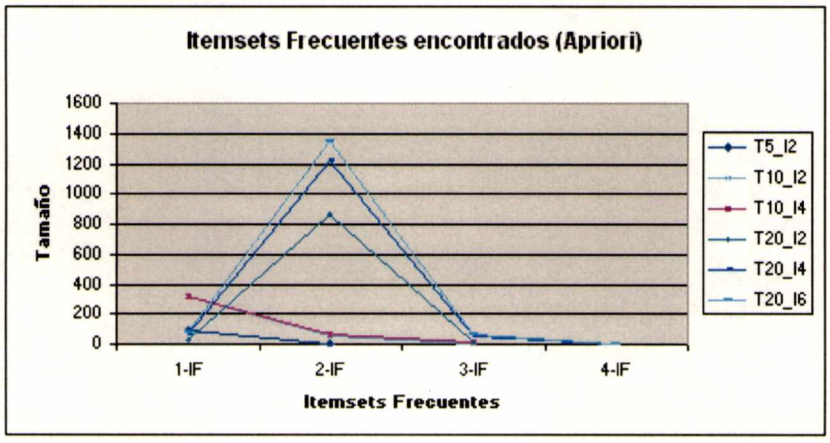

Figura 40
Itemsets Frecuentes descubiertos por *TDIC*.
Soporte=0.25 | Soporte Temp.=10

Seguimos analizando los gráficos precedentes y vemos que, a diferencia de los casos en los que el umbral para el soporte temporal era igual a 5 dónde veíamos que la curva de nivel decrecía de manera suave a medida que aumentaba el valor de *k*, cuando el umbral para el soporte temporal es igual a 10 se produce un pico máximo cuando se calculan los itemsets frecuentes de tamaño 2 para las bases de datos sintéticas generadas con T igual a 20, e I tomando los valores 2, 4, y 6.
En este caso particular, la única explicación que podemos encontrar está relacionada directamente con la forma en que *jBasket* genera dichas bases de datos.
Comparando estos gráficos con los analizados anteriormente, llegamos a la conclusión que el generador de datos sintéticos le asigna un valor semántico mayor a la unión de dos items dentro de una transacción, posibilitando de esta manera la creación de itemsets frecuentes en mayor número; manteniendo constante la distribución de los mismos en las transacciones de la base de datos.
De esta manera, encontramos que la proporción de itemsets frecuentes de tamaño 2 encontrados con soporte temporal igual a 5, es similar a la encontrada cuando el soporte

temporal es igual a 10; mientras que para el resto de los itemsets frecuentes esta proporción decrece considerablemente.

Además, analizando los archivos de salida generados al finalizar cada uno de los procesos, vemos que los intervalos de frecuencia de los itemsets frecuentes de tamaño 2 tienen mayor amplitud que los generados para el resto de los valores de *k*.
Por esta razón, al aumentar el valor del soporte temporal, el número de itemsets frecuentes de tamaño 2 se mantiene, mientras que para el resto de los itemsets frecuentes, la cantidad cae abruptamente.

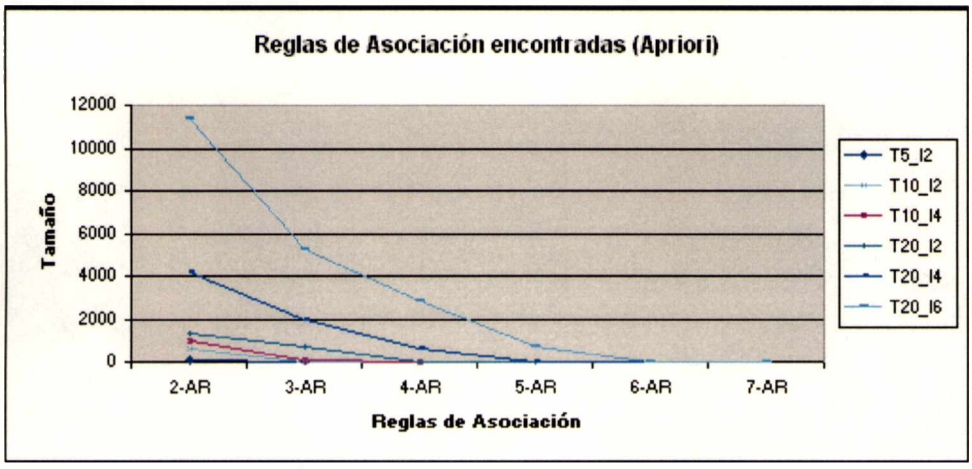

Figura 41
Tamaño de las reglas de asociación generadas por *Apriori*.
Soporte=0.25 | Soporte Temp.=5 | Confianza=0.25

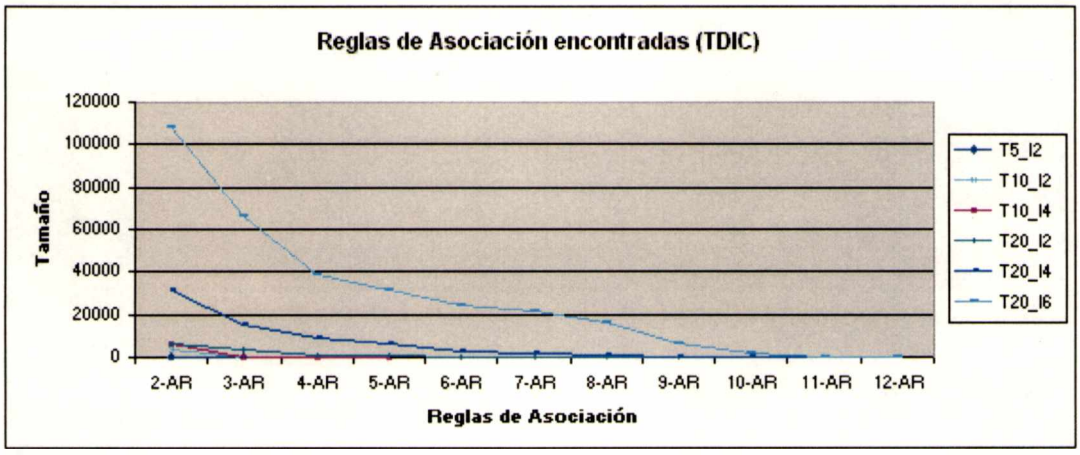

Figura 42
Tamaño de las reglas de asociación generadas por *TDIC*.
Soporte=0.25 | Soporte Temp.=5 | Confianza=0.25

La segunda tanda de pruebas estuvo enfocada en el estudio de la performance de los algoritmos *Apriori* y *TDIC* respecto de la cantidad de reglas de asociación encontradas una vez finalizado el proceso.
Este análisis está relacionado con la ejecución de la *fase 2* del proceso de generación de las reglas de asociación definido en [2][5] (para más información al respecto, ver apartado "**Descubrimiento de Reglas Temporales**", página 37).

En las **Figuras 41** y **42** podemos ver las curvas de nivel generadas en cada caso, asignando los siguientes valores a las variables de proceso: mínimo soporte igual a 0,25, mínimo soporte temporal igual a 5, y mínima confianza igual a 0,25.

Analizando en detalle las curvas de nivel generadas, vemos que se cumplen los tres puntos mencionados anteriormente cuando comparamos los dos algoritmos por la cantidad de itemsets encontrados durante el proceso, a saber:

a) en todos los casos, la cantidad de reglas de asociación generadas disminuye a medida que aumenta el valor del índice que indica el tamaño de las reglas de asociación de cada conjunto,
b) la cantidad de reglas de asociación generados por la ejecución de *TDIC* (en este caso hablamos de reglas de asociación temporal), es siempre mayor que cuando se procesan las bases de datos con *Apriori* para los mismos valores de k, y
c) el algoritmo *TDIC* generó tamaños de conjuntos dos veces más grandes (12 contra 6 en el mayor de los casos) que el algoritmo *Apriori*.

Los resultados obtenidos con este *conjunto de pruebas* se deben a la estrecha relación que existe entre los itemsets frecuentes encontrados durante el proceso de búsqueda y la generación de las reglas de asociación, ya que estas últimas son generadas utilizando los itemsets frecuentes previamente encontrados.

Por tal motivo, al tener menos cantidad de itemsets frecuentes a medida que aumenta el valor de k, entonces es razonable que tengamos una menor cantidad de reglas de asociación para generar, y determinar si su confianza está por encima del umbral especificado punto a).
Lo mismo ocurre en el punto b), ya que el algoritmo *TDIC* genera conjuntos de itemsets frecuentes (en el menor de los casos el conjunto generado por *TDIC* tiene el doble de itemsets frecuentes, y en el mayor de los casos es 10 veces más grande) de mayor tamaño que los generados por el algoritmo *Apriori*.
Con esta idea, es fácil imaginar que los conjuntos de reglas de asociación temporal generados a partir de estos conjuntos de itemsets frecuentes es mayor.

Más aún, teniendo conjuntos de itemsets frecuentes con mayor cantidad de elementos podemos generar mayor cantidad de reglas de asociación.
Por lo tanto, el punto c) puede verse como una consecuencia de los puntos a) y b) y cualquier otro tipo de análisis que se lleve a cabo sólo contribuirá a ratificar el razonamiento antes descripto.

En las **Figuras 43** y **44** podemos ver las curvas de nivel que muestran la cantidad de reglas de asociación generadas durante el procesamiento de las bases de datos sintéticas por ambos algoritmos, asignando los siguientes valores a las variables de proceso: mínimo soporte igual a 0,25, mínimo soporte temporal igual a 10, y mínima confianza igual a 0,25.

Analizando estas curvas de nivel, podemos ver que el comportamiento de los algoritmos cuando se varía el umbral para el soporte temporal es similar a lo visto anteriormente cuando comparamos la cantidad de itemsets frecuentes encontrados.

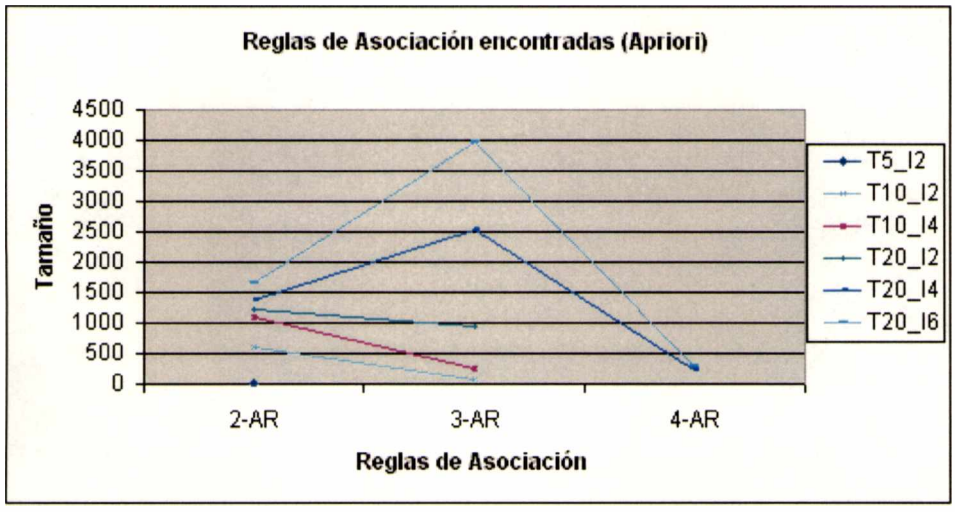

Figura 43
Tamaño de las reglas de asociación generadas por *Apriori*.
Soporte=0.25 | Soporte Temp.=10 | Confianza=0.25

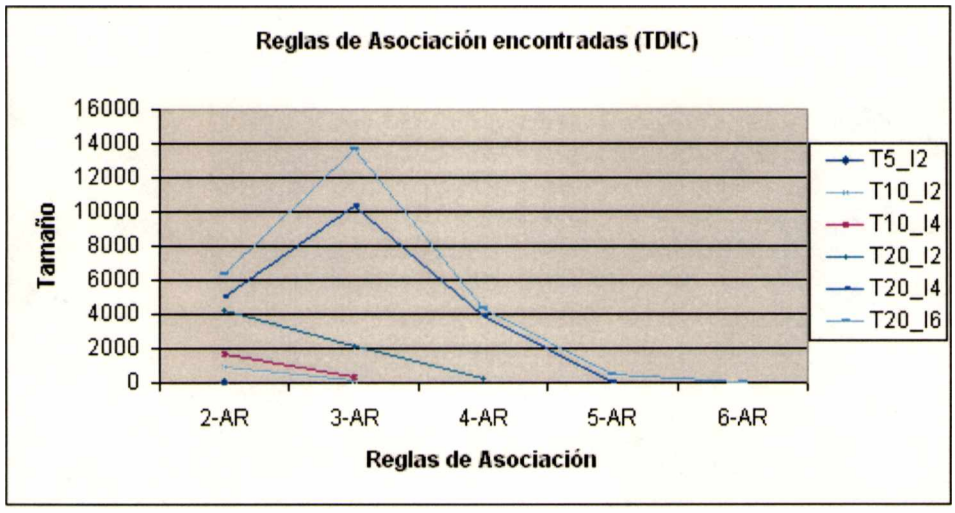

Figura 44
Tamaño de las reglas de asociación generadas por *TDIC*.
Soporte=0.25 | Soporte Temp.=10 | Confianza=0.25

Por otro lado, vemos que existe una simetría en cuanto a las curvas de nivel mostradas en las **figuras 39/40** y las mostradas en las **figuras 43/44**. Esto se debe principalmente a la relación existente entre los itemsets frecuentes encontrados y su posterior utilización en la generación de las reglas de asociación.

En este caso, podemos notar dos cosas. En primer lugar, el tamaño de las reglas de asociación generadas es aproximadamente la mitad que cuando se procesaron las bases de datos con un mínimo soporte temporal igual a 5.
En segundo lugar, la cantidad de itemsets frecuentes encontrados al ejecutar los algoritmos para el mismo valor de *k*, notamos lo siguiente: la cantidad de itemsets frecuentes encontrados es casi cuatro veces más grande al ejecutar el algoritmo *TDIC*, que cuando procesamos la base de datos con *Apriori*.

La tercer tanda de pruebas estuvo orientada a comparar los tiempos de ejecución requeridos por cada uno de los algoritmos para llevar a cabo el proceso completo de búsqueda de los itemsets frecuentes y generación de las reglas de asociación.

En el caso del algoritmo *TDIC*, vimos que el valor asignado a la variable de proceso **PROC_DATA.CALCULATE_PREMISE_SUPPORT** juega un papel muy importante, ya que su utilización está íntimamente relacionada con el tiempo requerido para realizar el proceso de generación de las reglas.
Como vimos, cuando el valor de esta variable es *false*, se asume que los items están distribuidos uniformemente en las transacciones de la base de datos, y por ello, no es necesario calcular el valor de soporte del itemset premisa en el intervalo de validez de la regla ya que, la variación de estos valores debería ser mínima, y no redundaría en una mejora considerable.
Si por el contrario, la variable es *true*, por cada regla de asociación a generar se debe calcular el valor de soporte de itemset premisa en el intervalo de validez de la regla antes de calcular la confianza de la misma.
Para realizar este cálculo, se utiliza el histograma que cada itemset premisa tiene asociado de la siguiente manera: contamos la cantidad de transacciones del histograma tal que su identificador temporal está incluido en el intervalo de validez de la regla.

Para llevar a cabo las pruebas de performance mencionadas asignamos *true* a dicha variable. Con esta decisión, estamos aceptando un costo mayor requerido para generar las reglas de asociación, priorizando el hecho de obtener reglas cuyos valores de confianza son más precisos.
Por otro lado, estamos seguros de que si hubiésemos elegido la otra opción, los tiempos de proceso requeridos para completar la fase de generación de las reglas de asociación hubiesen sido mucho menores en cada caso.

Con las consideraciones antes mencionadas y teniendo en cuenta que el requisito del cálculo del soporte de la premisa de regla es sólo propiedad del algoritmo *TDIC*, se han realizado pruebas con dicho algoritmo (únicamente) procesando las mismas bases de datos sintéticas utilizando en las pruebas anteriores, fijando el valor asignado a la

mínima confianza en **0,25**; variando el valor del mínimo soporte temporal para: **5**, **10**, y **20**; y tomando los siguientes valores para el mínimo soporte: **0,25**, **0,33**, **0,5**, **0,75**, **1**, **1,5**, y **2**.

En las **Figuras 45**, **46** y **47** vemos los gráficos del tiempo requerido, expresado en segundos, por el algoritmo *TDIC* para procesar las bases de datos sintéticos.
En los tres casos, los tiempos de ejecución requeridos para procesar la base de datos sintéticos **T5_I2_D1k** con los valores de soporte antes mencionados, están por debajo de los 95 segundos.
Por tal motivo, no han sido incluidos en los gráficos, ya que los tiempos requeridos para procesar el resto de las bases de datos sintéticos está muy por encima de dichos valores y su representación dificultaba la interpretación del resto de los valores de interés.

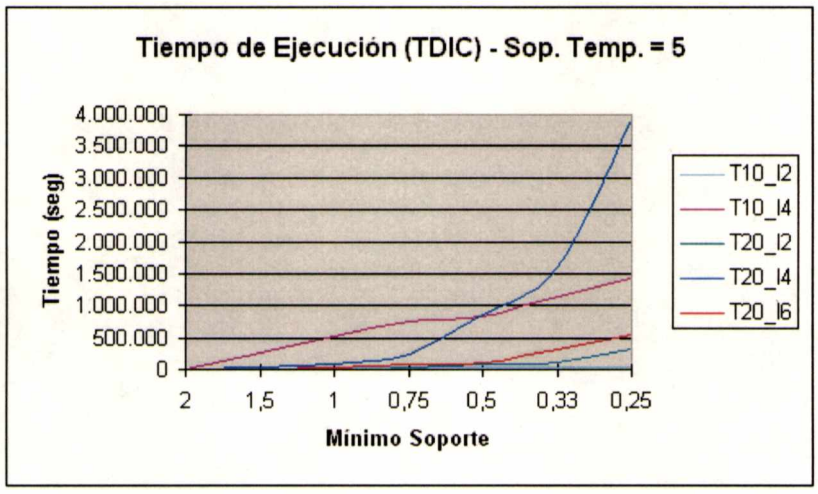

Figura 45
Tiempo de ejecución TDIC (con cálculo de soporte de la premisa)
Soporte Temp.=5 | Confianza=0.25

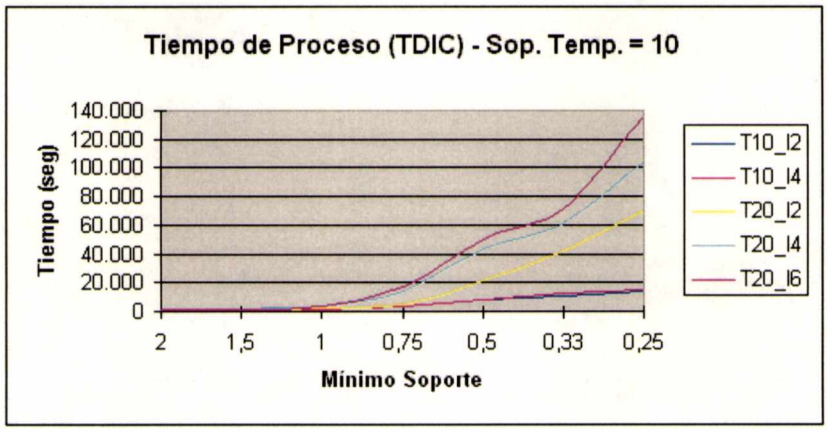

Figura 46
Tiempo de ejecución TDIC (con cálculo de soporte de la premisa)
Soporte Temp.=10 | Confianza=0.25

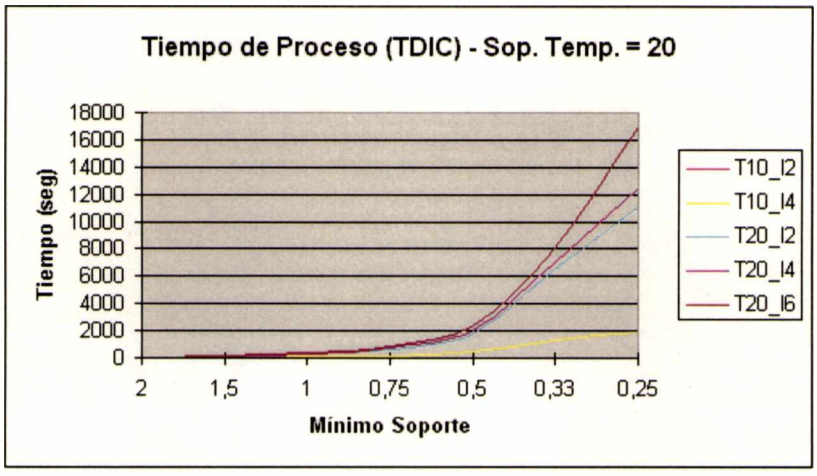

Figura 47
Tiempo de ejecución TDIC (con cálculo de soporte de la premisa)
Soporte Temp.=20 | Confianza=0.25

En primer lugar, vemos que el tiempo requerido por el algoritmo *TDIC* para realizar el proceso completo aumenta medida que decrece el umbral para el soporte seleccionado, independientemente del valor asignado al mínimo soporte temporal.
Esto se debe principalmente a la cantidad de itemsets candidatos y frecuentes encontrados durante el proceso de búsqueda y su posterior utilización en la generación de las reglas de asociación temporal, como vimos en las pruebas anteriores.

En segundo lugar, vemos que el tiempo requerido por el algoritmo *TDIC* es mayor para valores más pequeños de soporte temporal. De esta manera, vemos que, cuando ejecutamos el algoritmo sobre la misma base de datos con un mínimo soporte temporal de 5, el tiempo requerido en algunos casos está 10 veces por encima de los tiempos de proceso obtenidos cuando el algoritmo es ejecutado con mínimo soporte temporal igual a 10 o 20.

Esto se debe principalmente a que, cuando aumentamos el requerimiento para el soporte temporal, o sea que tomamos un umbral para el soporte temporal más pequeño; estamos obligando a que los intervalos de frecuencia de los itemsets frecuentes, y consecuentemente los intervalos de validez de las reglas de asociación temporal, tengan una amplitud menor.
Por un lado, esta apreciación parece ser más beneficiosa que perjudicial a los efectos del tiempo de proceso requerido pero esto no es así ya que, en el momento de aplicar la función ***a-posteriori*** para encontrar todos los intervalos maximales de un itemset, se recorre su histograma calculando el soporte del itemset en los "probables" intervalos de frecuencia generados a partir del intervalo de vida del itemset decrementando su amplitud, hasta determinar que el itemset es frecuente en dicho subintervalo o que la amplitud del mismo no es menor que el umbral especificado para el mínimo soporte temporal.
De esta manera, con valores de mínimo soporte temporal más pequeños tendremos mayores posibilidades de generar "probables" subintervalos en donde el itemset en cuestión es frecuente, y por tal motivo, el tiempo de proceso requerido en estos casos es mayor.

Por último, podemos ver los gráficos subsiguientes como una confirmación de las apreciaciones hechas en los puntos anteriores; ya que, en primer lugar y de acuerdo a la forma en la cual han sido creadas las bases de datos sintéticos utilizando *jBasket*, a medida que aumenta el valor de T e I, se van construyendo bases de datos en dónde las transacciones tienen mayor cantidad de items, los cuales se repiten con mayor frecuencia.
Por tal motivo, es más probable la generación de itemsets candidatos y frecuentes a partir de las bases de datos sintéticos con transacciones generadas de esta forma.
Consecuentemente, la cantidad de reglas de asociación temporal generadas en estos casos es mayor, y por lo tanto, el tiempo total de proceso requerido para procesarlas aumenta considerablemente.

Otro punto importante a tener en cuenta esta relacionado con el valor asignado como mínimo soporte y mínimo soporte temporal en cada caso, ya que a medida que el umbral asignado para el soporte decrece decimos que "estamos poniendo menos restricciones a los itemsets candidatos" y por tal motivo, el número de itemsets frecuentes encontrados es mayor, y el tiempo requerido para su descubrimiento aumenta exponencialmente.

Algo similar ocurre con el valor definido como umbral para el mínimo soporte temporal.
Como vimos anteriormente, en este sentido la función ***a-posteriori*** tiene una gravitación especial, ya que la forma en la cual desarrolla su tarea y el tiempo requerido para completarla están indefectiblemente relacionados con la cantidad de itemsets frecuentes encontrados, su posterior utilización en la generación de los nuevos itemsets candidatos y el tiempo total requerido para completar la *primera fase* del proceso llevado a cabo por el algoritmo.

Como última apreciación, cabe destacar que el valor especificado como umbral para la mínima confianza no tiene una importancia significativa en el tiempo total requerido para procesar una base de datos, ya que su utilización está relacionada exclusivamente con el filtrado de las reglas de asociación generadas.
Por tal motivo, teniendo en cuenta que primero se generan todas las reglas de asociación de un itemset frecuente y, posteriormente, se eliminan aquellas cuya confianza está por debajo del umbral especificado, entonces las variaciones en el umbral para la mínima confianza solamente tendrán incidencia en el tiempo requerido para visualizar los resultados finales en pantalla y generar automáticamente el archivo de salida correspondiente.
De lo anterior, llegamos a la conclusión que este tiempo no es significativo si tenemos en cuenta el tiempo total requerido para generar los datos de proceso en sus dos fases.

Para finalizar con las pruebas de *performance*, se han llevado a cabo un conjunto de pruebas tendientes a comparar los tiempos de proceso requeridos por cada algoritmo para completar las dos fases de proceso.
Como vimos en las pruebas descriptas anteriormente, la única manera de poder comparar los tiempos de ejecución de ambos algoritmos obliga a que asumamos una distribución uniforme en la base de datos de transacciones.
Por tal motivo, para llevar a cabo estas pruebas, a la variable de proceso **PROC_DATA.CALCULATE_PREMISE_SUPPORT** se le asignó el valor de *false* para que no se

calcule el soporte de la premisa en el momento de determinar la confianza de la regla, puesto que el algoritmo *Apriori* no incluye esta funcionalidad.

Con estas consideraciones, se han realizado pruebas de los algoritmos manteniendo fijo el valor de la mínima confianza en **0,25**; y variando los umbrales especificados para el soporte en **0,25**, **0,33**, **0,5**, **0,75**, **1**, **1,5**, y **2**; y el soporte temporal en **5** y **10**.

En las **Figuras 48** y **49**, podemos ver los resultados obtenidos al procesar las mismas bases de datos sintéticos con ambos algoritmos, fijando la mínima confianza en **0,25**, el umbral para el soporte en **0,25**, **0,33**, **0,5**; y el soporte temporal en **5** y **10**, respectivamente.
Antes de continuar analizando los resultados obtenidos, es importante documentar que las pruebas han sido realizadas tomando todos los valores de soporte antes mencionados, pero por una cuestión de practicidad, han sido tomados aquellos conjuntos de valores que pueden ser más representativos para describir el comportamiento de uno y otro algoritmo; y por tal motivo sólo se muestran en los gráficos los resultados cuando el soporte es igual a **0,25**, **0,33** o **0,5**.

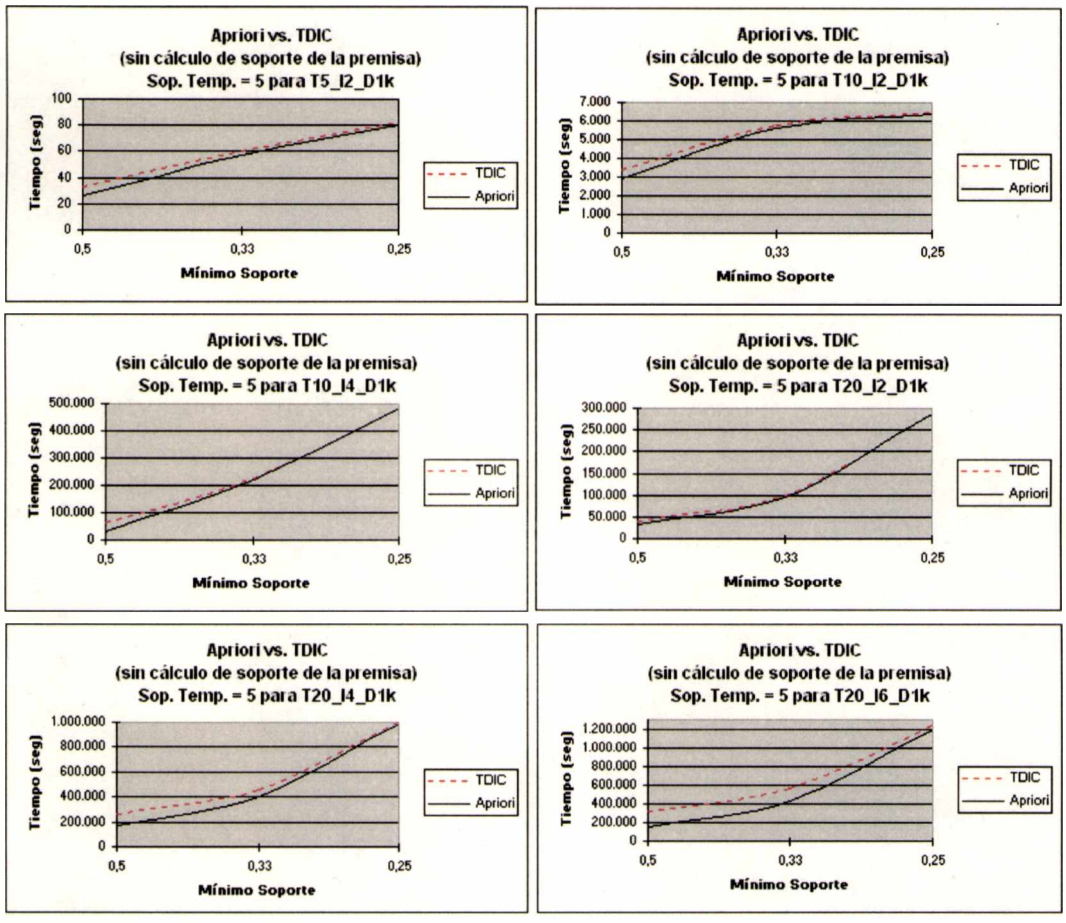

Figura 48
Tiempo de ejecución: Apriori vs. TDIC
(sin cálculo de soporte de la premisa)
Soporte Temp.=5 | Confianza=0.25

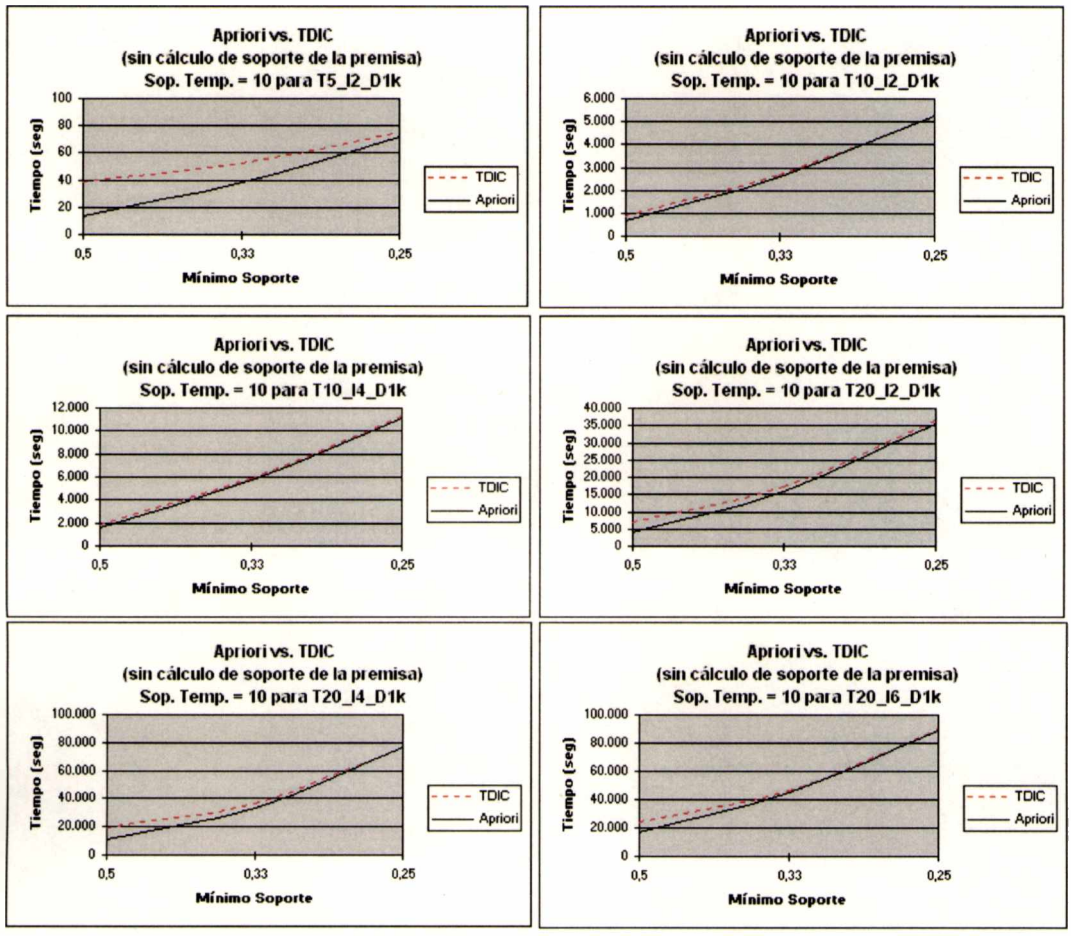

Figura 49
Tiempo de ejecución: Apriori vs. TDIC
(sin cálculo de soporte de la premisa)
Soporte Temp.=10 | Confianza=0.25

Ahora bien, de acuerdo a los resultados obtenidos vemos que, en la mayoría de los casos, el algoritmo *TDIC* requiere un mayor tiempo de ejecución para realizar las *dos fases del proceso*.
Sin embargo, la diferencia en los tiempos de ejecución requeridos por el algoritmo *Apriori* y *TDIC* se va reduciendo a medida que decrece el valor de soporte utilizado en las pruebas.
Estos resultados obtenidos obedecen principalmente a que a medida que se reduce el umbral para el mínimo soporte, el algoritmo *TDIC* se vuelve más eficiente ya que utiliza de manera más conveniente los recursos, generando menos pasadas sobre los datos (en algunos casos se redujo hasta un 35% el número de pasadas hechas sobre las transacciones de la base de datos).
Por otro lado, de acuerdo al análisis realizado anteriormente, el incremento en el tiempo de proceso del algoritmo *TDIC* se debe a que realiza una búsqueda más exhaustiva de los itemsets frecuentes tomando intervalos de vida que se ajustan a la existencia de cada itemset y su ocurrencia en las transacciones de la base.

De esta manera, a la búsqueda de cada itemset particular se le dedica más esfuerzo para determinar su posible frecuencia, lo cual representa un incremento en el tiempo total de proceso requerido para completar la tarea.

Por último, si comparamos estos resultados con los presentados en [3], vemos que los resultados obtenidos de procesar las bases de datos con el prototipo desarrollado (que implementa el algoritmo *TDIC*) coinciden con los obtenidos en [3], en donde se comparaban los algoritmos *Apriori* y *DIC*, entre sí.

En último término y a modo de ejemplo, se listan algunas reglas de asociación temporal generadas a partir de los itemsets frecuentes encontrados al procesar una base de datos sintética creada con *jBasket* a partir de los siguientes valores para |T|, |I| y |D|: 10, 6 y 1000 respectivamente.

```
39 => 21 : ([117,123]);
      Support=0.25;
      Temporal support=7;
      Confidence=[1.0, 0.937, 1.0, 0.937, 1.0, 0.958, 0.976, 1.0, 1.0,
                  0.937, 0.812, 1.0, 1.0, 1.0, 0.958, 1.0, 0.937]
28 => 46 : ([363,372]);
      Support=0.263;
      Temporal support=10;
      Confidence=[1.046]
39 => 18 : ([96,102]);
      Support=0.333;
      Temporal support=7;
      Confidence=[1.333, 1.25, 1.333, 1.25, 1.333, 1.277, 1.302, 1.333,
                  1.333, 1.25, 1.083, 1.333, 1.333, 1.333, 1.277, 1.333, 1.25]
45 => 39 : ([444,453]);
      Support=0.277;
      Temporal support=10;
      Confidence=[1.098, 1.111, 1.089, 1.06, 1.101, 1.111, 0.944, 1.111]
28 => 46 : ([525,531]);
      Support=0.25;
      Temporal support=7;
      Confidence=[0.994]
22 => 23 : ([327,339]);
      Support=0.25;
      Temporal support=13;
      Confidence=[0.903, 1.0, 0.958, 1.0, 0.968, 0.964, 1.0, 0.941, 0.977,
                  1.0, 0.958, 1.0, 0.979, 0.937]
33 => 29 : ([348,357]);
      Support=0.307;
      Temporal support=10;
      Confidence=[1.2, 1.23, 1.23, 1.212, 1.202, 1.224, 1.23, 1.23, 1.23,
                  1.179, 1.153, 1.23]
35 => 15 : ([150,165]);
      Support=0.266;
      Temporal support=16;
      Confidence=[1.034, 1.066, 1.066, 1.031, 0.993, 1.066, 1.028, 1.0,
                  1.066, 1.047, 1.066]
```

Figura 50
Reglas de asociación temporal generadas a partir de los
2-itemsets frecuentes encontrados
(sin cálculo de soporte de la premisa)
Soporte =0.25 | Soporte Temp.=5 | Confianza=0.25

```
25 45 => 17 : {[285,291]};
      Support=0.25;
      Temporal support=7;
      Confidence=[1.0]
17 45 => 25 : {[285,291]};
      Support=0.25;
      Temporal support=7;
      Confidence=[1.0]
17 25 => 45 : {[285,291]};
      Support=0.25;
      Temporal support=7;
      Confidence=[0.6]
25 32 => 17 : {[285,291]};
      Support=0.25;
      Temporal support=7;
      Confidence=[1.0]
17 32 => 25 : {[285,291]};
      Support=0.25;
      Temporal support=7;
      Confidence=[0.8]
17 25 => 32 : {[285,291]};
      Support=0.25;
      Temporal support=7;
      Confidence=[0.6]
```

Figura 51
Reglas de asociación temporal generadas a partir de los
3-itemsets frecuentes encontrados
(sin cálculo de soporte de la premisa)
Soporte =0.25 | Soporte Temp.=5 | Confianza=0.25

En las Figuras 50 y 51 se muestran las reglas de asociación temporal generadas a partir de los itemsets frecuentes de tamaño 2 y 3, respectivamente, descubiertos al procesar la base de datos sintética creada con los valores antes descriptos; utilizando 0.25 como umbral para el soporte, 5 como umbral para el soporte temporal y 0.25 como umbral para la confianza.
Además, estas reglas de asociación han sido calculadas asumiendo una distribución uniforme en los items de las transacciones de la base de datos sintéticos. Por tal motivo, no ha sido necesario volver a calcular el soporte de la premisa en el intervalo de la regla.

CONCLUSIÓN Y TRABAJOS FUTUROS

El presente trabajo de grado se basa en la aplicación *Weka* e incluye una implementación del *algoritmo TDIC*, propuesto por Ale-Rossi, para el descubrimiento de los *itemsets* que son *frecuentes maximales* y los utiliza para generar las reglas de asociación temporal, de acuerdo a parámetros especificados por el usuario.
La ejecución de dicho algoritmo se lleva a cabo tomando como entrada una base de datos sintética generada a partir del modelo *esquema calendario* propuesto por Yingjiu Li-Ning-Sean Wang-Sushil Jajodia con algunas adaptaciones que permiten identificar las transacciones generadas en el tiempo, asociándoles un indicador temporal que representa el momento en el que la transacción se llevó a cabo.

Utilizando las bases de datos sintéticas generadas, se han llevado a cabo diferentes pruebas a fin de medir la *performance* del algoritmo *TDIC* comparándolo con una implementación del algoritmo *Apriori* que acepta como entrada una base de datos con las características antes mencionadas, bajo diferentes circunstancias, variando los umbrales para el soporte, soporte temporal, la confianza y la amplitud de los intervalos de frecuencia requeridos en cada caso.

Algunos trabajos futuros posibles derivados del presente trabajo de grado podrían ser los siguientes:

- ✓ Utilizar los *archivos temporales de proceso* generados a partir de la ejecución de los algoritmos de asociación incorporados en el trabajo de grado, como entrada para otras aplicaciones tomándolos como base de conocimiento para procesar otras bases de datos sintéticos que contengan transacciones que se han producido en instantes de tiempo posteriores a los datos ya procesados.
- ✓ Utilizar el prototipo desarrollado (que incluye una implementación del algoritmo *TDIC*) para analizar la calidad de las reglas de asociación generadas basándose en el conocimiento adquirido en el procesamiento de los itemsets y los histogramas generados a partir de los mismos.
- ✓ Utilizar, con una leve modificación, la aplicación de unificación de datos sintéticos parciales para generar de bases de datos sintéticas tomando como entrada datos sintéticos parciales generados utilizando otros esquemas de representación diferentes al *esquema calendario*.

Estos y otros desarrollos podrían utilizar el modelo desarrollado en el trabajo de grado como base de conocimiento, respetando las características de las bases de datos procesadas, la estructura de los itemsets frecuentes encontrados y el tipo de reglas de asociación generadas sin necesidad de recodificar la forma en la cual los mismos son descubiertos y procesados.

REFERENCIAS BIBLIOGRÁFICAS

- ✓ [1] Agrawal, R.-Srikant, R.: **Fast Algorithms for Mining Associations Rules**. IBM Res. Rep. Rj9838, IBM Almaden. June 1994.
- ✓ [2] Ale, J.-Rossi, G.: **An Approach to Discovering Temporal Associations Rules**. Proc. ACM 15th Symposiumon Applied Computing, pages 294-300. March 2000.
- ✓ [3] Brim, S.-Motwani, R.-Ullman, J.-Tsur, S.: **Dynamic Itemset Counting and Implication Rules for Market Basket Data**. Proc. ACM SIGMOD: 255-264. 1997.
- ✓ [4] Chan, K.-Fu, A.: **Efficient Time Series Matching by Wavelets**. Proc. IEEE 15th Intl. Conf. On Data Engineering, 1999.
- ✓ [5] Ale, J.-Rossi, G.: **The Itemset's Lifespan Approach to Discovering Temporal Associations Rules**. Proc. ACM Workshop on Temporal Data Mining (KDD-2002). Julio 2002.
- ✓ [6] Agrawal, R.-Srikant, R.: **Mining Generalized Associations Rules.**
- ✓ [7] Park, J. S. – Chen, M. S. – Yu, P. S. : **An Effective Has-Based Algortithm for Mining Association Rules.**
- ✓ [8] Yingjiu Li, Peng.-Ning, X.-Sean Wang-Sushil Jajodia: **Generating Market Basket Data with Temporal Information.**
- ✓ [8a] Yingjiu Li, Peng.-Ning, X.-Sean Wang-Sushil Jajodia: **User's Manual of jBasket.**
- ✓ [9] Agrawal, R.-Imielinski, T-Swami, A.: **Mining Associations Rules between Sets of Items in Large Databases.** Proc. ACM SIGMOD Conference on Management of Data, pages 207-216, Washington, D.C., May 1993.
- ✓ [10] Houtsma, M-Swami, A.: **Set-oriented mining of associations rules.** Research Report RJ 9567, IBM Almaden Research Center, San Jose, California, October 1993.
- ✓ [11] Agrawal, R.-Srikant, R.: **Fast algorithms for mining associations rules in large databases**. IBM Res. Rep. RJ 9839, IBM Almaden Research Center, San Jose, California. June 1994.
- ✓ [12] Witten, I – Frank, E. : **Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations**.
- ✓ [13] Cabena, P-Hadjinian, P-Stadler, R-Verhees, J-Zanasi, A: **Discovering Data Mining. From Concept to Implementation**.

ANEXO I

En este anexo se describe la documentación complementaria entregada. Dicha documentación está relacionada con el prototipo desarrollado y las aplicaciones y/o software adicional requerido para el correcto funcionamiento del mismo.

Podemos catalogar la documentación entregada de la siguiente manera:

- Aplicaciones y/o software externo requerido.
  - Instalador *Java 2 Runtime Environment* versión 1.3,
  - Instalador de la aplicación *WEKA* en su versión base,
  - Archivos de funcionalidad básica de *Java* empaquetada (*JARs*),
  - Archivos fuentes que constituyen la aplicación *jBasket* utilizada para generar las bases de datos sintéticos.

- Aplicaciones y/o software desarrollado específicamente.
  - Archivos fuentes que constituyen el prototipo desarrollado (aplicación *WEKA* modificada),
  - Archivos fuentes que constituyen la aplicación *FileGenerator* utilizada para unificar los archivos independientes de datos sintéticos,
  - Archivos de bases de datos sintéticos que fueron utilizados para llevar a cabo las *pruebas de performance*.

El *Java Runtime Environment* (versión 1.3) se utiliza como entorno de ejecución de las aplicaciones desarrolladas en Java.

Por otro lado, como vimos anteriormente, se utilizó la aplicación *WEKA* como base para el desarrollo e implementación del prototipo propuesto. Por tal motivo, se adjunta el instalador de la aplicación *WEKA* en su versión original con el objetivo de que sirva como base de comparación de las funcionalidades adicionales desarrolladas y de las modificaciones que le han sido introducidas para cubrir la funcionalidad requerida por el prototipo.

Para el correcto funcionamiento del prototipo ha sido necesaria la inclusión del archivo de funcionalidades *Java* empaquetadas específico, conocido como **jbcl3.1.jar**.

En lo que respecta al generador de datos sintéticos ***jBasket*** utilizado, solo se modificó el archivo de configuración (***ParametersBundle.properties***) para generar los archivos independientes de datos sintéticos de acuerdo al período en el cual se llevaron a cabo las *pruebas de performance*.

Las clases desarrolladas para implementar el prototipo propuesto se adjuntan con las clases de la aplicación *WEKA* original para permitir la creación de un entorno de prueba unificado sin contratiempos.
Asimismo, se adjuntan los archivos del prototipo que han sido expresamente desarrollados para cubrir las funcionalidades requeridas. Esta decisión se tomó con el objetivo de entregar un conjunto de archivos que contienen la "nueva funcionalidad" y permitir de esta manera una rápida revisión de la misma.

Sin embargo, cabe aclarar que la nueva funcionalidad requerida para el prototipo no está totalmente contenido en nuevas clases Java, ya que se intentó mantener la funcionalidad

provista por la aplicación *WEKA* en su versión base. Por tal motivo, ha sido necesario modificar algunas clases originales de la aplicación para incluirle la funcionalidad que estábamos necesitado.
Estas últimas clases a las que hago mención, no son listadas independientemente del resto para no crear confusión en el momento del análisis.

Como vimos en los apartados anteriores, para realizar las pruebas de performance se utilizaron algunas bases de datos sintéticos generadas utilizando el generador denominado *jBasket*. Dicho generador, crea un conjunto de archivos independientes especificando en cada uno de ellos las transacciones que han sido asignadas a dicho intervalo de tiempo.
De acuerdo a la forma en la cual se especifica la base de datos a procesar en la aplicación *WEKA*, esta forma de generación de archivos de datos sintéticos no era conveniente. Por esta razón, se desarrolló la aplicación *FileGenerator* que toma todos los archivos independientes de datos sintéticos, previamente generados por *jBasket*, y devuelve una única base de datos conteniendo todas las transacciones especificadas en los archivos de entrada, con las características temporales de cada uno de ellos.

En último término, se incluyen los archivos de datos que fueron utilizados para llevar a cabo las *pruebas de performance* del prototipo.
Estos archivos de datos sintéticos han sido generados utilizando el generador *jBasket*, siguiendo los valores especificados en la **Figura 36** y unificados utilizando la aplicación *FileGenerator* antes descripta.
Estos archivos se incluyen en la documentación a modo de ejemplo, pudiendo ser nuevamente utilizados en pruebas subsiguientes, o bien a los efectos de comparación para evaluar la cantidad de transacciones existentes en cada base de datos, la frecuencia con que los atributos (ítems) ocurren en las transacciones de la misma base o de distintas bases, o bien la cantidad de transacciones por intervalo de tiempo contenidas en un archivo.

ANEXO II

En este anexo se describe la instalación, configuración y modo de ejecución de los archivos de aplicaciones que conforman el trabajo de grado y las aplicaciones complementarias.
El manera en la cual se debe llevar a cabo la instalación, configuración y ejecución de las aplicaciones y/o software externo requerido se detalla a continuación.

Para el correcto funcionamiento de las aplicaciones desarrolladas en el lenguaje Java, la PC donde se ejecute el software debe tener instalado el *Java 2 Runtime Environment*. Para ello hay que ejecutar el archivo **j2re1_3_0-win-i.exe** que se encuentra en la carpeta **..\Santiago Clac (1506-9)\Trabajo de Grado - Entrega Final\JDK 1.3 (JRE)** del CD de instalación provisto como documentación, aceptando las opciones por defecto sugeridas a medida que avanza la instalación.
Una vez finalizada la instalación del mismo, de acuerdo al idioma del Sistema Operativo, la ruta donde queda instalada la aplicación es ***C:\Archivos de programa\JavaSoft\JRE\1.3***.

Otro de los requisitos para el correcto funcionamiento de la aplicación, es agregar los archivos de funcionalidad básica empaquetada (*JARs*) de *Java*. Para ello, hay que copiar el archivo **jbcl3.1.jar** que se encuentra en la carpeta ***..\Santiago Clac (1506-9)\Trabajo de Grado - Entrega Final\JARs*** del CD de instalación provisto como documentación, a la carpeta ***[Ruta donde está instalado el JRE]\lib*** donde se instaló. En el caso de que se haya realizado una instalación aceptando los valores sugeridos por defecto, el archivo debe ser copiado en la carpeta ***C:\Archivos de programa\JavaSoft\JRE\1.3\lib***).

Una vez completados los pasos listados anteriormente, procedemos a la instalación, configuración y ejecución de las aplicaciones y/o software desarrollado o modificado específicamente para el trabajo de grado.
Para ello, sugiero seguir los pasos que se detallan a continuación.

Para la generación de bases de datos sintéticas se ha adaptado el generador de datos sintéticos de dominio público, conocido como *jBasket*. Dicho framework no requiere de una instalación propiamente dicha, sino que es suficiente con copiar los archivos que constituyen el framework en una carpeta del disco local y ejecutar la aplicación a través del visualizador de *Java*.

Para mantener el orden de las aplicaciones recomiendo que las mismas sean copiadas a partir de un directorio raíz común a las mismas. De esta manera será más fácil especificar la ruta donde se encuentran los archivos al momento de la ejecución de las aplicaciones.
Dentro del CD de instalación provisto como documentación se adjunta un archivo de ejecución por lotes, **instalar.bat**. Dicho archivo se encuentra ubicado en ***..\Santiago Clac (1506-9)\Trabajo de Grado - Entrega Final***.
Cuando se ejecuta ese archivo, se copian las aplicaciones ***jBasket***, ***FileGenerator*** y ***Trabajo de Grado*** en la siguiente ubicación: ***C:\Temp\SantiagoClac-15069*** permitiendo tener en un solo lugar las aplicaciones que permiten generar los archivos individuales de datos sintéticos, las bases de datos sintéticos a partir de los archivos

individuales, y ejecutar el *algoritmo TDIC* a partir del prototipo desarrollado basado en la aplicación *Weka*.

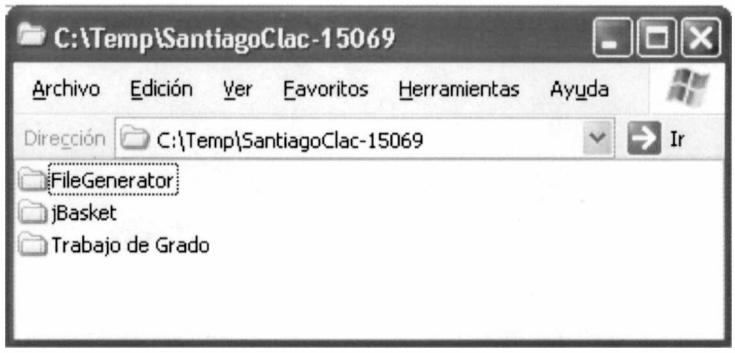

Figura 52
Contenido de la carpeta genera automáticamente
al ejecutar el archivo **instalar.bat**.

Por último, para simplificar la tarea de configurar cada una de las aplicaciones a la hora de ejecutarlas, para cada una de ellas se ha suministrado un archivo de ejecución por lotes (*.bat*) que realiza las configuraciones necesarias para ejecutar la aplicación en cuestión.
Para permitir identificarlos rápidamente, el nombre de cada uno de estos archivos está formado de la siguiente manera: ***nombreaplicacion*_ejecutable.bat**. Sin embargo, esta nomenclatura no imposibilita que el archivo se llame de otra forma ya que no se hace referencia a este nombre en ningún momento fuera de este ámbito.

De esta manera, para generar los archivos de datos sintéticos individuales, hay que ejecutar la aplicación *jBasket*. Esta tarea se lleva a cabo haciendo *doble click* en el archivo **jbasket_ejecutable.bat**, tal cual se muestra en la **Figura 53**.

Figura 53
Contenido de la carpeta *jBasket* que permite generar
los archivos individuales de datos sintéticos.

Luego, si lo que se quiere es unificar los archivos de datos sintéticos individuales en una única base de datos, debe ejecutar la aplicación *FileGenerator*. Para ello, hay que hacer *doble click* sobre el archivo **filegenerator_ejecutable.bat**, tal cual se muestra en la **Figura 54**.

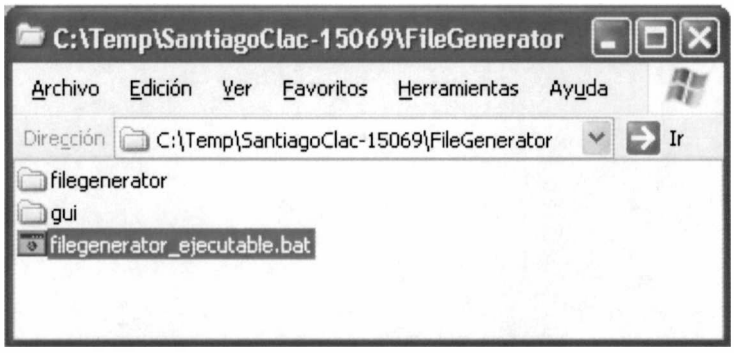

Figura 54
Contenido de la carpeta *FileGenerator* que permite generar
la base de datos sintéticos.

En lo que respecta a las aplicaciones propiamente dichas, es importante indicar que cuando estamos ejecutando la aplicación *FileGenerator* para unificar los archivos individuales de datos sintéticos generados con *jBasket*, tanto las carpetas de la ruta origen de los archivos generados con *jBasket*, como las carpetas de la ruta destino del archivo de base de datos que se va a generar, debe estar separada por "/" en lugar de "\" ya que *Java* utiliza este carácter en combinación con otros para indicar caracteres especiales, por ejemplo "\n" salto de línea.

Por último, para probar el prototipo desarrollado, hay que hacer *doble click* sobre el archivo **trabajodegrado_ejecutable.bat** que se encuentra en la carpeta *Trabajo de Grado*, tal cual se muestra en la **Figura 55**.

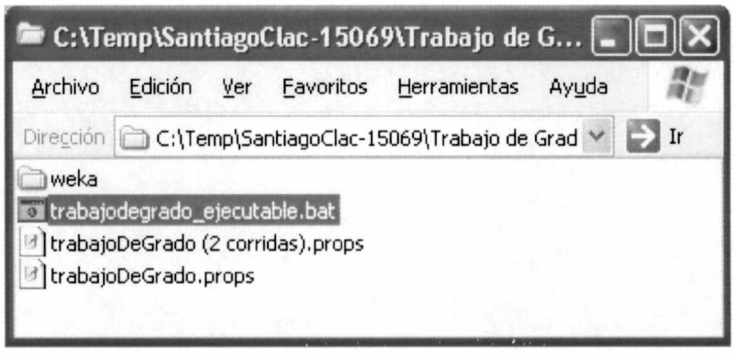

Figura 55
Contenido de la carpeta Trabajo de Grado que contiene
el prototipo desarrollado y permite ejecutar el algotirmo *TDIC*.

En el caso particular del prototipo desarrollado, vemos en la Figura 55 que hay dos archivos de configuración (extensión *.props*). Esto se debe a que el archivo de configuración ***trabajoDeGrado (2 corridas).props*** es anexado a modo de ejemplo para

indicar la forma de especificar dos o más corridas utilizando el mismo prototipo con diferentes valores de proceso.

El prototipo desarrollado recupera las variables de proceso del archivo con nombre ***trabajoDeGrado.props***. Por tal motivo, si se desea ejecutar el algoritmo *TDIC* desarrollando múltiples procesos, hay que renombrar convenientemente el archivo de configuración ***trabajoDeGrado (2 corridas).props*** para que puede ser interpretado por la aplicación (eliminándole la cadena de caracteres "***(2 corridas)***").