Università di Padova - DEI
Corso di Data Mining, Anno Accademico 2001-2002

Term paper
Ricerca di itemset frequenti

Studente: Carlo Fantozzi, XVI ciclo

1 Introduzione

La ricerca di itemset frequenti è un sottoproblema che nasce dalla ricerca di *association rules*, uno dei problemi classici di data mining. Per poter definire che cosa si intende per association rules e itemset frequenti è necessario introdurre subito alcune definizioni.

Sia $\mathcal{I} = \{i_1, i_2, \ldots, i_N\}$ un insieme di oggetti o *item*, e sia $\mathcal{D}$ tale che $t \subseteq \mathcal{I}\ \forall t \in \mathcal{D}$; gli elementi di $\mathcal{D}$ prendono il nome di *transizioni*, mentre i sottoinsiemi di $\mathcal{I}$ prendono il nome di *itemset*. Si noti che $\mathcal{D}$ può contenere più elementi uguali tra loro, mentre ciò non è vero per $\mathcal{I}$. Si definisce *supporto* di un itemset $\ell \subseteq \mathcal{I}$ la frazione di transizioni che lo contengono, ovvero la quantità

$$\mathrm{Sup}\,(\ell) := \frac{|\{t \in \mathcal{D} \text{ t.c. } \ell \subseteq t\}|}{|\mathcal{D}|}$$

Sono infine date due costanti reali s (*minimo supporto*) e c (*minima confidenza*) tali che $0 \leq s, c \leq 1$. Con riferimento alla notazione appena introdotta, una *association rule* è una coppia ordinata (X, Y) di itemset che gode delle seguenti proprietà:

P1. $X \cap Y = \emptyset$;

P2. $\mathrm{Sup}\,(X \cup Y) \geq s$;

P3. $\mathrm{Sup}\,(X \cup Y)\,/\mathrm{Sup}\,(X) \geq c$.

Nel seguito si userà la notazione standard della comunità di data mining, denotando una association rule con "$X \Rightarrow Y$". Si noti che una association rule non deve essere necessariamente soddisfatta da tutte le transazioni nel database: la proprietà P3, infatti, indica che si accettano tutte le rules $X \Rightarrow Y$ verificate da almeno una frazione c delle transazioni contenenti X. Risolvere il problema delle association rules significa determinare tutte le association rules compatibili con $\mathcal{D}$, $\mathcal{I}$, s e c.

Fin dagli inizi [3] il problema delle association rules è stato affrontato dividendo la computazione in due fasi distinte.

1. Nella prima fase vengono determinati tutti gli itemset $\ell \in \mathcal{I}$ tali che $\mathrm{Sup}\,(\ell) \geq s$; tali sottoinsiemi di $\mathcal{I}$ vengono chiamati *itemset frequenti* e costituiscono l'insieme $\mathcal{L}$.

2. Per ogni $\ell \in \mathcal{L}$ vengono determinati tutti gli insiemi Y tali che $\mathrm{Sup}\,(\ell)\,/\mathrm{Sup}\,(\ell - Y) \geq c$; $\ell - Y \Rightarrow Y$ è chiaramente una association rule valida.

La fase computazionalmente più onerosa è la prima*. In prima istanza, il numero degli itemset frequenti coincide con il numero dei possibili itemset, che sono $2^{|\mathcal{I}|}$; per ciascun candidato, inoltre, la determinazione

*Un algoritmo efficiente per la seconda fase è consultabile in [4].

del relativo supporto richiede una scansione dell'intero database, che tipicamente è così grande che solo una piccola porzione di esso può essere contemporaneamente presente in memoria centrale. Queste considerazioni mostrano che un approccio di forza bruta al problema è improponibile: non è quindi sorprendente che vari lavori in letteratura abbiano indagato approcci alternativi. In questo term paper, nella Sezione 2 verrà descritto un approccio ormai divenuto standard al problema , basato sulla generazione di itemset "candidati"; nella Sezione 3, invece, si illustrerà una strategia innovativa recentemente proposta in [6].

2 Ricerca di itemset frequenti: generazione dei candidati

Un approccio alla ricerca di itemset frequenti che ha trovato notevole riscontro in letteratura è quello originariamente proposto in [3]; esso prevede l'esecuzione ripetuta dei passi che seguono.

1. A partire dagli itemset frequenti che sono già noti individua nuovi itemset che, sulla base di un opportuno criterio, sono anch'essi presumibilmente frequenti: tali itemset prendono il nome di *candidati*.

2. Sulla base di considerazioni sulla struttura del problema, individua il maggior numero possibile di candidati che non possono essere frequenti ed eliminali.

3. Calcola $\mathrm{Sup}\,(x)$ per ogni candidato x che abbia superato il filtro di cui al punto 2. Aggiungi all'insieme degli itemset frequenti tutti i candidati il cui supporto è almeno pari a s.

4. Se il punto 3 ha determinato almeno un nuovo itemset frequente allora torna al punto 1, altrimenti interrompi l'esecuzione.

La scelta dei criteri di generazione (punto 1) e filtraggio (punto 2) è frutto della mediazione tra due esigenze parzialmente contrastanti: da un lato, ogni nuova scansione di $\mathcal{D}$ richiede costosi accessi in memoria secondaria e dunque si vorrebbe far terminare l'algoritmo nel minimo numero possibile di iterazioni, generando molti candidati (anche di scarsa qualità) per ciascuna iterazione; dall'altro, se i candidati sono davvero molti allora bisogna ricorrere alla memoria secondaria per memorizzarli e ciò danneggia le prestazioni dell'algoritmo. Bisogna pertanto filtrare i candidati in modo che alla fase di calcolo del supporto giungano solo quelli che hanno buone probabilità di "successo".

Il primo algoritmo di generazione dei candidati è stato proposto da Agrawal et al. in [3], ed è basato sul calcolo del supporto aspettato per i candidati. Sia $X \subset \mathcal{I}$ un itemset che è già stato determinato essere frequente (in particolare, ciò significa che il suo supporto $\mathrm{Sup}\,(X)$ è già stato calcolato ed è noto), e sia $t \in \mathcal{D}$ una transazione tale che $X \subset t$: t consente di ottenere nuovi candidati estendendo X con item di $t - X$. Sia $Y \subseteq (t - X)$ un insieme di tali item: ebbene, il *supporto aspettato* $\bar{s}_{X \cup Y}$ per l'itemset $X \cup Y$ è il prodotto di $\mathrm{Sup}\,(X)$ per i supporti delle singole item in Y. $X \cup Y$ è un candidato solo se $\bar{s}_{X \cup Y} \geq s$.

Successivamente, lo stesso Agrawal e Ramakrishnan Srikant hanno proposto una nuova tecnica di generazione dei candidati [5] che è diventata estremamente popolare con il nome di *strategia a priori*. Questo nome è dovuto alla tecnica di filtraggio dei candidati che viene adottata, imperniata sulla seguente osservazione: se un itemset ℓ di taglia k non è frequente, allora nessuno degli itemset di taglia $k + 1, k + 2, \ldots$ che contengono ℓ può a sua volta essere frequente. Di conseguenza, condizione necessaria perché un itemset sia frequente è che tutti i suoi sottoinsiemi propri siano pure frequenti; tutti i candidati che non rispettano questa condizione possono essere eliminati a priori, cioè senza consultare l'insieme $\mathcal{D}$ delle transizioni. La generazione dei candidati è organizzata in modo che la condizione possa essere facilmente verificata: vengono infatti generati candidati di taglia k solo quando sono già noti gli insiemi $F_1, F_2, \ldots, F_{k-1}$ degli itemset frequenti di cardinalità $1, 2, \ldots, k-1$. Per la precisione, l'insieme C_k dei candidati di taglia k viene generato all'iterazione k a partire dall'insieme F_{k-1} secondo l'algoritmo APRIORI-GEN che segue.

APRIORI-GEN(L_{k-1})
  sia $\ell \in L_{k-1}$
  sia $\ell[i]$ l'i-esimo item in L_{k-1} secondo un ordinamento arbitrario
  $C_k \leftarrow \emptyset$
  for each $\ell', \ell'' \in L_{k-1}$ such that ($\ell'[i] = \ell''[i], 1 \leq i < k-1$ and $\ell'[k-1] \neq \ell''[k-1]$) **do**
    aggiungi l'itemset $\ell'[1]\ell'[2]\cdots\ell'[k-1]\ell''[k-1]$ a C_k
  /* Filtraggio dei candidati generati */
  for each $\ell \in C_k$ **do**
    if $\exists \ell' \subset \ell$ such that ($|\ell'| = k-1$ and $\ell' \notin L_{k-1}$) **then** cancella ℓ da L_k
end

Una volta che C_k è stato costruito, è necessario calcolare il supporto di ogni elemento in esso contenuto per determinare quali candidati sono effettivamente itemset frequenti. Per far ciò, [5] propone due distinte strategie.

- Nella prima strategia, il supporto dei candidati viene determinato con una scansione dell'intero insieme $\mathcal{D}$: a ogni candidato viene associato un contatore, e per ogni transizione $t \in \mathcal{D}$ esaminata vengono incrementati di una unità i contatori dei candidati che sono sottoinsiemi di t. Per rendere più efficiente l'accesso ai candidati, che potrebbero essere molti e quindi anche memorizzati su disco, questi sono organizzati tramite funzioni hash in una struttura ad albero che prende il nome di *hash tree* (si consulti [5] per maggiori dettagli).

- Nella seconda strategia, i candidati sono memorizzati in una struttura dati che permette di calcolare il loro supporto senza consultare $\mathcal{D}$. Più precisamente, tale struttura dati è un insieme $\bar{C}_k$ costituito da un elemento $(t, \{X_k(t)\})$ per ogni transizione $t \in \mathcal{D}$ che contenga almeno un candidato: infatti $X_k(t) = \{c \in C_k \mid c \subseteq t\}$. Si noti che, se i candidati sono molti e presenti in molte transizioni, $\bar{C}_k$ può anche essere più grande di $\mathcal{D}$. $\bar{C}_k$ viene costruito a partire da C_k e $\bar{C}_{k-1}$.

L'algoritmo che utilizza la prima strategia viene indicato dagli autori con il nome di `Apriori`, mentre quello che utilizza la seconda viene chiamato `AprioriTid`. Prove sperimentali condotte dagli autori mostrano che `Apriori` è più veloce di `AprioriTid` durante le prime iterazioni (k prossimo a 0), mentre la situazione si capovolge per le ultime iterazioni: ciò è dovuto al fatto che per k elevato i candidati sono solitamente pochi e $\bar{C}_k$ è piccolo, pertanto una scansione di $\bar{C}_k$ richiede molto meno tempo di una scansione di $\mathcal{D}$. Questi risultati sperimentali portano all'introduzione di un algoritmo ibrido (chiamato `AprioriHybrid`) che utilizza `Apriori` per le prime iterazioni e passa poi a `AprioriTid` quando sono verificate opportune condizioni sulla taglia di $\bar{C}_k$.

Ulteriori strategie per la determinazione degli itemset frequenti che fanno ricorso alla generazione di candidati sono proposte in [7], [8] e [9].

3 Ricerca di itemset frequenti: un nuovo approccio

La strategia a priori è efficace nel ridurre il numero di candidati da esaminare rispetto ad un approccio di forza bruta e rende così trattabile il problema delle association rules. Tuttavia, questa strategia mostra i suoi limiti se gli itemset frequenti sono molti oppure se il numero di oggetti in ciascun itemset è elevato.

- Nel primo caso, la generazione dei candidati di dimensione $k+1$ a partire dagli itemset frequenti di dimensione k produce comunque molti itemset, con la possibilità che una parte di essi debba essere trasferita in memoria secondaria. Ad esempio, se il numero di itemset frequenti di taglia 1 è 10^4 allora il numero di candidati di taglia 2 prodotti dalla strategia a priori è superiore a 10^7.

- Nel secondo caso, per scoprire un itemset di taglia elevata è necessario prima esaminare tutti gli itemset di taglia inferiore, il che porta di nuovo alla generazione di un numero elevato di candidati. Ad esempio, determinare un itemset frequente ℓ di taglia 100 richiede $k = 100$ iterazioni dell'algoritmo a priori e la produzione di almeno 2^{100} candidati (tutti i sottoinsiemi di ℓ sono anch'essi itemset frequenti).

In entrambi i casi, appare evidente che nella strategia a priori ciò che limita le prestazioni è il processo di generazione dei candidati: per questa ragione, alcuni ricercatori [6] hanno proposto una strategia alternativa, radicalmente diversa da quella a priori, in grado di determinare gli itemset frequenti *senza* generare alcun candidato. Tale strategia si basa su una struttura dati innovativa chiamata *albero dei pattern frequenti*, o per brevità FP-tree.

3.1 FP-tree: struttura e costruzione

Nell' insieme $\mathcal{I}$ si può istituire un ordinamento totale F dato dalla frequenza di ciascun item in $\mathcal{D}$: in altre parole, per ogni coppia $i_a, i_b \in \mathcal{I}$ si ha che $i_a <_F i_b$ se i_a appare più frequentemente di i_b nel database delle transizioni $\mathcal{D}$. Nell'esposizione che segue chiameremo *pattern* un itemset $\ell \subseteq \mathcal{I}$ i cui oggetti sono ordinati in ordine crescente secondo F.

Un FP-tree è fondamentalmente un albero dei suffissi costruito su stringhe ottenute ordinando gli item in ciascuna transazione $t \in \mathcal{D}$; tale struttura dati, comunque, differisce da un semplice albero dei suffissi per alcuni dettagli implementativi che la adattano alla risoluzione del problema sotto esame. La struttura dell'FP-tree è suggerita dalle seguenti osservazioni di natura euristica.

1. Non è necessario considerare tutti gli item in ciascuna transazione: infatti, solo gli item frequenti (cioè quelli il cui supporto è almeno pari a s) compariranno negli itemset frequenti. Di conseguenza, nella costruzione della stringa corrispondente a una transizione t tutti gli item non frequenti possono essere eliminati. Questa osservazione, che si rifà al principio della strategia a priori, mira a ridurre le dimensioni dell'FP-tree.

2. Ordinare secondo F gli oggetti in ciascuna transazione $t \in \mathcal{D}$ aumenta la probabilità che due transazioni presentino un prefisso comune, e quindi diminuisce potenzialmente il numero di nodi nell'FP-tree.

3. Se k transizioni condividono un prefisso comune p , questo fatto può essere memorizzato incrementando di k un contatore posto in tutti i nodi del cammino associato a p.

Sulla base di queste osservazioni, un albero dei pattern frequenti (o FP-tree) viene definito come una coppia (T, H) in cui

- $T = (V, E)$ è un albero. La radice di T è etichettata con "NULL"; ogni altro nodo è etichettato con una terna (ℓ, k, v) con $\ell \in \mathcal{I}$, k intero non negativo, $v \in V$ tale che la prima componente dell'etichetta di v è ancora ℓ.

- H è una tabella (*header table*). Ogni riga della tabella è una coppia (ℓ, v) con $\ell \in \mathcal{I}$, $v \in V$ tale che la prima componente dell'etichetta di v è ancora ℓ. Le righe della tabella sono ordinate secondo F in ordine crescente; $\forall \ell \in \mathcal{I}$ che compare come prima componente dell'etichetta di un nodo di T esiste una e una sola riga nella tabella.

Si noti che, complessivamente, la riga della header table con prima componente ℓ e i nodi di T con prima componente ℓ costituiscono una lista concatenata. Come d'uso, [6] assume implicitamente che in una

realizzazione pratica dell'FP-tree ciascun nodo contenga ulteriori puntatori ai propri figli (se essi esistono) nonché al proprio padre (se esso esiste).

Dati un insieme di transazioni $\mathcal{D}$ e un supporto minimo s, l'FP-tree associato agli item frequenti in $\mathcal{D}$ può essere costruito mediante l'algoritmo che segue.

1. Scandendo sequenzialmente $\mathcal{D}$, costruisci una lista ordinata L degli item il cui supporto è almeno pari a s.

2. Crea la radice dell'FP-tree ed etichettala con "NULL". Crea una header table vuota in cui la prima colonna è uguale a L (in altre parole, l'elemento sulla riga i è uguale all'i-esima componente di L) e tutti gli elementi della seconda colonna sono impostati a "NULL".

3. Esamina sequenzialmente le transazioni di $\mathcal{D}$. Per ogni transazione $t \in \mathcal{D}$ costruisci la stringa $s(t)$ data dalla sequenza ordinata degli item comuni a t e a L. Sia $s(t) = s_1(t)s_2(t)$, dove $s1(t)$ è il più lungo prefisso di $s(t)$ a cui nell'albero è già associato un cammino che parte dalla radice[†]: inserisci $s(t)$ nell'FP-tree incrementando di 1 il contatore k nei nodi corrispondenti a $s_1(t)$, e aggiungendo nuovi nodi (con contatore inizializzato a 1) per il suffisso $s_2(t)$. Aggiorna di conseguenza la header table e le liste di puntatori nei nodi dell'albero[‡].

3.2 Ricerca degli itemset frequenti tramite FP-tree

L'FP-tree rende logico cercare itemset frequenti i cui elementi sono ordinati secondo F: come già spiegato, tali itemset prendono il nome di pattern. Nel seguito, quindi, si affronterà il problema della ricerca di pattern frequenti.

Innanzitutto, è importante osservare che l'FP-tree contiene tutta l'informazione necessaria alla determinazione dei pattern frequenti. Come si è visto nella Sezione 3.1, durante la costruzione dell'albero ogni transazione $t \in \mathcal{D}$ contribuisce con un pattern composto da soli item frequenti e di taglia *massima*: tale pattern può essere recuperato con un'opportuna visita dell'albero a partire dalla radice. Di conseguenza, ogni pattern composto da item frequenti che compaia in almeno una transizione è presente nell'FP-tree; poiché condizione necessaria perché un pattern sia frequente è che sia composto da sole item frequenti, si conclude. Da sola, però, questa proprietà non garantisce che qualunque strategia di ricerca che faccia uso dell'FP-tree sia automaticamente efficiente: una strategia che usi l'albero solo nella fase di validazione dei candidati, infatti, può ancora generare un numero esponenziale di candidati! Per questa ragione, [6] adotta una strategia radicalmente diversa. L'algoritmo proposto è ancora diviso in fasi, ma le fasi non coincidono più con l'identificazione di pattern frequenti di taglia via via più grande: questa volta, ciascuna fase identifica pattern associati ad un item diverso della header table H, partendo dagli item meno frequenti. L'item associato all'ultima riga di H, che per semplicità chiameremo $j_{|L|}$, viene esaminato per primo: vengono così determinati tutti i pattern frequenti che contengono $j_{|L|}$.Successivamente si passa all'esame di $j_{|L|-1}$, e con esso alla determinazione dei pattern frequenti che contengono $j_{|L|-1}$ ma non $j_{|L|}$, e così via. L'insieme contenente tutti i pattern trovati costituisce la soluzione del problema. L'algoritmo è di tipo ricorsivo: ciascuna fase può richiedere la creazione (con relativo mining) di un FP-tree più piccolo costruito su un sottoinsieme dei pattern dell'albero originario. In dettaglio, la fase i dell'algoritmo, $0 \leq i < L$, richiede l'esecuzione dei passi che seguono.

1. Usando la riga $h = |L| - i$ della header table, determina l'itemset $j_h \in \mathcal{I}$ associato alla fase corrente e tutti i cammini dalla radice di T che contengono j_h; elimina da tali cammini tutti gli itemset meno frequenti di j_h. Si ricordi che tutti i nodi di T con prima componente dell'etichetta j_h sono contenuti

[†]Tale prefisso può essere determinato mediante una visita dell'albero.

[‡]I dettagli di queste operazioni, che non sembrano computazionalmente trascurabili, *non* sono descritti in [6].

in una lista concatenata che parte dalla riga h di H, e che i nodi in ogni cammino di T sono ordinati per frequenze decrescenti: di conseguenza, per determinare i cammini voluti basta percorrere la lista concatenata, e da ogni nodo della lista risalire fino alla radice di T.

2. Sia $C_h = \{c_1^h, c_2^h, \ldots\}$ l'insieme dei cammini determinati al punto precedente: a ogni cammino c_a^h è associato un pattern p_a^h, cioè una sequenza ordinata di item. Il supporto del pattern p_a^h è dato dalla seconda componente k_a^h dell'etichetta dell'ultimo nodo lungo il cammino, cioè del nodo associato all'item j_h (è facile verificarlo ricordando che i nodi precedenti hanno supporto $\geq k$, dove i nodi con supporto strettamente maggiore corrispondono a pattern che "abbandonano" il cammino). Sia $P_h = \{(p_1^h, k_1^h), (p_2^h, k_2^h), \ldots\}$ l'insieme dei pattern con i relativi supporti.

3. Elimina j_h dai pattern in P_h (ottenendo così P_h') e costruisci un FP-tree ausiliario (T', H') sull'insieme così ottenuto. Questo implica: il calcolo di un nuovo supporto per ciascun item ora presente in P_h, ottenuto come rapporto tra il numero di occorrenze in P_h e il numero di occorrenze nell'*intero* insieme delle transazioni $\mathcal{D}$; l'eliminazione degli item il cui supporto è ora inferiore a s; la definizione di un ordinamento F' basato sui nuovi supporti.

4. Esamina (T', H'). Se T' è vuoto, allora nessun pattern frequente è associato a j_h. Se T' è costituito da un unico cammino, allora i pattern frequenti sono dati dai pattern associati a tutti i sottoinsiemi dei nodi del cammino[§] più j_h stesso. Se infine T' è non vuoto e almeno un nodo ha cardinalità maggiore di 1, allora invoca ricorsivamente l'algoritmo di pattern mining su (T', H'): i pattern frequenti sono ottenuti concatenando j_h ai pattern restituiti da tale chiamata ricorsiva.

La terminazione della fase i dell'algoritmo è assicurata dal fatto che T' ha un numero di nodi strettamente inferiore a T perché sono stati rimossi almeno i nodi corrispondenti all'item j_h. La correttezza dell'algoritmo è assicurata dalle due proprietà che seguono.

- **Fragment Growth.** Il numero di occorrenze di un itemset β in P_h' è uguale al numero di occorrenze dell'itemset $\beta \cup j_k$ in $\mathcal{D}$.

- **Prefix Path Property.** Per calcolare tutti i pattern frequenti contenenti gli item $L_1, L_2, \ldots, L_h$ è sufficiente considerare i cammini in T dalla radice fino ai nodi associati all'h-esima riga della header table H.

La prima proprietà assicura che i pattern restituiti da una fase dell'algoritmo siano effettivamente frequenti, e la seconda permette di dimostrare che l'algoritmo nel suo complesso restituisce tutti e soli i pattern frequenti.

3.3 Analisi delle prestazioni

Il lavoro di Han et al. non contiene alcuna analisi formale dell'algoritmo proposto: ci si limiterà quindi a compiere alcune osservazioni. La costruzione dell'FP-tree (T, H) richiede due scansioni del database $\mathcal{D}$ delle transazioni, più un consistente numero di accessi alla lista L delle transazioni frequenti, all'albero T in costruzione e alla header table H. Anche adottando come modello di costo l'*I/O model* [2, 10], il che permette di trascurare tutta la computazione in memoria centrale, è impossibile fare una stima del tempo di esecuzione senza sapere se L, T e H sono sufficientemente piccoli da poter essere contenuti in memoria centrale. Le uniche osservazioni, piuttosto banali, fatte in [6] sulle dimensioni dell'FP-tree sono le seguenti.

[§]Se l'albero contiene un unico cammino, gli item ad esso associati compaiono insieme un numero di volte pari al supporto dell'ultimo item nel cammino, supporto che per costruzione è almeno pari a s.

- L'altezza di T è limitata superiormente dal massimo numero di item frequenti in una transazione $t \in \mathcal{D}$ (ci sembra si possa affermare che l'altezza è necessariamente *uguale* a tale numero, a meno di una costante).

- Il numero di nodi in T è limitato superiormente dal numero *totale* di occorrenze degli item frequenti in tutte le transazioni del database $\mathcal{D}$.

Sulla scorta di queste osservazioni, [6] conclude che l'FP-tree (T, H) non è asintoticamente più grande dell'insieme $\mathcal{D}$ sul quale è stato costruito. Questo bound, per quanto debole, è sicuramente interessante in quanto la strategia a priori può invece richiedere una quantità esponenziale di spazio per la generazione dei candidati, ma non è sufficiente per stabilire se l'FP-tree è memorizzabile in memoria centrale; a creare maggior incertezza, in [6] si sostiene che il bound sul numero di nodi è estremamente pessimistico, e che in pratica l'FP-tree risulta una struttura compatta e "notevolmente più piccola" dell'insieme $\mathcal{D}$ di partenza (da 20 a 100 volte più piccolo, secondo [6]). Se comunque (T, H) e L sono contenuti in memoria centrale, allora il tempo di esecuzione è intuitivamente "buono" in quanto l'unico costo computazionale è quello dovuto agli accessi a $\mathcal{D}$. Per i casi in cui questa condizione favorevole non è verificata, [6] suggerisce due possibili approcci: partizionare l'insieme $\mathcal{D}$ in modo che l'FP-tree costruito su ciascuna partizione sia contenuto in memoria centrale, oppure memorizzare l'FP-tree su disco tramite una struttura dati (ad esempio un B+-tree) che consenta una accesso efficiente. Entrambe le strategie, comunque, non sono descritte in dettaglio e non sono neppure oggetto di analisi sperimentale.

Il tempo per la determinazione dei pattern frequenti, ancora una volta, è oggetto in [6] di sole considerazioni euristiche. Gli autori osservano che l'algoritmo da essi proposto non richiede alcun ulteriore accesso al database delle transizioni $\mathcal{D}$: ciascuna fase richiede una semplice scansione dell'FP-tree per la determinazione di C_h, e ci aspetta che tale FP-tree sia assai più piccolo di $\mathcal{D}$. Oltre a ciò, ciascuna fase può richiedere la costruzione e il mining ricorsivo di un nuovo FP-tree costruito a partire da P_h: tale insieme, però, contiene tipicamente solo una piccola frazione dei pattern associati all'FP-tree originario, quindi la chiamata ricorsiva si conclude in genere "velocemente". Infine, gli autori osservano che l'algoritmo non include alcun procedimento potenzialmente esponenziale di generazione di candidati: la taglia dell'output è esponenziale solo se il numero di pattern frequenti è effettivamente esponenziale. La bontà della strategia di mining proposta viene messa alla prova con un'analisi sperimentale. Per la precisione, gli autori confrontano il proprio algoritmo `FP-growth` con quello a priori di [5] e con un algoritmo proposto in tempi recenti e chiamato `TreeProjection` [1]; non è chiaro quale variante dell'algoritmo a priori venga adottata tra `Apriori`, `AprioriTid` e `AprioriHybrid`. I tre algoritmi sotto indagine sono stati implementati dagli autori ("al meglio della loro conoscenza") su un PC dotato di processore Intel a 450 MHz e 128 MB di RAM; gli algoritmi sono stati poi collaudati sul medesimo sistema ricorrendo a due dataset generati artificialmente secondo la procedura descritta in [5]:

- $\mathcal{D}_1$ contiene 10^4 transizioni costruite su $|\mathcal{I}_1| = 10^3$ item. Il numero medio di item per transizione è 25, e la lunghezza media di un itemset frequente è 10;

- $\mathcal{D}_2$ contiene 10^5 transizioni costruite su $|\mathcal{I}_2| = 10^4$ item. Il numero medio di item per transizione è 25, e la lunghezza media di un itemset frequente è 20.

La dimensione media di un item non è specificata: non è quindi possibile stabilire quanta memoria occupino $\mathcal{D}_1$ e $\mathcal{D}_2$. Gli autori, comunque, ammettono che entrambi i dataset sono sufficientemente piccoli da risiedere interamente in memoria centrale.

Gli algoritmi vengono confrontati facendo variare il minimo supporto s dallo 0.1% al 3%, e osservando i tempi di esecuzione; i tempi per `FP-growth` includono naturalmente anche la costruzione dell'FP-tree. Tutti e tre gli algoritmi mostrano una crescita molto ripida del tempo di esecuzione al decrescere di s: anche

se non viene esplicitamente detto, tale crescita è probabilmente esponenziale dato che gli autori affermano che in $\mathcal{D}_1$ e $\mathcal{D}_2$ il numero di itemset frequenti cresce esponenzialmente al diminuire di s. L'algoritmo più veloce è `FP-growth`, seguito da `TreeProjection` e infine dall'algoritmo a priori.

In una seconda batteria di esperimenti, il numero di transazioni in $\mathcal{D}_2$ viene fatto variare da 10^4 fino a 10^5 mantenendo costanti gli altri parametri: tutti e tre gli algoritmi mostrano una crescita lineare del tempo di esecuzione con il numero delle transazioni. Ancora una volta `FP-growth` sembra l'algoritmo più veloce: non si può tuttavia fare a meno di notare che `FP-growth` viene confrontato con l'algoritmo a priori impostando il supporto minimo s all'1.5%, mentre viene confrontato con `TreeProjection` ricorrendo a un supporto dell'1%.

4 Considerazioni finali

In questo term paper si è definito il problema della ricerca di association rules, concentrandosi poi sul sottoproblema della ricerca di itemset frequenti; per tale sottoproblema si è descritto l'approccio più citato in letteratura, fondato sulla generazione di itemset candidati, e una strategia alternativa suggerita recentemente in [6] e basata su una nuova struttura dati chiamata FP-tree.

Il lavoro [6] manifesta una certa carenza nell'analisi teorica delle prestazioni della strategia proposta, carenza che è comunque comune ad altri studi nello stesso ambito. Anche l'analisi sperimentale, tuttavia, non è qui esente da critiche: l'impressione è che essa utilizzi dataset di dimensioni troppo ridotte. A sostegno di questa tesi bisogna ricordare che non solo i dataset utilizzati sono interamente contenuti in memoria centrale, ipotesi non verificata dalle basi di dati del "mondo reale", ma anche i tempi di esecuzione sono sempre estremamente ridotti (pochi minuti al massimo) nonostante la limitata potenza dell'hardware a disposizione. Di conseguenza, sarebbe auspicabile un'analisi sperimentale su dataset di dimensioni decisamente maggiori in modo da verificare il comportamento dell'algoritmo in presenza di una gerarchia memoria/dischi.

Riferimenti bibliografici

[1] Ramesh C. Agarwal, Charu C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, 2001.

[2] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining associations between sets of items in massive databases. In *Proceedings of the ACM-SIGMOD 1993 International Conference on Management of Data*, pages 207–216, Washington D.C., may 1993.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. Technical Report TR-RJ 9834, IBM Almaden Research Center, 1994.

[5] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th Very Large Data Base Conference*, pages 487–499, 1994.

[6] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the ACM-SIGMOD 2000 International Conference on Management of Data*, pages 1–12, Dallas, TX USA, 2000.

[7] Maurice Houtsma and Arun Swami. Set-oriented mining of association rules. Technical Report RJ 9567, IBM Almaden Research Center, San Jose, CA, USA, 1993.

[8] Heikki Mannila, Hannu Toivonen, and Inkeri Verkamo. Efficient algorithms for discovering association rules. In *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, pages 181–192, Seattle, WA, USA, jul 1994.

[9] Ashoka Savasere, Edward Oniecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of 21st International Conference on Very Large Data Bases*, pages 432–444, Zurich, Switzerland, sep 1995. Morgan Kaufmann.

[10] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.