

DIPARTIMENTO DI INGEGNERIA DELL’INFORMAZIONE

Corso di

Algoritmi e Strutture Dati

Relazione di progetto:

Calcolo dei Minimal Hitting Set

Docente: Prof.ssa Marina Zanella

Esaminando:  
Edoardo Coppola

Matricola n. 719599

Anno Accademico 2020/2021

*Introduzione*

Questa relazione ha come scopo la documentazione delle fasi di sviluppo di un’applicazione software per la generazione di tutti e soli i *minimal hitting set* dato un dominio e una collezione di sottoinsiemi dello stesso. Nelle sezioni successive verranno illustrate le scelte progettuali relative alle strutture dati adottate, gli algoritmi alla base del funzionamento dei diversi moduli impiegati (di seguito indicati in grassetto), le sperimentazioni effettuate e i risultati ottenuti.

L’applicazione realizzata tiene conto dell’*alternativa b* illustrata all’interno delle specifiche progettuali: dotare il programma di una funzionalità di pre-elaborazione. Questa sarà discussa in una sezione dedicata.

Al termine di questo documento saranno riportate anche alcune semplici indicazioni per l’utilizzo dell’applicativo.

Il linguaggio di programmazione che si è scelto per la realizzazione è *Python 3.6* e l’ambiente di sviluppo è *PyCharm*, sebbene porzioni di codice siano state scritte da riga di comando.

1. *La forma e la lettura dell’input*

L’algoritmo proposto nelle specifiche, detto *MBase*, prevede in ingresso una matrice A i cui elementi sono ‘1’ o ‘0’. Tale matrice ha tante colonne quanti sono gli elementi del dominio M, e tante righe quanti sono gli insiemi della collezione N. Va sottolineato che M gode di un ordinamento totale lessicografico che consente di calcolare l’elemento minimo e massimo entro il dominio. Analogamente, è sempre possibile individuare il successore o il predecessore di un dato elemento. La matrice A viene riportata all’interno di specifici file *.matrix* il cui contenuto è illustrato in figura 1.

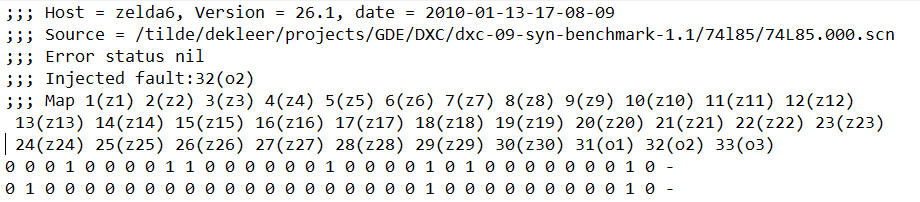


Figura – Contenuto di un file .matrix

Le righe che iniziano con ‘;;;’ rappresentano dei commenti. Tra queste, l’ultima è la più importante perché riporta gli elementi del dominio M e i loro identificativi. Ad esempio, ‘*1(z1)’* significa che ‘*z1’* è un elemento del dominio e ‘*1’* è il suo identificativo. Le ultime righe di questi file rappresentano la vera e propria matrice A in forma binaria. Dalla figura 1 possiamo notare che, in questo caso, gli insiemi della collezione N siano due mentre gli elementi del dominio M siano trentatré. All’interno di A, se una generica cella aij contiene un ‘1’ significa che l’insieme Ni annovera il j-esimo elemento di M. Si può notare che a ciascun elemento del dominio corrisponde una precisa colonna di A e le stesse seguono l’ordinamento lessicografico presente in M. Ad esempio, la prima colonna della matrice specifica per il primo elemento di M, ossia *z1*.

Per quanto riguarda la lettura e l’interpretazione del contenuto di questi file, ci si è affidati alla funzione **getMatrixFromFile(filename)** presente e documentata all’interno del codice. In un primo momento vengono lette tutte le righe presenti nel file e vengono scartate quelle riportanti un commento. Arrivati al contenuto della futura matrice A, ne vengono lette le righe, gli spazi separatori vengono sostituiti con delle virgole, quindi tali stringhe vengono convertite in vettori con i quali costruire una matrice vera e propria.

Per quanto riguarda la struttura dati impiegata per contenere A, si è scelto di evitare una lista di liste per cercare di risparmiare spazio. Infatti, in *Python* le liste sono strutture dati dinamiche e capaci di contenere dati eterogenei. Per queste ragioni lo spazio che viene dedicato loro in memoria è sovrabbondante per i nostri scopi. Si è scelto quindi di utilizzare gli *array* della libreria *numpy* il cui contenuto è immutabile e fortemente tipizzato. Questo consente di risparmiare spazio in memoria e offre la possibilità di usufruire di tantissime funzione di utilità all’interno dei *package* della libreria. Tali funzioni sono ottimizzate e consentono quindi di ottenere prestazioni migliori anche per quel che concerne il tempo di calcolo.

2. *L’algoritmo MBase*

La prima versione realizzata di MBase prevedeva come parametro in ingresso solamente la matrice A vista nella sezione precedente, mentre la versione successiva, nonché definitiva, ha previsto anche parametri aggiuntivi e facoltativi. Difatti, Python offre la possibilità di scrivere funzioni i cui parametri formali siano in grado anche di assumere valori di default se non specificato altrimenti al momento delle chiamate. La firma della funzione appare quindi in questo modo: **mbase(A, timeEnabled=True, mapping=None)**. Il parametro *timeEnabled* è booleano e consente, quando assume il valore *True,* di riportare il tempo impiegato dall’algoritmo per il calcolo di tutti i *mhs*. Il secondo parametro aggiuntivo, chiamato *mapping*, verrà descritto nella sezione dedicata alla pre-elaborazione. All’interno di *Mbase*, oltre a quanto appare già nello pseudo-codice fornito dalle specifiche, avviene anche il calcolo di una matrice detta *singletonRepresentativeMatrix* per mezzo di **getSingletonRepresentativeMatrix(A)**. Quest’ultima matrice, chiamata per comodità S, racchiude nelle proprie colonne i vettori rappresentativi dei sottoinsiemi singoletto di M. La costruzione di S avviene sfruttando direttamente la matrice A e gli identificativi degli elementi di M. La figura 2 riassume in forma pittorica il contenuto della funzione sopracitata.



Figura – Costruzione dei vettori rappresentativi degli insiemi singoletto in forma matriciale



Possiamo osservare che, data in ingresso la matrice A, la matrice S si ottiene applicando una banale trasformazione: se una cella aij = 1 allora sij = j per j=1,2, … |M|. Il calcolo di S avviene una sola volta in tutta l’esecuzione di *MBase* e la matrice stessa viene poi passata come parametro al metodo **check(lambda, S)** che in questo modo evita di doverla continuamente ricalcolare.

Quest’ultima funzione, che controlla che un dato sottoinsieme di M, detto *lambda*, sia o meno un *mhs*, utilizza due funzioni ausiliarie per il calcolo del vettore rappresentativo associato a *lambda* e per la costruzione della proiezione di tale vettore sull’insieme in esame. **Build\_representativeVector(lambda, S)** restituisce un vettore nullo avente dimensione (1 x | *lambda* |) se *lambda* = Ø, il vettore rappresentativo dell’unico elemento di *lambda* se lo stesso ha cardinalità unitaria oppure un vettore rappresentativo opportunamente calcolato se la cardinalità è maggiore o uguale a due. La funzione **combine\_columns(S[: lamda])** assolve a quest’ultimo caso seguendo la regola più generale per la costruzione dei vettori rappresentativi (riportata nelle specifiche progettuali). Infine, la funzione **build\_projection(lambda, representativeVector)** restituisce l’insieme di elementi di *lambda* contenuti in *representativeVector*. Mediante il supporto di queste due funzioni, il corpo di **check** si limita a controllare che la proiezione coincida o meno con *lambda* stesso e, nel caso, che non siano presenti valori nulli all’interno del vettore rappresentativo associato. Se si verificano queste due condizioni, l’esito del controllo è ‘*MHS’*, mentre se si verifica solo la prima ma non la seconda il risultato sarà ‘*OK’*. Tutte le funzioni finora elencate sono ben descritte anche all’interno del codice.

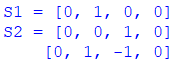
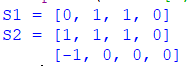
L’ultimo modulo richiamato da *MBase* è **output(lambda, countMHS, mapping)**. Quest’ultimo si occupa della stampa di quegli insiemi lambda che si sono rivelati dei mhs. Il secondo parametro ha il solo scopo di contare quanti *minimal hitting set* sono stati trovati fino a quel momento, mentre il ruolo di *mapping* verrà descritto in modo approfondito nella sezione dedicata alla pre-elaborazione.

Per quanto riguarda le strutture dati utilizzate, si è scelto di utilizzare i cosiddetti *numpy array* per la rappresentazione degli insiemi generati e controllati dall’algoritmo e le motivazioni sono le stesse che hanno portato a scegliere di utilizzare i *numpy array* (o *numpy matrix*) per la matrice A. Inoltre, sebbene *Python* fornisca una struttura dati nativa, detta *Set*, che gode di tutte le vantaggiose proprietà delle *tabelle hash*, non è possibile stabilire un ordinamento su tali strutture. Questo avrebbe rappresentato un grave problema visto che *MBase* lavora generando insiemi secondo un preciso ordinamento lessicografico. Gli insiemi denotati col nome di *lambda* assumono quindi la forma sopracitata e, va sottolineato, lavorano con gli identificativi degli elementi di M e non propriamente con gli elementi stessi. La matrice S, invece, è della stessa natura della matrice A: appartiene alla classe dei *numpy array*. La medesima struttura dati è stata adottata anche per rappresentare i vettori rappresentativi. Difatti, essi si ottengono a partire dalla matrice S quindi è normale che ne condividano le caratteristiche.

3. La pre-elaborazione

3.1 Le funzioni del\_rows e del\_cols

In questa sezione verrà illustrato lo pseudo-codice degli algoritmi **del\_rows(A)** e **del\_cols(A)**, ossia le due operazioni richieste dalla pre-elaborazione illustrata nelle specifiche progettuali. La prima elimina righe della matrice A che specificano per insiemi Ni che sono super-insiemi di altri insiemi Nj (con i ≠ j). Per illustrarne il funzionamento procediamo con alcuni esempi:

*  In questo primo caso consideriamo una riga di A che specifica per l’insieme S1 e una seconda riga che specifica per l’insieme S2. Il primo insieme annovera il primo, il secondo e il terzo elemento di M, mentre il secondo possiede solamente il secondo e il terzo elemento del dominio. Emerge quindi chiaramente come S2 S1. Effettuando una differenza tra i due vettori che rappresentano tali insiemi, troviamo il vettore [1, 0, 0, 0] che assume una forma caratteristica. Infatti, il vettore *differenza* assume il valore ‘1’ nella j-esima posizione se e solo se S1 possiede l’elemento j-esimo di M che a S2 invece manca; se invece la j-esima posizione è occupata da ‘0’ significa che entrambi S1 e S2 possiedono il j-esimo elemento oppure che quest’ultimo non compare in nessuno dei due insiemi. La presenza di un ‘-1’, invece, sta a significare che S2 possiede l’elemento j-esimo di M ma non S1. Quindi, per determinare se un generico insieme è super-insieme di un altro, è sufficiente scansionare il vettore *differenza* (calcolato sottraendo la riga di S2 a quella di S1) e accertarsi che non vi siano ‘-1’. In quel caso, la riga relativa a S1 va eliminata da A.
*  In questo secondo esempio distinguiamo che S1 non è un super-insieme di S2. Infatti, il primo non possiede il quarto elemento di M al contrario del secondo. Il vettore *differenza* riporta quindi un ‘-1’ in quarta posizione ad indicare che non esiste alcun tipo di inclusione tra i due. In questo caso non sarà quindi necessario eliminare alcuna riga da A.
*  In questo terzo esempio possiamo notare che S1 e S2 siano due insiemi disgiunti e che quindi, seguendo lo stesso ragionamento di prima, sia possibile discernere anche questa casistica ed evitare ancora una volta di eliminare righe dalla matrice A.
*  In questo esempio, invece, notiamo che la relazione di inclusione di inclusione insiemistica si è ribaltata: questa volta S1 S2. Seguendo il ragionamento condotto finora, non saremmo in grado di rilevare questa inclusione insiemistica sicché vengono considerati unicamente casi in cui S1 S2 e non viceversa. Eppure, S2 è comunque un super-insieme e, in quanto tale, dovrebbe vedere la propria riga in A eliminata. A tale scopo, è sufficiente scansionare la matrice in ingresso simultaneamente dall’ultima riga verso la prima così da trattare anche casi come quello corrente. Seguendo quest’ultima soluzione però, in caso di righe duplicate, cioè qualora S1 sia uguale a S2, l’indice della riga da eliminare verrebbe inserito due nella lista delle righe da eliminare. È quindi necessaria un’operazione di cancellazione dei duplicati da questo elenco una volta terminata la scansione di A prima di procedere con l’effettiva eliminazione.

Vediamo di seguito lo pseudo-codice della funzione **del\_rows(A)**:

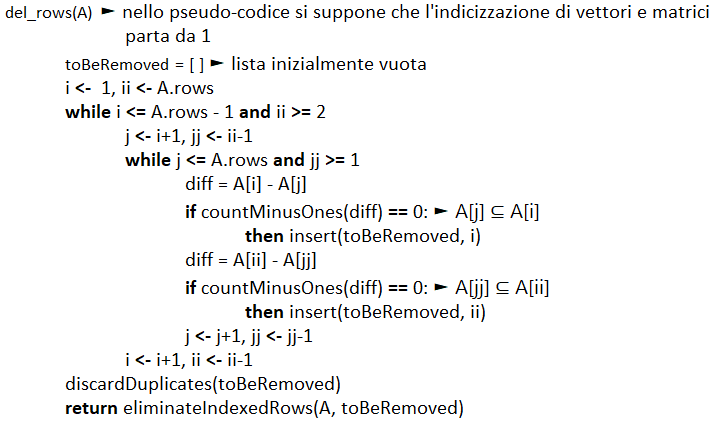


Figura – Pseudo-codice di del\_rows(A)

Un secondo approccio utilizzabile, ma scarsamente efficiente, vede l’utilizzo dei già menzionati *Python Set*. Difatti, sarebbe possibile costruire la matrice S, trasformare ogni riga della stessa in un *set* e verificare, tramite i metodi *isSuperSet* e *isSubSet*, l’eventuale presenza di inclusioni insiemistiche per ogni coppia di insiemi (cioè di righe di S). Questo approccio condurrebbe al medesimo risultato ma sarebbe poco scalabile dal momento che la creazione di ulteriori strutture dati come i *set* richiederebbe spazio aggiuntivo in memoria pari a . Inoltre, sarebbe necessaria una riconversione di S’, ottenuta dopo l’eliminazione di righe e colonne da S, in una nuova matrice A’ da fornire in ingresso a *MBase*. Tutte queste operazioni farebbero solo peggiorare le prestazioni temporali[[1]](#footnote-2).

Per quanto riguarda invece la funzione che gestisce l’eliminazione di colonne nulle, detta **del\_cols(A),** essa si concentra nell’individuare quali colonne siano interamente nulle e nel cancellarle successivamente dalla matrice A ricevuta in ingresso. Non sono necessarie particolari spiegazioni all’infuori di quanto si può dedurre dallo pseudo-codice riportato in figura 4.

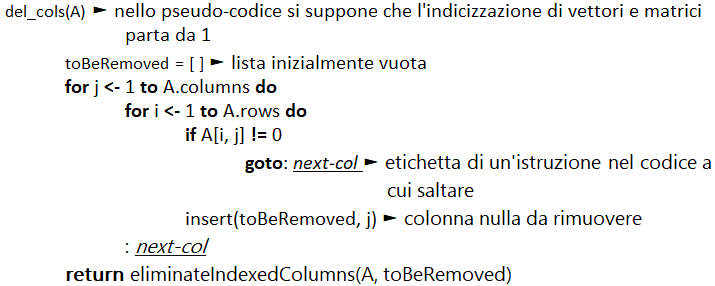


Figura – Pseudo-codice di del\_cols(A)

Una volta terminata la pre-elaborazione, che restituisce una matrice A’ le cui dimensioni sono minori o uguali a quelle di A, si può mandare in esecuzione MBase dando in ingresso quanto ottenuto da questa fase preliminare.

3.2 Il problema del mappaggio delle colonne di A’

Osservando i risultati prodotti dall’esecuzione di **MBase(A’)** si può notare come i tempi di calcolo, misurati grazie al parametro *tiimeEnabled* menzionati in precedenza, siano calati drasticamente ma salta immediatamente all’occhio come siano stati prodotti risultati differenti dai precedenti. Ricercando il motivo di queste discrepanze ci si rende conto che l’algoritmo non è cambiato e che la correttezza dei risultati precedenti deve valere anche per questa seconda esecuzione. Il motivo di tale incongruenza risiede nel fatto che, a causa dell’eliminazione di alcune colonne di A, il generico elemento j ∈ M si trova ad occupare la posizioni diverse dalla j-esima in A’. La figura 5 mostra ciò che succede nel caso in cui |M’|<|M|.



Figura – Cancellazione di una colonna nulla nella costruzione di A’

In questo caso, dove si è trascurata l’eliminazione delle righe per semplicità, la prima colonna, relativa al primo elemento di M, è interamente nulla e viene quindi cancellata. Nella nuova matrice 3x2 il secondo elemento di M, a cui veniva associata la seconda colonna di A, vede adesso associata la prima colonna di A’. Lo stesso vale per il terzo elemento di M che adesso corrisponde alla seconda colonna di A’ quando in precedenza era associato alla terza colonna di A. Questo “spostamento” porta quindi ad una risoluzione del problema degli *hitting set* minimali che produce risultati corretti ma nella rappresentazione di A’. Per effettuare un confronto tra le due esecuzione di *MBase* è quindi necessario mappare i risultati prodotti da *MBase(A’)* su una rappresentazione che tenga conto di tutti gli elementi di M e non solo quelli che colpiscono almeno una collezione di N.

Ecco quindi dove entra in gioco il parametro *mapping* menzionato in precedenza. Questo non è altro che una lista numerica che mappa elementi di M’ in elementi di M. La figura 6 offre un’idea del funzionamento di questo mappaggio.



Figura – Mappaggio dei valori di M’ su M

Seguendo l’esempio in figura 5, il primo elemento di M’ deve essere mappato sul secondo elemento di M, mentre il secondo elemento di M’ deve essere mappato sul terzo elemento di M. In generale, lo spostamento che deve essere calcolato dipende dal numero di colonne che sono state cancellate fra due colonne che invece sono state lasciate inalterate. La funzione **getMaps(indecesRemoved, MprimeLength)** documentata all’interno del codice assolve alla costruzione del vettore *mapping* che viene utilizzato dal modulo *output* in caso si sia compiuta la pre-elaborazione.

4. La sperimentazione

1. La costruzione di S a partire da A richiede un tempo ; il controllo di tutte le coppie di insiemi in S richiede un tempo O(N2) se i super-insiemi vengono eliminati man mano e non una volta terminato il processo; la costruzione di A’ richiede un tempo [↑](#footnote-ref-2)