UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

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

Sommario

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), alcune tra 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 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.

```
;;; Host = zelda6, Version = 26.1, date = 2010-01-13-17-08-09
;;; Source = /tilde/dekleer/projects/GDE/DXC/dxc-09-syn-benchmark-1.1/74l85/74L85.000.scn
;;; Error status nil
;;; Injected fault:32(o2)
;;; Map 1(z1) 2(z2) 3(z3) 4(z4) 5(z5) 6(z6) 7(z7) 8(z8) 9(z9) 10(z10) 11(z11) 12(z12)
 13(z13) 14(z14) 15(z15) 16(z16) 17(z17) 18(z18) 19(z19) 20(z20) 21(z21) 22(z22) 23(z23)
 24(z24) 25(z25) 26(z26) 27(z27) 28(z28) 29(z29) 30(z30) 31(o1) 32(o2) 33(o3)
0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 -
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 -
```

Figura 1 – 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 a_{ij} contiene un '1' significa che l'insieme N_i 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 funzioni 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.

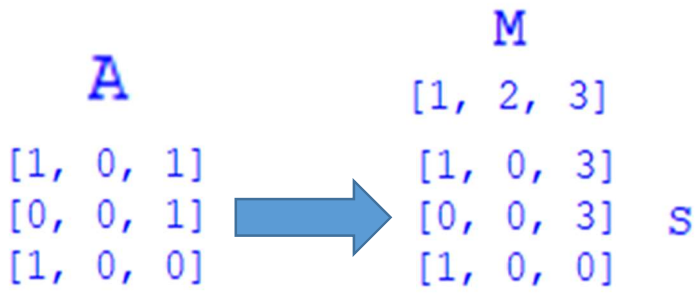

Figura 2 – Costruzione dei vettori rappresentativi degli insiemi singoletto in forma matriciale

Possiamo osservare che, data in ingresso la matrice A, la matrice S si ottiene applicando una banale trasformazione: se una cella $a_{ij} = 1$ allora $s_{ij} = j$ per $j=1,2, \ldots |M|$. 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.

Un'ottimizzazione che è stata inserita nel codice di *MBase* riguarda il calcolo del minimo e del massimo elemento di M, utili all'interno del ciclo più esterno dell'algoritmo. Il dominio viene rappresentato tramite un vettore che è costruito secondo un ordinamento crescente. In questo modo, per ottenere il minimo e il massimo è sufficiente accedere alla prima e all'ultima cella, rispettivamente, del vettore stesso anziché calcolarli ogni volta. L'operazione di accesso richiede un tempo costante contrariamente ai calcoli sopracitati che vengono svolti in tempo lineare.

La funzione *check*, 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 |N| 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 l'ausilio 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. In particolare, **check** abbraccia la seconda alternativa indicata nelle specifiche progettuali e limita i tempi di calcolo evitando di costruire il vettore rappresentativo di lambda a partire da $C_{\emptyset}$. Inoltre, grazie all'impiego degli oggetti *numpy*, sono contenute anche le occupazioni di memoria consentendo quindi l'elaborazione di istanze di dimensioni maggiori rispetto a quelle che sarebbero trattabili se si utilizzassero strutture dati differenti (come le liste native del linguaggio).

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 N_i che sono super-insiemi di altri insiemi N_j (con $i \neq j$). Per illustrarne il funzionamento procediamo con alcuni esempi:

- ```
  S1 = [1, 1, 1, 0]
  S2 = [0, 1, 1, 0]
       [1, 0, 0, 0]
  ```
  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. S1 annovera il primo, il secondo e il terzo elemento di M, mentre S2 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 contrariamente a 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.
  ```

- ```
  S1 = [1, 1, 1, 0]
  S2 = [0, 1, 1, 1]
       [1, 0, 0, -1]
  ```
  In questo secondo esempio distinguiamo che S1 non è un super-insieme di S2. Infatti, S1 non possiede il quarto elemento di M al contrario di S2. Il vettore *differenza* riporta quindi un '-1' in quarta posizione ad indicare che non esiste alcun tipo di inclusione insiemistica tra i due. In questo caso non sarà quindi necessario eliminare alcuna riga da A.

- ```
  S1 = [0, 1, 0, 0]
  S2 = [0, 0, 1, 0]
       [0, 1, -1, 0]
  ```
  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.

- ```
  S1 = [0, 1, 1, 0]
  S2 = [1, 1, 1, 0]
       [-1, 0, 0, 0]
  ```
  In questo esempio, invece, notiamo che la relazione 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 S2 ⊆ S1 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 applicare lo stesso ragionamento anche alla differenza tra la riga di S2 e quella di S1.

La figura 3 mostra lo pseudo-codice della funzione **del_rows(A)**.
  ```

```
del_rows(A)
        i <- 1
        toBeRemoved <- []
        while i <= A.rows - 1
                j <- i+1
                while j <= A.rows
                        diff = A[i] - A[j]
                        diff2 = A[j] - A[i]
                        if countMinusOnes(diff) == 0
                                insert(toBeRemoved, i)
                        else if countMinusOnes(diff2) == 0
                                insert(toBeRemoved, j)
                        j <- j+1
                i <- i+1
        return eliminateIndexedRows(A, toBeRemoved)
```

Figura 3 – Pseudo-codice di del_rows(A)

L'approccio seguito ha una complessità temporale complessiva di $\theta(N^2)$.

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 $\theta(N * M)$[1]. 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[2].

[1] Sarebbe necessario creare un set di cardinalità |M| per ciascuna delle N righe di S.
[2] La costruzione di S a partire da A richiede un tempo $\theta(N * M)$; il controllo di tutte le coppie di insiemi in S richiede un tempo $O(N^2)$ se i super-insiemi vengono eliminati man mano e non una volta terminato il processo; la costruzione di A' a partire da S' richiede un tempo $\theta(N' * M')$

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.

```
del_cols(A) ► nello pseudo-codice si suppone che l'indicizzazione di vettori e matrici
              parta da 1
    toBeRemoved = [ ] ► lista inizialmente vuota
    for j <- 1 to A.columns do
        for i <- 1 to A.rows do
            if A[i, j] != 0
                goto: next-col ► etichetta di un'istruzione nel codice a
                                 cui saltare
            insert(toBeRemoved, j) ► colonna nulla da rimuovere
        : next-col
    return eliminateIndexedColumns(A, toBeRemoved)
```

Figura 4 – 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* fornendo 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 *timeEnabled* menzionato 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 $j \in M$ 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 5 – 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')* in 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.

```
M' values [2, 3]
M values: [1, 2, 3]
```

Figura 6 – 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*

La sperimentazione è stata condotta su un sottoinsieme dei file *.matrix* a disposizione nei *benchmark* e ha coinvolto matrici in ingresso le cui dimensioni arrivavano fino a migliaia di colonne. Alcuni esempi sono riportati nelle figure sottostanti.

```
(|N| = 2, |M| = 33)

MHS found: [22] of dimension 1
MHS encountered : 1

MHS found: [32] of dimension 1
MHS encountered : 2

MHS found: [2 4] of dimension 2
MHS encountered : 3

MHS found: [2 9] of dimension 2
MHS encountered : 4

MHS found: [ 2 10] of dimension 2
MHS encountered : 5

MHS found: [ 2 17] of dimension 2
MHS encountered : 6

MHS found: [ 2 24] of dimension 2
MHS encountered : 7

MBASE required 0.2127 seconds to execute
```

```
74L85.000.matrix
Rows dropped in preprocessing:  []
Columns dropped in preprocessing:  [1, 3, 5, 6, 7, 8, 11, 12, 13,
14, 15, 16, 18, 19, 20, 21, 23, 25, 26, 27, 28, 29, 30, 31, 33]
Preprocessing required 0.012400 seconds to execute
(|N| = 2, |M| = 8)

MHS found: [22] of dimension 1
MHS encountered : 1

MHS found: [32] of dimension 1
MHS encountered : 2

MHS found: [2, 4] of dimension 2
MHS encountered : 3

MHS found: [2, 9] of dimension 2
MHS encountered : 4

MHS found: [2, 10] of dimension 2
MHS encountered : 5

MHS found: [2, 17] of dimension 2
MHS encountered : 6

MHS found: [2, 24] of dimension 2
MHS encountered : 7

MBASE required 0.0918 seconds to execute
(|N| = 2, |M| = 33)
```

Figura 7 – 74L85.000.matrix sperimentazione

In figura 7 possiamo osservare l'esecuzione dell'algoritmo su una matrice 2x33. Nel primo caso, senza pre-elaborazione, è stato necessario un tempo di circa 0.21 secondi. Successivamente, è stata applicata la pre-elaborazione che ha richiesto un tempo di circa 0.012 secondi. Le righe e le colonne che sono state cancellate sono riportate in alto a destra così come le dimensioni di A'. Il tempo di calcolo di *MBase(A')* è stato di 0.0918, quindi migliore della prima esecuzione. Infine, è possibile osservare che non vi è alcuna discrepanza nei due risultati: sono stati trovati gli stessi *mhs* e nello stesso ordine.

La figura 8, invece, illustra le prestazioni ottenute dall'elaborazione di una matrice A 4x160. Non sono riportati i risultati completi della risoluzione di questa istanza per motivi di spazio ma viene comunque mostrato l'ultimo *mhs* trovato sia per la normale esecuzione che per quella preceduta da pre-elaborazione. Anche in questo caso possiamo notare che l'ultimo *hitting set* minimale trovato coincide in entrambi i casi così come il numero di *mhs*

individuati. Le prestazioni sono migliori nel secondo caso anche se rimangono dello stesso ordine di grandezza. Una prima motivazione può essere il fatto che A ha un elevato numero di colonne e il numero di *hitting set* minimale è davvero elevato (854). Secondariamente, occorre tenere a mente che i tempi di esecuzione di un qualunque programma variano di volta in volta in base all'occupazione dei processori, il cui numero influisce a sua volta, e allo *scheduling* dei processi. Per queste ragioni può essere che l'esecuzione dell'algoritmo sia stata effettuata in un momento di particolare concentrazione di applicazioni in stato *running*. Sono quindi state condotte altre prove che però hanno dato risultati simili se non leggermente peggiori. Infine, è necessario ricordare che *Python* è un linguaggio interpretato e dinamicamente tipizzato (al contrario di *C* o *Java*) quindi registra generalmente prestazioni peggiori rispetto ad altri linguaggi.

```
(|N| = 4, |M| = 160)  MBASE required 29.2276 seconds to execute
MHS found: [142 158 159] of dimension 3
MHS encountered : 854

c432.000.matrix
Rows dropped in preprocessing:  []
Columns dropped in preprocessing:  [1, 2, 4, 5, 6, 8, 10, 12, 13, 14, 15, 16, 17
, 18, 19, 21, 23, 27, 28, 29, 30, 31, 33, 34, 35, 36, 37, 39, 41, 43, 45, 50, 52
, 55, 56, 57, 58, 59, 60, 61, 66, 67, 69, 78, 79, 80, 81, 82, 83, 84, 85, 87, 88
, 89, 90, 91, 92, 93, 94, 95, 96, 99, 100, 101, 102, 103, 104, 105, 106, 107, 10
8, 109, 110, 111, 112, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 12
5, 126, 127, 128, 129, 130, 131, 132, 133, 135, 136, 137, 138, 139, 145, 146, 14
7, 148, 152, 153, 157]
Preprocessing required 0.051634 seconds to execute
(|N| = 4, |M| = 53)
MBASE required 24.3910 seconds to execute
MHS found: [142, 158, 159] of dimension 3
MHS encountered : 854
```

Figura 8 – c432.000.matrix sperimentazione

5. Utilizzo dell'applicazione

Per utilizzare l'applicazione è necessario installare *Python* 3.6 sulla propria macchina, scaricare il package *numpy* e avere cura che esso sia posizionato nella stessa cartella in cui è presente il compilatore. Queste operazioni possono essere svolte automaticamente e senza preoccupazioni utilizzando un qualsiasi ambiente di sviluppo *Python* o attraverso linea di comando.

È inoltre necessario che il file *.matrix* contenente l'istanza del problema sia all'interno della stessa cartella del codice sorgente *mhs.py*.

Per avviare l'esecuzione del programma si può utilizzare un qualunque *IDE* ed è possibile fermarla anticipatamente tramite il pulsante di stop fornito dall'ambiente di sviluppo stesso. In alternativa, è possibile eseguire l'applicazione da riga di comando scrivendo ' $ python3 mhs.py ' oppure tramite *Python Shell IDLE*. Anche in questo caso è possibile arrestare anticipatamente l'esecuzione del programma tramite la combinazione "*Ctrl + c*". Per quanto riguarda la terminazione anticipata, qualunque sia l'ambiente dal quale è partita l'esecuzione del programma, essa verrà notificata fornendo un messaggio riportante la pila delle chiamate delle funzioni al momento dell'arresto oppure tramite un messaggio di "interruzione anomala". Se invece il programma giunge alla normale terminazione verrà stampato il messaggio "*Execution completed*".

All'avvio del programma sarà richiesto all'utente di inserire il nome del file *.matrix* da elaborare.

6. *Conclusioni*

In conclusione, si può affermare che dalle prove condotte, compreso l'esempio illustrato all'interno delle specifiche progettuali, sembra che appaiano risultati corretti e in tempi ragionevoli, fintantoché il numero di colonne di A si mantiene nell'ordine delle centinaia. Qualora invece il loro numero cresca ulteriormente, le prestazioni degradano e le cause sono da ricondursi anche alla natura del linguaggio utilizzato.

Sono state adottate ottimizzazioni riguardo l'occupazione di memoria tramite le strutture dati fornite dalla libreria *numpy* e si sono sfruttate pesantemente le funzioni associate a questo *package* in quanto ben progettate e, generalmente, più "veloci" delle omologhe native del linguaggio.

Nel complesso, l'applicazione è facilmente utilizzabile e l'output è estremamente comprensibile. Inoltre, la funzionalità che permette una terminazione anticipata evita di impiegare un'eccessiva quantità di tempo prima di avere i primi *minimal hitting set*.