

Algoritmi e strutture dati  
Implementazione di un algoritmo per il calcolo  
degli hitting set minimali

Lorenzi Patrick mat.719080, Rossi Francesco mat.706086

Anno accademico 2020-2021

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Il linguaggio Python . . . . .	1
1.1.1	Tipi di allocazione della memoria . . . . .	2
1.1.2	Quanta memoria occupa un oggetto? . . . . .	3
<b>2</b>	<b>Preparazione dei dati &amp; strutture dati</b>	<b>8</b>
2.1	Lettura dei dati . . . . .	8
2.1.1	La funzione leggi_matrice . . . . .	9
2.1.2	La funzione leggi_dominio . . . . .	10
2.1.3	Convenzioni utilizzate . . . . .	10
2.2	Strutture dati . . . . .	12
2.2.1	Lista . . . . .	12
2.2.2	Queue . . . . .	19
2.2.3	Array . . . . .	25
<b>3</b>	<b>L'algoritmo MBase</b>	<b>28</b>
3.1	Massimo di un insieme e successore di un elemento . . . . .	28
3.2	Algoritmo base . . . . .	30
3.2.1	Calcolo vettore rappresentativo . . . . .	31
3.2.2	La funzione check . . . . .	33
3.2.3	Output . . . . .	34
3.3	Accorgimenti sull'uso della memoria . . . . .	35
3.3.1	Overhead di memorizzazione . . . . .	35
3.3.2	La classe MultiInsieme . . . . .	36
<b>4</b>	<b>Funzioni di pre-elaborazione</b>	<b>41</b>
4.1	Rimozione delle righe . . . . .	41
4.1.1	Contiene . . . . .	41

4.1.2	Costruisci_array . . . . .	42
4.1.3	Togli_righe . . . . .	44
4.2	Rimozione delle colonne . . . . .	45
4.2.1	Colonna_di_zero . . . . .	45
4.2.2	Togli_colonne . . . . .	46
<b>5</b>	<b>Sperimentazione</b>	<b>48</b>
5.1	Metodo di raccolta dati . . . . .	48
5.1.1	Formato dei dati raccolti . . . . .	49
5.2	Limitazioni riscontrate . . . . .	51
5.2.1	Gestione dell'out of memory . . . . .	51
5.2.2	Swap . . . . .	52
5.3	Risultati . . . . .	53
5.3.1	Prestazioni temporali . . . . .	54
5.3.2	Prestazioni spaziali . . . . .	57
5.3.3	Accorgimenti per la riduzione dell'uso della memoria . . . . .	59
<b>6</b>	<b>Manuale utente</b>	<b>63</b>
6.1	Installazione . . . . .	63
6.2	Eseguire il programma . . . . .	64
6.3	Opzioni . . . . .	70
6.4	PyPy . . . . .	70
<b>7</b>	<b>Conclusioni</b>	<b>72</b>

## Sommario

In questo elaborato viene discussa l'implementazione dell'algoritmo per la ricerca di un insieme di *hitting minimal set*, data una collezione di insiemi  $N$ , definita su un dominio  $M$ . Il gruppo di lavoro ha scelto di realizzare la seconda alternativa proposta nell'elaborato, ovvero l'implementazione dell'algoritmo MBase e dello stesso algoritmo preceduto dalle operazioni di pre-elaborazione definite nella specifica, `togli_righe` e `togli_colonne`.

Nella prima parte di questo lavoro viene fatta una panoramica sul linguaggio di programmazione utilizzato, e ne vengono illustrate le caratteristiche principali. Successivamente sono presentate le funzioni per la lettura dei dati da sottomettere, le strutture dati utilizzare. Viene poi commentata l'implementazione dell'algoritmo principale, le scelte implementative fatte, con l'aggiunta delle funzionalità di pre-elaborazione richieste. Nel capitolo 5 vengono discussi i risultati ottenuti dell'esecuzione di entrambe le varianti (con o senza pre-elaborazione) su una selezione dei file `.matrix` forniti. Infine è possibile consultare le istruzioni per eseguire il programma. L'applicazione è stata realizzata utilizzando il linguaggio Python, scelta fatta vista l'estrema semplicità e leggibilità, sia a livello di codice, sia nella manipolazione dei dati. Il linguaggio utilizzato però, non è esente da limitazioni, le quali vengono illustrate nella sezione successiva.

# Capitolo 1

## Introduzione

Diamo ora, in questa prima parte, uno sguardo generale sul linguaggio Python, questo può essere utile per capire alcune scelte effettuate.

### 1.1 Il linguaggio Python

Python è un linguaggio di programmazione ad oggetti, dinamico, interpretato, e non tipizzato, cioè non è necessario dichiarare il tipo di una variabile che si vuole utilizzare, discorso valido anche per le funzioni. La gestione della memoria di Python, come in altri linguaggi di programmazione a oggetti, è a carico di un'entità chiamata garbage collector, che si occupa di deallocare eventuali oggetti rimasti senza riferimento. Nella maggior parte dei casi il programmatore non deve intervenire sull'operato del garbage collector, consentendo così di concentrarsi esclusivamente sul codice. A causa della sua semplicità, tuttavia, Python non offre molta libertà nella gestione dell'utilizzo della memoria, a differenza di linguaggi come C e C++ in cui è possibile allocare manualmente e liberare memoria.

Come detto in precedenza, Python è un linguaggio di programmazione ad oggetti. Affinché questi oggetti siano utili, devono essere archiviati nella memoria per potervi accedere. Prima che possano essere archiviati, è necessario allocare o assegnare un blocco di memoria per ciascuno di essi.

Al livello più basso, l'allocatore di memoria di Python si assicurerà prima che ci sia spazio disponibile nell'heap privato per archiviare questi oggetti. Lo fa interagendo con il gestore della memoria del sistema operativo. Succes-

sivamente, diversi allocatori specifici dell'oggetto operano sullo stesso heap e implementano politiche di gestione distinte a seconda del tipo di oggetto.

### 1.1.1 Tipi di allocazione della memoria

Esistono 2 modi per allocare la memoria: allocazione statica e dinamica.

Allocazione statica:

- Le variabili allocate staticamente sono permanenti, il che significa che devono essere allocate in anticipo e durano finché il programma viene eseguito.
- La memoria viene allocata durante la compilazione o prima dell'esecuzione del programma.
- Implementata utilizzando lo stack, come struttura dati per contenere le variabili.
- La memoria che è stata allocata non può essere riutilizzata.

Allocazione dinamica:

- Le variabili allocate dinamicamente non sono permanenti e possono essere allocate mentre un programma è in esecuzione.
- La memoria viene allocata in fase di esecuzione o durante l'esecuzione del programma.
- Implementata utilizzando un heap come struttura dati per contenere le variabili.
- La memoria allocata può essere rilasciata e riutilizzata.

Un vantaggio (o svantaggio sotto certi punti di vista) dell'allocazione dinamica della memoria in Python è che non dobbiamo preoccuparci in anticipo di quanta memoria abbiamo bisogno per il nostro programma. Tuttavia, poiché l'allocazione dinamica della memoria viene eseguita durante l'esecuzione del programma, richiede più tempo per il suo completamento. Inoltre, la memoria allocata deve essere liberata dopo che è stata utilizzata. In caso contrario, potrebbero verificarsi problemi di perdite di memoria.

I metodi e le variabili vengono creati nello stack, mentre gli oggetti e le variabili di istanza vengono creati nello heap. Non appena le variabili e le funzioni vengono restituite, gli oggetti inutilizzati verranno liberati tramite il garbage collector.

È importante sottolineare che il gestore della memoria di Python non rilascia necessariamente la memoria al sistema operativo, ma viene restituita all'interprete. Python ha un piccolo allocatore di oggetti che mantiene la memoria allocata per un ulteriore utilizzo. Nei processi di lunga durata, potrebbe essere un problema perchè si creerebbe una riserva incrementale di memoria inutilizzata. Se da una parte la gestione automatica della memoria può risultare comoda, nei casi in cui è necessario lavorare con istanze di problemi di grandi dimensioni che richiedono un uso intensivo di essa, senza poter gestirla direttamente, rischia di diventare una forte limitazione.

### 1.1.2 Quanta memoria occupa un oggetto?

Diversamente dagli altri linguaggi, come il C o il Java in cui le variabili hanno uno spazio ben definito all'interno della memoria, in Python stabilire la dimensione di questi è più difficile: di seguito abbiamo 3 risposte a questo quesito, la prima semplice ma sbagliata, la seconda leggermente più complessa e l'ultima quella esatta.

#### La risposta semplice e sbagliata

In Python la funzione base per misurare le dimensioni di un oggetto è *sys.getsizeof()*. Vediamo un esempio:

```
1 >>> sys.getsizeof(1)
2 28
3 >>> sys.getsizeof('Python')
4 55
5 >>> sys.getsizeof([1, 2, 3])
6 88
7
```

Questi tre esempi mostrano le dimensioni di un intero, di una stringa e di una lista in bytes. Vediamo adesso un altro esempio:

```
1 >>> sys.getsizeof(',')
2
```

```
3 49
4 >>> sys.getsizeof('P')
5 50
6 >>> sys.getsizeof('Py')
7 51
8 >>> sys.getsizeof('Pyt')
9 52
```

Nella primo caso si ha una stringa vuota che occupa 49 byte, mentre il secondo è una stringa con un solo carattere e la sua dimensione è di 50 byte. Quindi ho una stringa con un solo carattere e la sua dimensione è di 50 byte. Aggiungendo un carattere la dimensione della stringa è aumentata di un byte. La stringa è un oggetto, non è solo una sequenza di caratteri. Un oggetto (in questo caso, un oggetto stringa), oltre al suo valore (cioè, la sequenza di caratteri), ha attributi diversi e componenti correlati che noi non vediamo. Quando creiamo un oggetto, Python memorizza tutte queste informazioni in memoria. Pertanto, si ha comunque una certa occupazione di memoria anche per una stringa vuota. Lo stesso discorso vale per una lista:



```

1
2 >>> sys.getsizeof([])
3 64
4 >>> sys.getsizeof([1])
5 72
6 >>> sys.getsizeof([1, 2])
7 80

```

Vediamo ora un esempio più interessante:

```

1
2 >>> sys.getsizeof([1, 2])
3 80
4 >>> sys.getsizeof([3, 4, 5, 1])
5 96
6 >>> sys.getsizeof([1, 2, [3, 4, 5, 1]])
7 88

```

Prima si ha una lista che occupa 80 bytes di memoria. Si ha una seconda lista che ne occupa 96. Per una lista si hanno 64 bytes di "metadati" e 8 bytes aggiuntivi per ogni elemento. Ora, inserendo la seconda lista nella prima ottengo un nuovo oggetto che occupa 88 bytes, ben minore delle dimensioni della seconda lista (di 96 bytes). Come è possibile?

Innanzitutto, avevo una lista di due elementi (di numeri interi per la precisione). Quando è stato aggiunto un nuovo elemento, che era una lista, ha aggiunto 8 byte alla lista in memoria. Sembra che, in ogni caso, un elemento aggiuntivo richieda 8 byte. Sembra che una lista non stia memorizzando gli elementi ma un riferimento agli elementi (cioè gli indirizzi di memoria), ed infatti è così. Quando si crea una lista, da sola occupa 64 byte di memoria e ogni elemento aggiunge 8 byte di memoria alla dimensione della lista a causa dei riferimenti ad altri oggetti. Significa che nell'esempio precedente, la lista a riga 6 è memorizzata come [riferimento a obj1, riferimento a obj2, riferimento a obj3]. La dimensione di ogni riferimento è di 8 byte. In questo caso, obj1, obj2 e obj3 sono archiviati da qualche altra parte nella memoria. Pertanto, per ottenere la dimensione effettiva della lista in questione, oltre a ottenere la dimensione della lista, va inclusa la dimensione di ciascun oggetto dentro ad essa.

## Una risposta più complessa e più accurata

Come visto nel paragrafo precedente, `sys.getsizeof()` restituisce solo la dimensione e i suoi attributi in memoria. Non comprende le dimensioni degli oggetti referenziati e ai loro attributi. Per ottenere la dimensione di un oggetto dovremmo iterare su tutti i componenti dello stesso e sommare le loro dimensioni, come in figura.

Object		Size	Memory Address
[1, 2, [3, 4, 5, 1]]		88 Bytes	0x241f767be08
+	1	28 Bytes	0x7ffdf176a190*
+	2	28 Bytes	0x7ffdf176a1b0
+	[3, 4, 5, 1]	96 Bytes	0x241f767bf88
+	3	28 Bytes	0x7ffdf176a1d0
+	4	28 Bytes	0x7ffdf176a1f0
+	5	28 Bytes	0x7ffdf176a210
+	1	28 Bytes	0x7ffdf176a190*
size([1, 2, [3, 4, 5, 1]])		352 Bytes	

\* This answer is wrong. See the next figure for the correct number.

Come mostra la figura, la dimensione dell'oggetto finale è di 352 byte. Tuttavia, c'è un errore di calcolo. Nell'elenco degli oggetti nella figura, compaiono gli stessi indirizzi di memoria sulle righe 2 e 8 (evidenziate con \*). Sembra che 1 (cioè un oggetto intero) nell'elenco principale e 1 nell'elenco nidificato siano archiviati nello stesso indirizzo di memoria. Python memorizza i numeri interi tra [-5, 256] una volta, e punta tutti i riferimenti allo stesso indirizzo di memoria (per l'ottimizzazione della memoria). Pertanto, è meglio identificare i duplicati utilizzando i loro indirizzi di memoria (tramite `id()`) e contare le dimensioni della memoria una volta. Pertanto, vanno rimossi i duplicati prima di sommare le loro dimensioni di memoria. Perciò va rimossa l'ultima riga e il risultato ottenuto è di 324 byte.

## Una risposta ancora più accurata

La risposta precedente era più accurata di quella di partenza, ma necessita ancora di qualche accorgimento. Quando una classe è caricata, alcuni elementi nascosti (ad es. `obj.__dict__` o `obj.__slots__`), potrebbero venir inseriti nella memoria. Tracciare questi elementi manualmente è difficile, e talvolta impossibile. Il modo migliore per cercare tutti gli elementi collegati ad un oggetto è quello di usare una funzione dall'interfaccia di Python Garbage Collector chiamata `gc.get_referents()`. Il codice seguente itera su tutti gli oggetti ed elementi collegati all'oggetto principale e aggiunge le loro dimensioni alla dimensione totale dell'oggetto finale.

```
1
2 import sys
3 import gc
4
5 def actual_size(input_obj):
6     memory_size = 0
7     ids = set()
8     objects = [input_obj]
9     while objects:
10         new = []
11         for obj in objects:
12             if id(obj) not in ids:
13                 ids.add(id(obj))
14                 memory_size += sys.getsizeof(obj)
15                 new.append(obj)
16         objects = gc.get_referents(*new)
17     return memory_size
18
19 actual_size([1, 2, [3, 4, 5, 1]])
```

Determinare la dimensione di un oggetto in Python non è così facile. non esiste una soluzione pronta all'uso e semplice. La soluzione proposta funziona con molti (ma non con tutti) oggetti in Python, e non è sicuramente l'unica.

# Preparazione dei dati & strutture dati

## 2.1 Lettura dei dati

[illegible]

Le prime 4 righe non sono d'interesse per i test, mentre la 5° contiene il mapping del dominio, posizione = valore, e in questo caso la posizione parte dall'indice 1 fino ad arrivare a M, il numero di colonne della matrice. Segue poi la matrice di 0 e 1, dove ad ogni riga corrisponde un diverso insieme. Un 1 indica che quella posizione è occupata da un elemento del dominio, in accordo con il mapping precedentemente descritto.

Sono stati quindi realizzate le funzioni necessarie per la lettura della matrice e del dominio da file, al fine di salvarli in strutture dati apposite.

### 2.1.1 La funzione `leggi_matrice`

```
1 def leggi_matrice(percorso_file):
2     matrice = []
3     with percorso_file.open('r') as file:
4         for riga in file:
5             if riga[0] != ";":
6                 nuova_riga = riga.replace("-", "")
7                 elementi = nuova_riga.split(" ")
8                 elementi.remove("\n")
9                 matrice.append(arr.array('B', list(map(int,
10 elementi))))
11     return matrice
```

Il metodo *leggi\_matrice* riceve in ingresso il percorso relativo al file *.matrix* al fine di estrarre la matrice contenuta in esso. Il file viene fatto scorrere riga per riga, escludendo le righe che iniziano con ";" perchè non sono di interesse. Vengono poi rimossi i caratteri "-" che indicano la fine di una riga e vengono separati tutti gli 0 e 1 nella riga. Successivamente viene fatto un "cast" per trasformare i valori di cella ottenuti da stringhe a interi (realizzato tramite un map della funzione *int* sull'intera lista). Infine la lista così ottenuta è data in ingresso al metodo *arr.array* che costruisce un array di valori di tipo "B" che sta ad indicare un vettore che contiene tutti valori di tipo unsigned char. Ogni valore occupa quindi 1 byte. Infine, ogni riga che rappresenta un insieme viene inserita in un oggetto *list*, che rappresenta quindi la matrice letta da file.

### 2.1.2 La funzione `leggi_dominio`

```
1 def leggi_dominio(nome_file):  
2     riga_dominio = linecache.getline(nome_file, 5)  
3     dominio = re.findall('\(([^\)]+)\)', riga_dominio)  
4     return dominio, riga_dominio
```

Questa funzione estrae dal file (passato tramite il suo nome) esclusivamente la riga 5 del file di input, e grazie all'espressione regolare alla riga 4 estrae i valori effettivi del dominio contenuti tra le parentesi, e li ritorna in una lista. Oltre a ciò ritorna anche la riga stessa, affinché possa essere inserita successivamente nei file di output rappresentanti i MHS calcolati per una matrice.

### 2.1.3 Convenzioni utilizzate

In generale in questo lavoro è stato deciso di adottare le seguenti convenzioni a livello operativo:

- Guardando con attenzione i file di benchmark forniti, è possibile che si verifichi il caso in cui il numero dei valori del dominio siano diversi dal numero delle colonne della matrice, questo creerebbe l'impossibilità di mappare tutti i valori contenuti nei vettori di  $N$ , con i rispettivi valori di dominio  $M$ , secondo la logica `posizione=valore`. Per questo, una volta estratti matrice e dominio viene fatto un controllo sulle loro rispettive dimensioni: in caso positivo lo script procede regolarmente, in caso negativo lo script viene interrotto presentando un errore del tipo: "Problema sulle dimensioni tra dominio e matrice". Per ipotesi abbiamo assunto che tutte le righe della matrice avessero uguali dimensioni.
- Al posto di lavorare direttamente con i valori di dominio del tipo `['z1', 'z2', 'z3']`, all'interno dell'algoritmo base, si è preferito lavorare con le rispettive posizioni es. `[0,1,2]` come illustrato dal "map" nei file `.matrix`. In questo modo il salvataggio e lo scorrimento di numeri (int) dentro alle strutture dati risulta più efficiente in termini di memoria e di operazioni. I risultati finali sono restituiti riconducendoci al dominio letto in precedenza, oppure rappresentati utilizzando come formato lo stesso utilizzato nel file `.matrix` di input.

M	z1	z2
---	----	----

M	z1	z2	z3	z4
---	----	----	----	----

N	0	0	1
N	1	1	1
N	1	0	0

N	0	0	1
N	1	1	1
N	1	0	0

Figura 2.1: Esempi di matrici trovati nei file di benchmark, che causano problemi

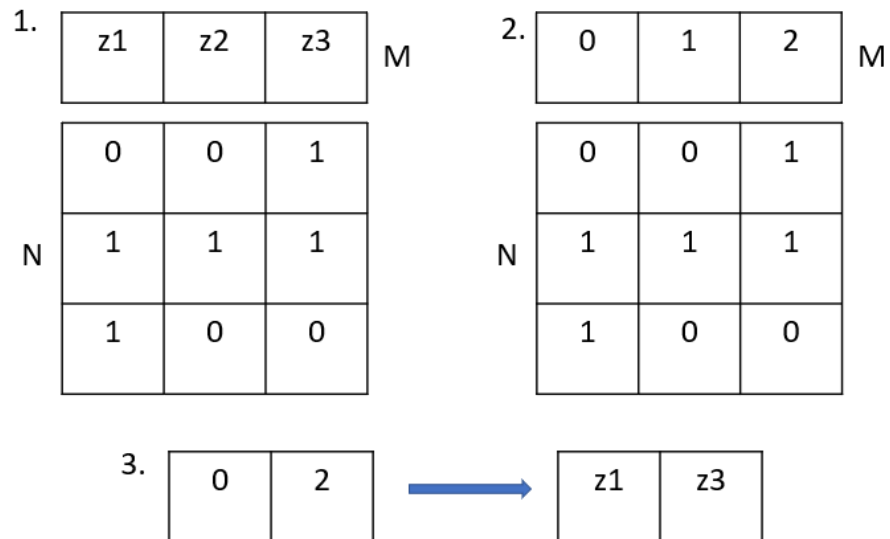


Figura 2.2: 1. Dati rappresentati dal file, la matrice N e il dominio M; 2. Matrice N data in ingresso con la convenzione su indice=elemento; 3. Risultato in output basato sugli indici e di come va ri-mappato.

## 2.2 Strutture dati

Sono ora discusse le strutture dati messe a disposizione da Python e come sono state utilizzate all'interno dell'algoritmo.

### 2.2.1 Lista

La lista è la struttura dati più comune messa a disposizione dal linguaggio. Essa è una collezione di oggetti ordinata, indicizzata e modificabile e i suoi elementi sono oggetti arbitrari. E' stato fatto largo uso nel codice in quelle situazioni in cui non si conosceva con precisione il numero di elementi da mantenere nella struttura dati o in situazioni nelle quali fosse richiesto di far inserimenti o cancellazioni multiple e veloci di elementi. Internamente, una lista è rappresentata come un array; i costi maggiori derivano dall'aumento oltre l'attuale dimensione allocata (perché tutto va spostato) o dall'inserimento o eliminazione in qualche cella nella zona iniziale (perché tutto ciò che segue deve spostarsi). Volendo questa struttura dati può essere utilizzata anche come una coda, tuttavia Python mette a disposizione un'implementazione della stessa, fornita nel modulo *collections.deque*. I principali metodi di una lista sono:

	Nome	C. temporale medio	C. spaziale
Aggiunta in coda	<code>list.append(x)</code>	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Rimozione elem. x	<code>list.remove(x)</code>	$\mathcal{O}(n)$	$\mathcal{O}(n)$
get alla posizione x	<code>list[x]</code>	$\mathcal{O}(1)$	$\mathcal{O}(n)$

Tabella 2.1: Principali metodi di una lista

E' bene ricordare che Python viene implementato a basso livello utilizzando altri linguaggi di programmazione, per esempio il C. Questo porta all'esistenza di diverse implementazioni del linguaggio, caratterizzate da avere caratteristiche comuni definite da Python, e altre dipendenti dal linguaggio sottostante. Nel nostro caso viene discussa l'implementazione chiamata CPython, che fa uso del linguaggio C. Un oggetto di tipo lista in CPython è rappresentato dalla seguente struttura C. "ob\_item" è una matrice di puntatori agli elementi della lista. "allocated" è il numero di slot allocati in memoria.



```

1 typedef struct {
2     PyObject_VAR_HEAD
3     PyObject **ob_item;
4     Py_ssize_t allocated;
5 } PyListObject;

```

Vediamo ora cosa succede quando inizializziamo una lista vuota, `l=[]`:

```

1 arguments: size of the list = 0
2 returns: list object = []
3 PyListNew:
4     nbytes = size * size of global Python object = 0
5     allocate new list object
6     allocate list of pointers (ob_item) of size nbytes = 0
7     clear ob_item
8     set list's allocated var to 0 = 0 slots
9     return list object

```

È importante notare la differenza tra gli slot allocati e la dimensione della lista. La dimensione di una lista è la stessa di `len(l)`. Il numero di slot allocati è quello che è stato allocato in memoria. Spesso si nota che *allocated* è maggiore della dimensione, al fine di evitare chiamate a *realloc* ogni volta che un nuovo elemento viene aggiunto alla lista.

Aggiungiamo un numero alla lista: `l.append(1)`. La funzione interna `app1()` viene chiamata:

```

1 arguments: list object, new element
2 returns: 0 if OK, -1 if not
3 app1:
4     n = size of list
5     call list_resize() to resize the list to size n+1 = 0 +
6     1 = 1
7     list[n] = list[0] = new element
8     return 0

```

Vediamo `list_resize()`. Essa va a allocare memoria in eccesso per evitare di chiamare `list_resize` troppe volte. Il pattern di crescita di una lista è: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...

```

1 arguments: list object, new size
2 returns: 0 if OK, -1 if not

```

```

3  list_resize:
4      new_allocated = (newsize >>> 3) + (newsize < 9 ? 3
      : 6) = 3
5      new_allocated += newsize = 3 + 1 = 4
6      resize ob_item (list of pointers) to size new_allocated
7      return 0

```

Sono stati allocati 4 slot per contenere gli elementi, ed il primo è l'intero 1. Nella figura successiva si può vedere che `l[0]` punta all'oggetto intero aggiunto. I quadrati tratteggiati rappresentano gli slot assegnati ma non ancora utilizzati. La complessità ammortizzata dell'*append* è  $\mathcal{O}(1)$ .

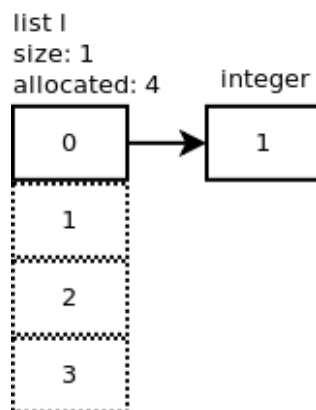


Figura 2.3: Aggiunta di un elemento alla lista `l`.

Continuiamo aggiungendo un altro elemento: `l.append(2)`. `list_resize` viene chiamato con `n+1=2` ma poiché la dimensione allocata è 4, non è necessario allocare più memoria. La stessa cosa accade quando aggiungiamo altri 2 interi: `l.append(3)`, `l.append(4)`.

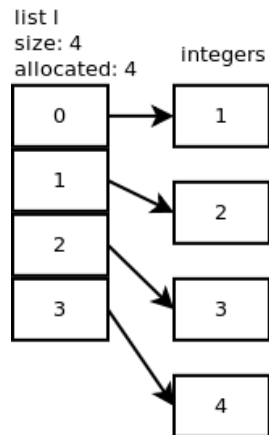


Figura 2.4: Lista contenente 4 elementi.

Proviamo ora ad inserire un altro intero, 5, alla posizione 1: *l.insert(1,5)*: *ins1()* viene chiamato internamente.

```

1 arguments: list object, where, new element
2 returns: 0 if OK, -1 if not
3 ins1:
4     resize list to size n+1 = 5 -> 4 more slots will be
      allocated
5     starting at the last element up to the offset "where"
      right shift each element
6     set new element at offset "where"
7     return 0

```

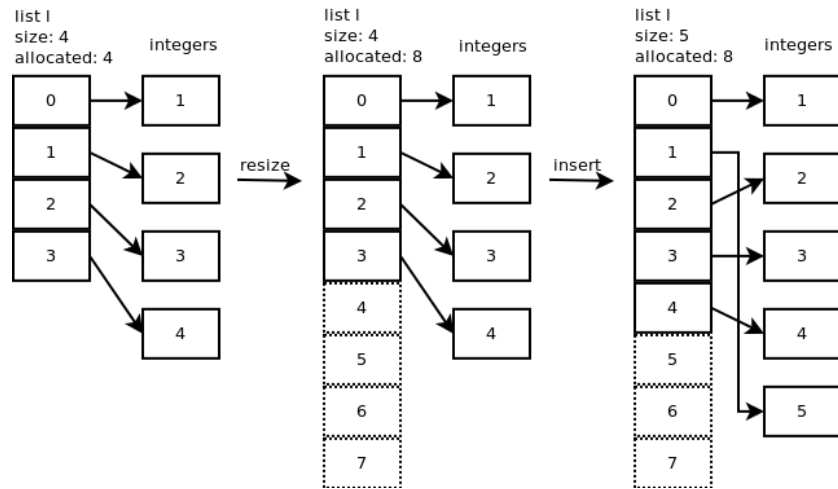


Figura 2.5: Inserimento dell'elemento 5 in posizione '1'.

I quadrati tratteggiati rappresentano gli slot assegnati ma non ancora utilizzati. Qui vengono allocati 8 slot, ma la dimensione o la lunghezza dell'elenco è solo 5. La complessità dell'operazione di inserimento è  $\mathcal{O}(n)$ .

Quando faccio `l.pop()`, viene chiamato `listpop()`. `list_resize` viene chiamato all'interno di `listpop()` e se la nuova dimensione è inferiore alla metà della dimensione allocata, la lista viene ridotta.

```

1 arguments: list object
2   returns: element popped
3 listpop:
4   if list empty:
5       return null
6   resize list with size 5 - 1 = 4. 4 is not less than 8/2
   so no shrinkage
7   set list object size to 4
8   return last element

```

La complessità di `pop()` è  $\mathcal{O}(1)$ .

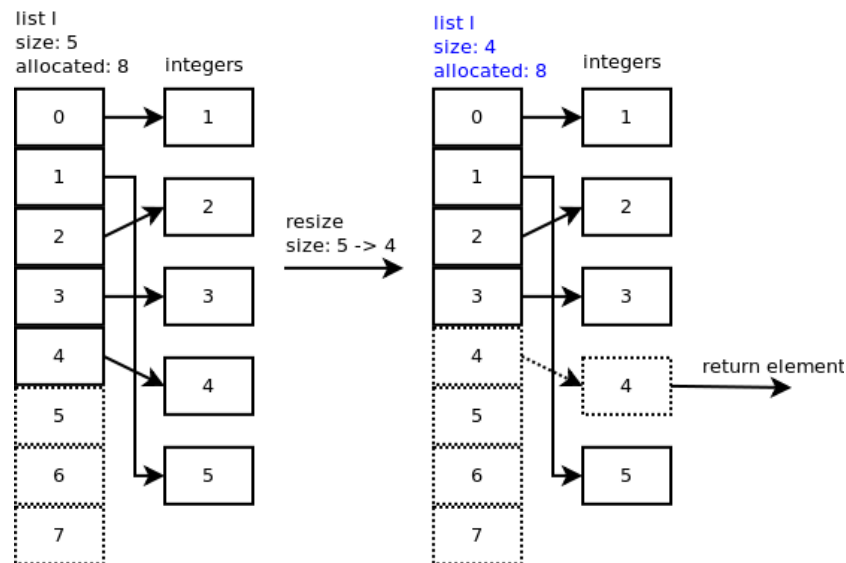


Figura 2.6: Esecuzione di 'pop'.

Lo slot 4 punta ancora all'intero, ma l'importante è che la dimensione dell'elenco è 4.

Invochiamo un altro `pop()`. In `list_resize()`,  $\text{size} - 1 = 4 - 1 = 3$  è meno della metà degli slot allocati, quindi la lista viene ridotta a 6 slot e la nuova dimensione è 3. Si osserva che gli slot 3 e 4 puntano ancora ad alcuni numeri interi, ma la dimensione è di 3.

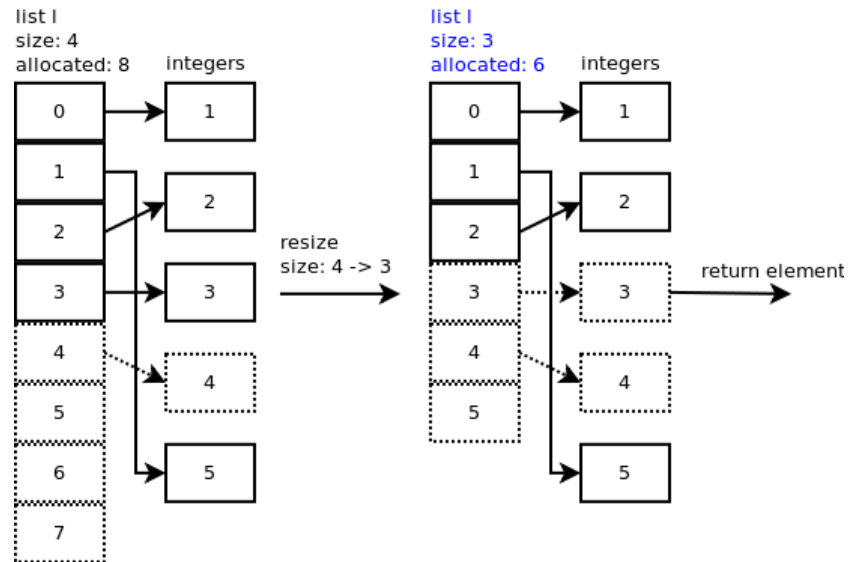


Figura 2.7: Ulteriore esecuzione di 'pop'. Adesso 3 e 4 non sono più in l.

Rimuoviamo ora l'elemento 5 tramite *l.remove(5)* invocando *listremove()*.

```

1 arguments: list object, element to remove
2 returns none if OK, null if not
3 listremove:
4     loop through each list element:
5         if correct element:
6             slice list between element's slot and element's
7             slot + 1
8             return none
9         return null

```

Per dividere la lista e rimuovere l'elemento, viene chiamato *list\_ass\_slice()*. Qui, l'offset basso è 1 e l'offset alto è 2, poiché stiamo rimuovendo l'elemento 5 nella posizione 1. La complessità del metodo remove è  $\mathcal{O}(n)$ .

```

1 arguments: list object, low offset, high offset
2 returns: 0 if OK
3 list_ass_slice:
4     copy integer 5 to recycle list to dereference it
5     shift elements from slot 2 to slot 1
6     resize list to 5 slots
7     return 0

```

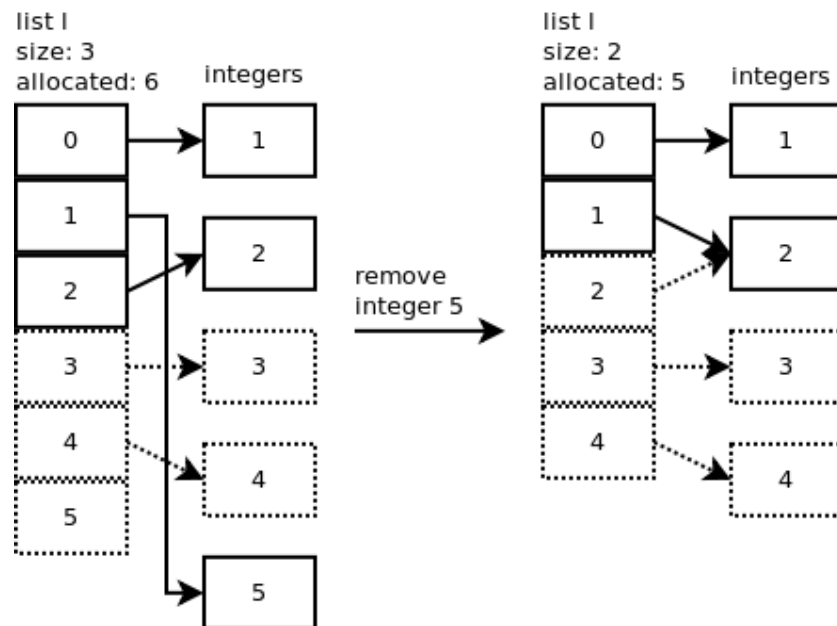


Figura 2.8: Esecuzione di `remove` sull'elemento 5.

## 2.2.2 Queue

È possibile utilizzare una lista come coda, secondo l'approccio FIFO ("first-in, first-out"); tuttavia, le liste non sono efficienti per questo scopo. Mentre aggiunta e la rimozione dalla fine di una lista sono veloci, farlo dall'inizio è lento (perché tutti gli altri elementi devono essere spostati di uno). Per implementare una coda, abbiamo usato il modulo *collections.deque*, progettato per avere operazioni veloci da entrambe le estremità. Questa implementazione è stata utilizzata all'interno dello script per contenere quegli insiemi che non sono *MHS* ma potrebbero ancora diventarlo, così come prescritto dallo pseudocodice dell'algoritmo fornito. Di seguito verrà presentato come

la coda è stata implementata a basso livello e come si comporta con alcune chiamate ai suoi metodi.

	Nome	C. temporale medio	C. spaziale
Aggiunta in coda	<code>append(x)</code>	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Rimozione elem. x	<code>remove(x)</code>	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Get elem alla posizione x	<code>list[x]</code>	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Rimuovi e ritorna l'ultimo elem	<code>pop()</code>	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Rimuovi e ritorna il primo elem	<code>popleft()</code>	$\mathcal{O}(1)$	$\mathcal{O}(n)$

Tabella 2.2: Principali metodi di deque

L'oggetto deque è una coda con doppia estremità. Si possono aggiungere elementi alle estremità o rimuoverli in un tempo ammortizzato a quello costante. Creando un oggetto vuoto, Python va a creare una linked list di 64 puntatori nulli. L'indice destro è inizializzato posizionandolo al centro del blocco, sulla cella 31, ed è usato come puntatore all'ultimo elemento nella coda. L'indice di sinistra è speculare, è posizionato sullo slot 32, andrà a puntare sul primo oggetto nella coda.

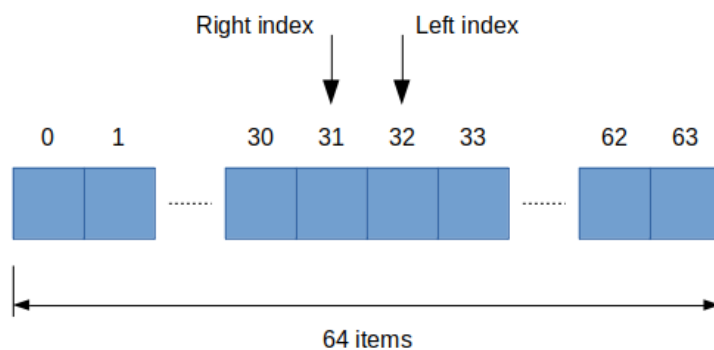


Figura 2.9: Deque vuota.

La coda vista sull'implementazione del codice C presenta anche un puntatore al blocco più a sinistra e più a destra. Esiste anche un attributo opzionale "maxlen" usato per ottenere code limitate.



```

1 typedef struct {
2     ...
3     block *leftblock;
4     block *rightblock;
5     Py_ssize_t leftindex;
6     Py_ssize_t rightindex;
7     ...
8     Py_ssize_t maxlen;
9     ...
10 } dequeobject;

```

Un blocco ha un link sinistro e destro. Ha anche un array di 64 puntatori a degli oggetti.

```

1 typedef struct BLOCK {
2     struct BLOCK *leftlink;
3     PyObject *data[BLOCKLEN];
4     struct BLOCK *rightlink;
5 } block;

```

Aggiungiamo adesso 2 elementi in fondo tramite *append()* 'e1' ed 'e2'.

L'indice di destra è incrementato di 2, posizionandosi su 33. Quello di sinistra non si è mosso e punta il primo elemento.

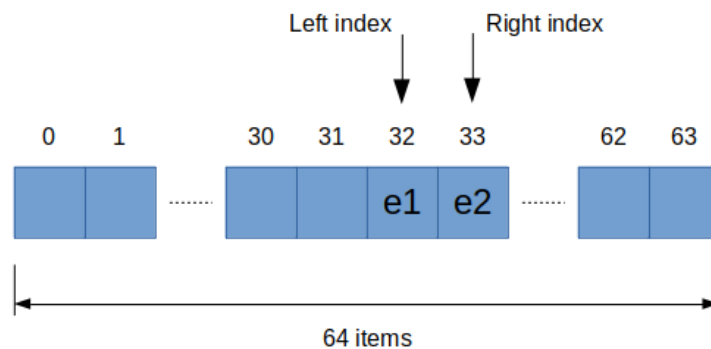


Figura 2.10: *append()* di e1 ed e2.

La seguente funzione C viene chiamata internamente, richiede l'oggetto deque e l'oggetto da aggiungere, la lunghezza della coda viene aumentata di 1.

```

1 static inline int deque_append_internal(dequeobject *deque,
    PyObject *item, ...) {
2     ...
3     Py_SET_SIZE(deque, Py_SIZE(deque) + 1);
4     deque->rightindex++;
5     deque->rightblock->data[deque->rightindex] = item;
6     ...
7 }

```

Aggiungendo ora un insieme di oggetti, da 'e3' fino a 'e32'. L'indice destro si posizionerà sulla cella 63 (l'ultima) contenente 'e32'. Andando ora ad aggiungere un ulteriore elemento 'e33', un nuovo blocco di 64 celle verrà allocato. L'indice di destra andrà a puntare sulla prima cella del nuovo blocco.

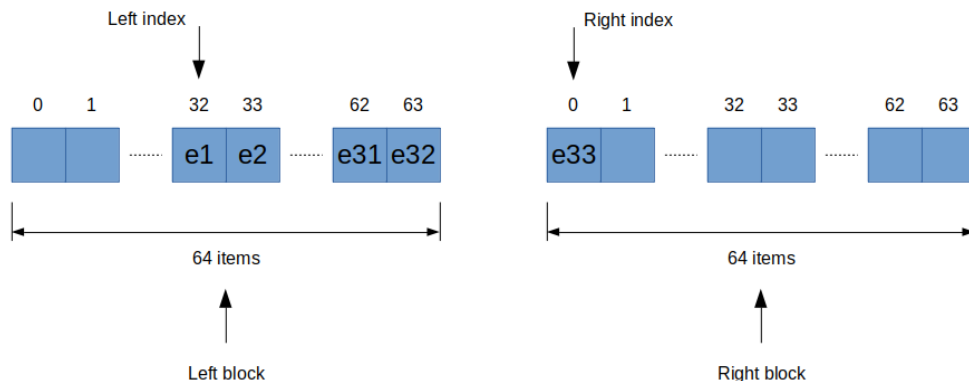


Figura 2.11: append() di 'e3' fino a 'e33'

La stessa funzione C interna, vista sopra, viene richiamata, ma con del codice aggiuntivo per poter allocare un altro blocco, aggiornare i links al blocco e il puntatore dx dell'oggetto deque.

```
1  static inline int deque_append_internal(dequeobject *deque,
    PyObject *item, Py_ssize_t maxlen){
2      if (deque->rightindex == BLOCKLEN - 1) {
3          block *b = newblock();
4          if (b == NULL)
5              return -1;
6          b->leftlink = deque->rightblock;
7          deque->rightblock->rightlink = b;
8          deque->rightblock = b;
9          deque->rightindex = -1;
10     }
11     Py_SET_SIZE(deque, Py_SIZE(deque) + 1);
12     deque->rightindex++;
13     deque->rightblock->data[deque->rightindex] = item;
14     ...
15 }
```

Vediamo ora come si comporta la struttura dati chiamando il metodo *popleft()* che rimuove il primo oggetto nella coda e lo ritorna, nel nostro esempio 'e1'. il puntatore sinistro è spostato in avanti di 1, alla posizione 33.

```
1  static PyObject *deque_popleft(dequeobject *deque, PyObject
    *unused){
2      PyObject *item;
3      ...
4      item = deque->leftblock->data[deque->leftindex];
5      deque->leftindex++;
6      Py_SET_SIZE(deque, Py_SIZE(deque) - 1);
7      ...
8      return item;
9  }
```

Continuando a rimuovere gli elementi con *popleft()* fino all'elemento 'e32'. L'indice *sx* ora vale 64 che è la lunghezza del blocco. Il blocco di sinistra può essere deallocato per risparmiare memoria e l'indice *sx* viene impostato a 0. I puntatori dei 2 blocchi punteranno entrambi allo stesso blocco.

```

1 static PyObject *deque_popleft(dequeobject *deque, PyObject
    *unused){
2     ...
3     if (deque->leftindex == BLOCKLEN) {
4         if (Py_SIZE(deque)) {
5             prevblock = deque->leftblock->rightlink;
6             freeblock(deque->leftblock);
7             deque->leftblock = prevblock;
8             deque->leftindex = 0;
9         }
10        ...
11    }
12    ...
13 }

```

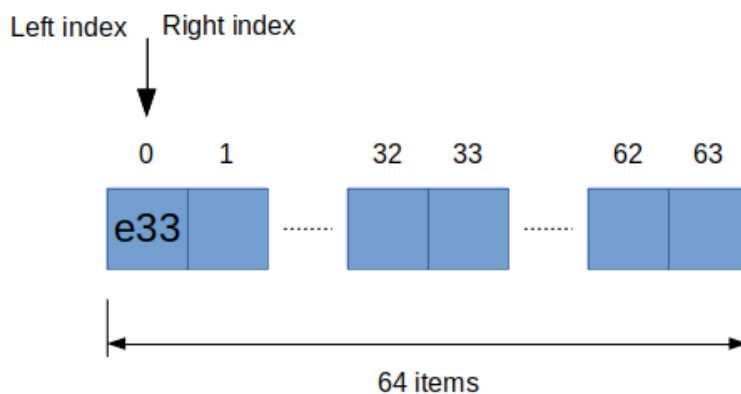


Figura 2.12: 'e33' unico elemento rimasto nella coda.

### 2.2.3 Array

Nativamente, il linguaggio Python non mette a disposizione strutture dati simili agli array visti in altri linguaggi come il C, ma esistono moduli esterni, tra cui *numpy* e *array*, che permettono di implementare tali strutture. Un array è una struttura dati allocata staticamente di lunghezza prefissata, contenente tutti dati dello stesso tipo (è infatti possibile definire un tipo qui). Nel nostro caso questo tipo di struttura ci torna utile perché lavorando con una grossa quantità di dati, è possibile ottimizzare l'uso della memoria in maniera più efficiente di quanto si potrebbe fare usando le liste. Vediamo ora un confronto tra le 2 possibili scelte.

```
1      # Inizializzazione degli array con 2000 celle
2      # Il primo con il modulo array, 'npArray' tramite numpy e
      'l' una semplice lista
3
4      array = arr.array('H', [0] * 2000)
5      npArray = np.array([1] * 2000, dtype=np.uint16)
6      l = [2] * 2000
```

**Osservazione:** Il parametro "H" per il modulo array e "dtype = np.uint16" per numpy indicano il tipo di dato contenuto nel vettore. In questo caso entrambi indicano un intero senza segno di 16 bit, il quale permette di rappresentare valori compresi tra 0 a 65.536. Nell'elaborato è stato scelto proprio questo formato, per cercar di ottimizzare il consumo di memoria, in relazione al numero di colonne della matrice e del numero di elementi del dominio. Questo significa l'imposizione di un vincolo allo script: in linea teorica non è possibile elaborare un problema con un dominio di oltre 65.536 elementi. Le matrici fornite hanno un numero di colonne decisamente inferiore al limite massimo che può essere trattato, per cui questo limite non è stato vincolante per le analisi effettuate.

```

1      # Calcolo dell'occupazione di memoria con la funzione
      introdotta nel capitolo 1
2
3      print(actual_size(array))
4      >>> 4064 B
5      print(actual_size(npArray))
6      >>> 4104
7      print(actual_size(l))
8      >>> 16100 B

```

```

1  def do(arr):
2      start_time = process_time()
3      for cell in arr:
4          if cell == 0:
5              cell = 12
6          elif cell == 2:
7              cell = 20
8          else:
9              cell = 10
10         time = process_time()-start_time
11         return time
12
13     # Calcolo delle prestazioni eseguendo una semplice
      sostituzione tra i valori
14
15     print(do(array))
16     >>> 0.00010160599999942121 s
17     print(do(npArray))
18     >>> 0.0028904910000004946 s
19     print(do(l))
20     >>> 0.0001552180000032876 s

```

```

1  # Append su una lista
2
3  sTime = process_time()
4  l.append(40)
5  ftime = process_time()-stime
6  print(ftime)
7  >>> 6.262299999981735e-05 s
8
9  # Append su un array
10
11  stime = process_time()
12  array.append(5)
13
14  ftime = process_time()-stime

```

```

15 print(ftime)
16 >>> 6.963200000065228e-05 s
17
18 # Append su un np array
19
20 stime = process_time()
21 np.append(n, 23)
22 ftime = process_time()-stime
23 print(ftime)
24 >>> 0.0001551270000001992 s
25
26 # Un append su un array vuol dire istanziare un nuovo array

```

Nel codice precedente è stato mostrato un confronto tra le prestazioni delle strutture dati prese in esame; nel primo caso vediamo come *array* sia quella che permette la minore occupazione di memoria. Nel secondo confronto si è testata una semplice sostituzione di valori, e anche in questo caso *array* è risultata la migliore in termini di tempo. Infine, è stato testato il tempo con il quale un nuovo elemento viene aggiunto in coda. In quest'ultimo la migliore è la lista (vantaggio esiguo rispetto ad *array*) grazie alla sua dinamicità, ed in genere un *array* non è pensato per questo tipo di operazioni, perché significa andare a ri-allocare un nuovo *array* di dimensione più grande. In virtù di questi risultati abbiamo scelto il modulo *array* per la creazione di vettori. Nello script sono stati utilizzati per andare a contenere prima di tutto le elementi delle righe della matrice principale (0, 1), dove abbiamo utilizzato il tipo di dato 'B' che sta per unsigned char. Successivamente per i vari vettori rappresentativi e gli insiemi generati dallo script abbiamo usato il tipo 'H' (2 byte) discusso sopra.

**Osservazione:** Eseguendo lo script passando come input la stessa matrice ma utilizzando per primo una versione che implementa esclusivamente *array* numpy, e l'altro esclusivamente il modulo *array*, si è constatato come il primo, in termini temporali, impieghi quasi il doppio del tempo del secondo per giungere a terminazione.

# Capitolo 3

## L'algoritmo MBase

In questo capitolo vengono discusse 2 implementazioni dell'algoritmo MBase, che differiscono nel modo in cui viene gestita la memoria. Inizialmente era stato adottato l'approccio per il quale si memorizzavano il vettore rappresentativo ed il suo insieme di riferimento, ma visti i limiti di gestione della memoria e delle strutture dati imposte dal linguaggio, e visti i risultati dei test fatti, abbiamo deciso di non memorizzare i vettori rappresentativi. Durante l'esecuzione dei test su alcune matrici si è constatato che l'occupazione di memoria è il principale fattore che impedisce di trattare matrici di dimensioni più grandi. Per ovviare a questo problema si è lavorato quindi sul modo in cui gli insiemi vengono rappresentati internamente durante l'esecuzione. Questo ha portato alla realizzazione della seconda implementazione dell'algoritmo, che viene discussa alla fine di questo capitolo.

### 3.1 Massimo di un insieme e successore di un elemento

La funzione *max\_insieme* riceve in ingresso l'insieme del quale si vuol ricavare l'elemento massimo, secondo l'ordine lessicografico. L'insieme è composto da interi e, per costruzione, risulta essere ordinato. Da ciò segue che il valore massimo sarà sempre l'ultimo. Nel caso in cui fosse vuoto, e questo si verifica durante la prima iterazione del ciclo while dell'algoritmo base, viene restituito il valore -1, che nella convenzione adottata rappresenta il valore  $\epsilon_{min}$ .



```

1 def max_insieme(insieme):
2     if not insieme:
3         return -1
4     else:
5         return insieme[-1]

```

**Complessità temporale**  $T(m) = \mathcal{O}(1)$

**Complessità spaziale**  $S(m) = \Theta(m)$ , la dimensione massima di un insieme

La funzione *succ* serve per calcolare il successore di un elemento, secondo l'ordine lessicografico introdotto. Nel nostro caso, ogni elemento della matrice è rappresentato con un intero da 1 a M, dove M rappresenta il numero di colonne della matrice. Per cui il successore di un elemento è l'elemento stesso + 1. Due casi particolari sono degni di attenzione: il primo è il caso in cui l'elemento di cui si vuole calcolare il successore è  $\epsilon_{min}$ , ovvero -1. La funzione *succ* in questo caso restituisce il valore 1. L'altro caso è quello del valore M, il massimo secondo l'ordine lessicografico. In questo caso, il valore restituito è M + 1, il rappresentante di  $\epsilon_{max}$ .

```

1 def succ(elem):
2     if elem == -1:
3         return 1
4     else:
5         return elem + 1

```

**Complessità temporale**  $T(m) = \mathcal{O}(1)$

**Complessità spaziale**  $S(m) = \Theta(1)$  elem è sempre un intero

## 3.2 Algoritmo base

L'algoritmo MBase è stato implementato seguendo lo pseudocodice fornito nella specifica dell'elaborato. Per contenere tutti i MHS trovati si è usata una lista, chiamata *lista\_mhs*. La variabile *coda* è l'oggetto di tipo deque utilizzato per contenere tutti gli insiemi che sono 'ok', *valore\_max* è l'ultimo elemento del dominio, secondo l'ordine lessicografico adottato, quindi il numero di colonne della matrice (ottenuto calcolando la lunghezza della prima riga della matrice), *break\_program* è una variabile globale che viene utilizzata per interrompere l'esecuzione del programma quando l'utente preme un tasto. La funzione *on\_press* andrà a modificare il valore di questa variabile da False a True nel momento in cui il tasto viene premuto, e a quel punto l'esecuzione dell'algoritmo termina.

```
1 def alg_base(matrice, dominio):
2     global break_program
3     start_time = process_time()
4     n_iter = 0
5     lista_mhs = []
6     coda = deque()
7     valore_max = len(matrice[0])
8     coda.append(arr.array('H', []))
9     with keyboard.Listener(on_press=on_press) as listener:
10         while coda and not break_program:
11             insieme = coda.popleft()
12             m = max_insieme(insieme)
13             e = succ(m)
14
15             for elem in range(e, valore_max + 1):
16                 n_iter += 1
17                 nuovo_insieme = insieme[:]
18                 nuovo_insieme.append(elem)
19                 vett_rapp = crea_vett_rapp(nuovo_insieme,
20 matrice)
21                 result = check(nuovo_insieme, vett_rapp)
22                 if result == "ok" and elem != valore_max:
23                     coda.append(nuovo_insieme)
24                 elif result == "mhs":
25                     lista_mhs.append(nuovo_insieme)
26             output(lista_mhs, dominio)
27     return lista_mhs, process_time() - start_time, n_iter
```

### 3.2.1 Calcolo vettore rappresentativo

La funzione `crea_vett_rapp` ha il compito di calcolare il vettore rappresentativo dell'*insieme* in input. Inizialmente, viene creato un array rappresentativo dalla variabile `vett_rapp` con dimensione 'numero di righe della matrice', riempito di 0. Si procede quindi a calcolare ogni valore del vettore rappresentativo, cella per cella (il ciclo `for` più esterno). Per ogni elemento dell'*insieme* di cui si vuole calcolare il vettore rappresentativo si va a verificare se la cella della matrice è 1 (l'elemento colpisce l'*insieme* indicato dalla riga *i*), e se lo è, verifica in `vett_rapp` esistenza di un valore diverso da '. Se il valore è diverso da 0 sovrascrive la cella di `vett_rapp` con 65535, l'ultimo intero rappresentabile su 2 byte (equivalente di 'x' della specifica), perché ci sono almeno due elementi che colpiscono l'*insieme*. Quindi si interrompe il ciclo `for` più interno, dato che il valore della cella non cambierà più. Altrimenti, se il valore della cella fosse 0, ciò significa che al momento l'elemento *j* è l'unico a colpire la riga *i*, per cui viene scritto il valore *j* nella cella del vettore rappresentativo. Una volta terminato il ciclo `for` più esterno si restituisce il vettore rappresentativo così calcolato.

```
1 def crea_vett_rapp(insieme, matrice):
2     vett_rapp = arr.array('H', [0] * len(matrice))
3     for i in range(len(matrice)):
4         for j in insieme:
5             if matrice[i][j - 1]:
6                 if vett_rapp[i]:
7                     vett_rapp[i] = 65535
8                     break
9                 else:
10                    vett_rapp[i] = j
11     return vett_rapp
```

**Complessità temporale**  $T(n, m) = \Theta(n * m)$  *n* è la dimensione di '*insieme*', *m* è il numero di righe di '*A*'.

**Complessità spaziale**  $S(n, m) = \Theta(n + (m * o))$  *m* \* *o*, dimensione della matrice '*A*'.

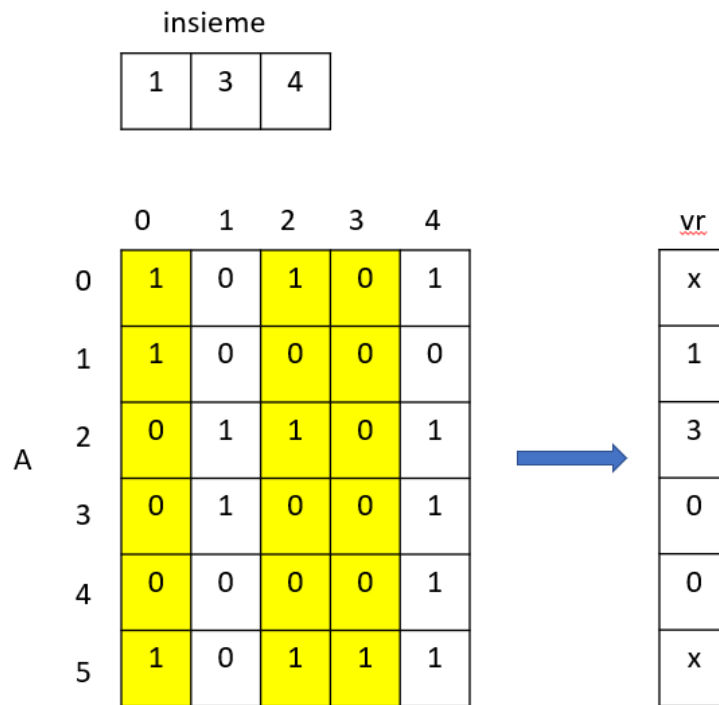


Figura 3.1: Rappresentazione grafica del calcolo di un vettore rappresentativo, in giallo sono evidenziate le colonne d'interesse per il calcolo considerato.

### 3.2.2 La funzione check

La funzione *check* va a stabilire se l'insieme passato sia o no un MHS o se lo potrebbe essere esteso per diventarlo. Il controllo è stato fatto seguendo le regole fornite nella specifica dell'elaborato. Il vettore rappresentativo in input (indicato qui con *vettore\_rapp*), potrebbe contenere molti '0' o molti '65535' (x). Per ridurre il tempo richiesto dal primo ciclo for, esso viene trasformato in un 'set', che al più contiene tutti gli elementi dell'insieme (chiamato nuovo\_insieme) e i valori 0 e 'x' (un set non contiene valori duplicati). Dopo di che si procede a verificare se tutti gli elementi di nuovo\_insieme sono contenuti nell'insieme ottenuto prima. Se anche un solo elemento non è presente si restituisce immediatamente "ko", perché la proiezione del vettore rappresentativo è diversa dall'insieme considerato. Successivamente si verifica se '0' sia presente o meno nella *proiezione* ed in caso affermativo si restituisce "ok", altrimenti "mhs".

```
1 def check(nuovo_insieme, vett_rapp):
2     proiezione = set(vett_rapp)
3     for i in nuovo_insieme:
4         if i not in proiezione:
5             return "ko"
6
7     if 0 in proiezione:
8         return "ok"
9     else:
10        return "mhs"
```

**Complessità temporale**  $T(n, m) = \mathcal{O}(n * m)$  n è la dimensione di 'nuovo insieme', m è quella di 'vettore'.

**Complessità spaziale**  $S(m) = \Theta(n + m)$

### 3.2.3 Output

La funzione *output* va a rimappare i valori dei MHS trovati nei valori di posizione, accorgimento discusso nel capitolo 2. I valori contenuti in *lista\_mhs* possono andare da 1 a 'num di colonne di A', con il remap vengono scalati adattandosi al dominio del problema che va da 0 a numero di colonne di A -1. La funzione è stata realizzata in questo modo per gestire anche eventuali operazioni di pre-elaborazione, al fine di garantire che sia possibile restituire i MHS calcolati con lo stesso formato della matrice di input.

```
1 def output(lista_mhs, dominio):  
2     for mhs in lista_mhs:  
3         for i in range(len(mhs)):  
4             mhs[i] = dominio[mhs[i] - 1]
```

**Complessità temporale**  $T(n, m) = \mathcal{O}(n * m)$  n è la dimensione di 'lista\_mhs', m è quella di 'mhs'.

**Complessità spaziale**  $S(n, d) = \Theta(n + d)$  d dimensione di 'dominio'

## 3.3 Accorgimenti sull'uso della memoria

Come accennato all'inizio di questo capitolo, la prima implementazione dell'algoritmo comporta un'occupazione di memoria notevole in molti casi, anche con matrici relativamente piccole, risultando quindi un fattore limitante per la grandezza delle matrici che possono essere trattate. È stata quindi cercata una soluzione migliore per gestire la memoria.

### 3.3.1 Overhead di memorizzazione

Nel capitolo 2 sono stati mostrati i motivi che hanno portato alla scelta di array come la struttura dati da utilizzare per memorizzare gli insiemi. Analizzando però un caso reale di array che la nostra applicazione si trova ad affrontare, emergono ulteriori dettagli:

```
1 print(actual_size(arr.array('H', [])))
2 >>> 64 B
3 print(actual_size(arr.array('H', [1, 2, 3, 4, 5])))
4 >>> 74 B
```

Se prendiamo un array di lunghezza 5, si ha una memoria occupata di 74 byte, di cui però solo 10 costituiscono effettivamente dei dati. 64 byte costituiscono l'overhead di memorizzazione, necessario per la gestione dell'oggetto array creato.

Guardando il numero di iterazioni (il numero di insiemi valutati come possibili MHS) si è visto come questo numero fosse decisamente elevato, in alcuni casi vengono valutati anche decine di miliardi di insiemi. È ragionevole quindi pensare che, in un dato momento, in coda siano presenti almeno centinaia di migliaia di insiemi, i quali però contengono un numero di elementi piccolo, per le matrici testate siamo attorno ai 5-10 elementi. Il risultato è che la maggior parte dell'occupazione della memoria non è dovuta alla memorizzazione di dati, ma a memorizzare le strutture per gestire gli stessi. Occorre quindi agire su questo aspetto per ottenere un miglioramento dell'efficienza spaziale dell'algoritmo.

### 3.3.2 La classe MultiInsieme

La classe *MultiInsieme* viene introdotta per gestire meglio l'occupazione di memoria dovuta agli insiemi. Essa si basa sull'assunzione, verificata, che l'algoritmo generi sempre dei MHS candidati di cardinalità non-decrescente. Quindi vengono generati prima tutti i possibili MHS di cardinalità 1, poi tutti quelli di cardinalità 2 e così via. La classe *MultiInsieme* va ad introdurre un livello di aggregazione tra insiemi basato sulla loro cardinalità: tutti gli insiemi di una data cardinalità  $n$  vengono memorizzati insieme utilizzando  $n$  array di lunghezza  $m$ , il numero di insiemi. Se rappresentiamo questi  $n$  array come una matrice di  $n$  righe e  $m$  colonne, allora sulle colonne abbiamo i singoli insiemi, e quindi è possibile ottenere un insieme dato l'indice. Una volta che un insieme "ok" o "mhs" viene generato non lo si aggiunge immediatamente alla coda oppure alla lista di MHS, ma si memorizza in una lista l'elemento che, aggiunto ad un insieme precedentemente generato, ci dà l'insieme considerato. L'aggiunta alla coda avviene in seguito, quando tutte le possibili estensioni di un dato insieme sono state considerate. L'operazione fatta è quella di copiare l'elemento di indice  $i$ , dell'insieme di partenza, nell'array di indice  $i$  dell'oggetto *MultiInsieme* e poi di copiare nell'ultimo array l'elemento aggiunto. Questa operazione viene ripetuta per ogni elemento che abbiamo aggiunto durante l'iterazione del ciclo `for` più interno, ovvero per ogni insieme che abbiamo generato.

MultiInsieme prima				insieme da estendere		
1	1	1	1	2	3	5
4	4	4	4			
6	6	6	6			
9	12	15	16			

elementi da aggiungere		
8	13	21

Figura 3.2: MultiInsieme prima dell'inserimento di un nuovo gruppo di insiemi



MultiInsieme dopo

1	1	1	1	2	2	2
4	4	4	4	3	3	3
6	6	6	6	5	5	5
9	12	15	16	8	13	21

Figura 3.3: MultiInsieme dopo dell'inserimento di un nuovo gruppo di insiemi

Una volta ottenuto il MultiInsieme di cardinalità  $i$ , noi lo andiamo ad aggiungere alla coda, ed estraiamo il prossimo. Per ottenere l'insieme da estendere dobbiamo estrarlo dal MultiInsieme considerato: andiamo ad effettuare un ciclo dove andiamo a considerare tutti gli indici da 0 fino al numero di insiemi contenuti - 1, e per ognuno di questo indice andiamo ad estrarre l'insieme. A questo punto l'esecuzione dell'algoritmo procede normalmente.

```

1 class MultiInsieme:
2
3     def __init__(self, cardinalita):
4         if cardinalita == 0:
5             self.lista_array = []
6         else:
7             self.lista_array = [arr.array('H', []) for _ in
range(cardinalita)]
8         self.cardinalita = cardinalita
9
10    def __len__(self):
11        if not self.lista_array:
12            return 0
13        return len(self.lista_array[0])
14
15    def get(self, indice):
16        if not self.lista_array:
17            return arr.array('H', [])

```

```

18         return arr.array('H', [self.lista_array[i][indice]
19         for i in range(self.cardinalita)])
20
21     def aggiungi_insieme(self, base, elementi):
22         if not len(base):
23             self.lista_array[0].extend(elementi)
24         else:
25             for i in range(len(base)):
26                 self.lista_array[i].extend(arr.array('H', [
27                 base[i] * len(elementi)]))
28             self.lista_array[len(base)].extend(elementi)

```

L'introduzione di questa classe permette di ridurre notevolmente l'occupazione di memoria. Per dare un riscontro numerico approssimativo del miglioramento, si considerino 10000 insiemi di cardinalità 5. Utilizzando l'approccio usato finora, per memorizzare ognuno di questi insiemi sono richiesti 74 byte ( $64 + 5 * 2$ ), per un totale di 740000 byte, mentre utilizzando la classe MultiInsieme si ha:  $5 * (64 + 10000 * 2) = 100320$  byte, una riduzione di più di 7 volte della memoria utilizzata in questo esempio. La riduzione può essere meno significativa per le matrici più piccole, visto che dipende dal numero di insiemi considerati, ma per le matrici più grandi può fare una notevole differenza. Di seguito è riportato il codice relativo all'implementazione modificata dell'algoritmo base.

```

1 def alg_base(matrice, dominio):
2     global break_program
3     start_time = process_time()
4     n_iter = 0
5     lista_mhs = []
6     coda = deque()
7     valore_max = len(matrice[0])
8     coda.append(MultiInsieme(0))
9     with keyboard.Listener(on_press=on_press) as listener:
10         while coda and not break_program:
11             multi_insieme = coda.popleft()
12             multi_insieme_ok = MultiInsieme(multi_insieme.
cardinalita + 1)
13             multi_insieme_mhs = MultiInsieme(multi_insieme.
cardinalita + 1)
14             len_multi_insieme = len(multi_insieme)
15             if not len_multi_insieme:
16                 len_multi_insieme += 1
17             for i in range(len_multi_insieme):
18                 insieme = multi_insieme.get(i)
19                 m = max_insieme(insieme)
20                 e = succ(m)
21                 ok_array = arr.array('H', [])
22                 mhs_array = arr.array('H', [])
23                 for elem in range(e, valore_max + 1):
24                     n_iter += 1
25                     vett_rapp = crea_vett_rapp(insieme, elem,
matrice)
26                     result = check(insieme, elem, vett_rapp)
27                     if result == "ok" and elem != valore_max:
28                         ok_array.append(elem)
29                     elif result == "mhs":
30                         mhs_array.append(elem)
31                     if ok_array:
32                         multi_insieme_ok.aggiungi_insiemi(insieme
, ok_array)
33                     if mhs_array:
34                         multi_insieme_mhs.aggiungi_insiemi(
insieme, mhs_array)
35                     if len(multi_insieme_ok):
36                         coda.append(multi_insieme_ok)
37                     if len(multi_insieme_mhs):
38                         lista_mhs.append(multi_insieme_mhs)
39     output(lista_mhs, dominio)
40     return lista_mhs, process_time() - start_time, n_iter

```

Qui invece è riportato il codice modificato delle funzioni ausiliarie. Le considerazioni fatte in precedenza rimangono comunque valide.

```
1 def check(insieme, elem, vett_rapp):
2     proiezione = set(vett_rapp)
3     if elem not in proiezione:
4         return "ko"
5
6     for i in insieme:
7         if i not in proiezione:
8             return "ko"
9
10    if 0 in proiezione:
11        return "ok"
12    else:
13        return "mhs"

1 def crea_vett_rapp(insieme, elem, matrice):
2     vett_rapp = arr.array('H', [matrice[i][elem - 1] * elem
3     for i in range(len(matrice))])
4     for i in range(len(matrice)):
5         for j in insieme:
6             if matrice[i][j - 1]:
7                 if vett_rapp[i]:
8                     vett_rapp[i] = 65535
9                     break
10                else:
11                    vett_rapp[i] = j
12    return vett_rapp

1 def output(lista_mhs, dominio):
2     for multi_mhs in lista_mhs:
3         for riga in multi_mhs.lista_array:
4             for i in range(len(riga)):
5                 riga[i] = dominio[riga[i] - 1]
```

# Capitolo 4

## Funzioni di pre-elaborazione

In questo capitolo viene illustrata l'implementazione delle funzioni di pre-elaborazione.

### 4.1 Rimozione delle righe

La funzione `togli_righe` effettua la prima operazione di pre-elaborazione richiesta, rimuovendo tutte le righe che ne contengono un'altra. Per fare ciò si utilizzano due funzioni ausiliarie, `contiene` e `costruisci_array`.

#### 4.1.1 Contiene

Questa funzione riceve in ingresso la matrice su cui si sta effettuando la pre-elaborazione (indicata come *matrice* nel codice) e due indici,  $i$  e  $j$ , che rappresentano le righe considerate al momento dell'invocazione della funzione.

La funzione restituisce come output `True` o `False`: `True` se la riga  $i$  contiene la riga  $j$ , dove con contiene si intende anche il caso in cui le righe rappresentano lo stesso insieme, quindi contengono gli stessi elementi, `False` altrimenti.

Il controllo viene fatto elemento per elemento, ovvero viene fatto un ciclo in cui si confrontano gli elementi in posizione  $k$  di entrambe le righe: se  $A[i][k]$  è uguale a zero (l'elemento di indice  $k$  non è presente nell'insieme di indice  $i$ ) e  $A[j][k]$  è uguale a uno (presenza dell'elemento), allora la riga  $i$  NON contiene la riga  $j$ , perché l'insieme rappresentato da quest'ultima contiene un elemento che non è presente nell'insieme di indice  $i$ . In tutti gli altri casi, la riga

$i$  può ancora contenere la riga  $j$ , quindi è necessario considerare l'elemento successivo.

Se il ciclo termina senza restituire False, allora tutti gli elementi contenuti nella riga  $j$  sono anche contenuti nella riga  $i$ , quindi viene restituito il valore True.

La complessità temporale di questa funzione risulta essere:  $\mathcal{O}(m)$ , dove  $m$  = numero di colonne. Questo perché al più vengono eseguite  $m$  iterazioni del ciclo for nel caso in cui la riga contenga la riga  $j$ , ovvero vengono confrontati tutti gli elementi delle due righe, che sono pari al numero di colonne della matrice, quindi  $m$ .

```
1 def contiene(matrice, i, j):
2     # len(matrice[0]) = numero di colonne
3     for k in range(len(matrice[0])):
4         if (not matrice[i][k]) and matrice[j][k]:
5             return False
6     return True
```

**Complessità temporale**  $T(m) = \mathcal{O}(m)$

**Complessità spaziale**  $S(n, m) = \mathcal{O}(1)$  dovuta al record di attivazione della funzione. Non è inclusa la complessità spaziale dovuta alla memorizzazione della matrice,  $\mathcal{O}(n * m)$

### 4.1.2 Costruisci\_array

La funzione costruisci\_array viene utilizzata per ordinare gli indici delle righe in base al numero di elementi contenuti nelle stesse. Per calcolare il numero di elementi contenuti in un insieme rappresentato dall'indice di riga  $i$ , è sufficiente calcolare la somma di tutti gli elementi della riga. I possibili valori per una cella della matrice sono 0 e 1, dove 1 indica la presenza di un elemento nell'insieme, 0 la sua assenza, quindi effettuando la somma si ottiene il numero di elementi contenuti nell'insieme.

Viene quindi costruito un array di  $n$  celle, dove  $n$  rappresenta il numero di righe, dove ogni cella contiene l'oggetto (il riferimento) che rappresenta la coppia di valori indice di riga e numero di elementi della stessa. Viene quindi poi utilizzato un algoritmo di ordinamento in loco che ordina questi oggetti in ordine decrescente, in base al valore del secondo campo (il numero di elementi della riga) e poi viene restituita l'array che rappresenta gli indici di

riga così ordinati. La complessità temporale di `costruisci_array` è ottenuta sommando diversi contributi:

- costruzione della lista: la somma di  $m$  elementi richiede  $m - 1$  operazioni di somma per cui  $\mathcal{O}(m)$ , e questa operazione viene ripetuta  $n$  volte, una per ogni riga, per cui si ha una complessità pari a  $\mathcal{O}(n * m)$
- ordinamento in loco della lista: eseguibile per esempio utilizzando heapsort, quindi  $\mathcal{O}(n \log n)$

```
1 def costruisci_array(matrice):
2     lista = []
3     for i in range(len(matrice)):
4         lista.append((sum(matrice[i]), i))
5     lista.sort(key=lambda x: x[0], reverse=True)
6     return arr.array('H', list(zip(*lista))[1])
```

**Complessità temporale**  $T(n, m) = \mathcal{O}(n * m + n \log n)$

**Complessità spaziale**  $S(n) = \mathcal{O}(n)$  in quanto è necessario memorizzare un array contenente gli indici delle righe, che sono  $n$ .

### 4.1.3 Togli\_righe

La funzione `togli_righe`, utilizzando le funzioni precedentemente descritte, effettua la rimozione delle righe della matrice che soddisfano la condizione richiesta. Gli indici delle righe vengono considerati nell'ordine dato dalla funzione `costruisci_array`. In questo modo è sufficiente confrontare la riga definita dall'indice `array[i]` (dove `array` rappresenta l'array costruito in precedenza con il metodo `costruisci_array`) con le righe indicate dagli indici successivi presenti nell'array.

Questo perché gli indici precedenti rappresentano delle righe che contengono un numero di elementi maggiore o uguale alla riga `array[i]`, quindi non è necessario effettuare alcuna operazione.

Se due righe contengono lo stesso numero di elementi, la funzione contiene verrà invocata una sola volta, in base all'ordinamento relativo che esiste tra gli indici delle due righe nell'array.

Per esempio se la riga  $i$  è uguale alla riga  $j$  e l'indice  $i$  precede l'indice  $j$  nella lista `l`, allora sarà effettuata la chiamata `contiene(A, i, j)`, ma non la chiamata `contiene(A, j, i)`.

Se la riga di indice `array[i]` contiene la riga di indice `array[j]` allora l'indice `array[i]` viene aggiunto ad una lista per andare poi ad effettuare la rimozione della riga indicata dalla matrice `A`. Altrimenti si procede considerando l'indice `array[j]` successivo. Il ciclo più esterno termina quando si sono considerati tutti gli indici presenti nell'array `array`. A quel punto si procede all'eliminazione delle righe indicate con un ciclo, fornendo in uscita il numero di righe rimosse e gli indici delle stesse.

La complessità temporale è la somma di alcuni contributi:

- il ciclo più esterno viene eseguito  $n$  volte
- il ciclo più interno viene eseguito al più  $n - i - 1$  volte
- il ciclo interno effettua una chiamata a `contiene`,  $\mathcal{O}(m)$
- nel caso in cui `contiene` restituisca `True`, viene aggiunto l'indice `l[i]` all'array, quindi  $\mathcal{O}(n)$
- l'operazione di rimozione di una riga ha complessità di  $\mathcal{O}(n)$  e viene eseguita al più  $n - 1$  volte, per cui  $\mathcal{O}(n^2)$

La complessità temporale risulta quindi essere  $\mathcal{O}(n^2(n + m))$



```

1 def toglirighe(matrice):
2     indici_rimossi = []
3     array = costruisci_array(matrice)
4     j = len(array)
5     for i in range(j):
6         for k in range(i + 1, j):
7             if contiene(matrice, array[i], array[k]):
8                 indici_rimossi.append(array[i])
9                 break
10
11     indici_rimossi.sort(reverse=True)
12     for i in indici_rimossi:
13         del matrice[i]
14     indici_rimossi = arr.array('H', indici_rimossi)
15     return indici_rimossi, len(indici_rimossi)

```

**Complessità temporale**  $T(n, m) = \mathcal{O}(n^2(n + m))$

**Complessità spaziale**  $S(n) = \mathcal{O}(n)$  dovuta all'occupazione dell'array ottenuto con la chiamata di costruisci\_array,  $\mathcal{O}(n)$  e l'array utilizzato per restituire gli indici da rimuovere,  $\mathcal{O}(N)$ .

## 4.2 Rimozione delle colonne

La funzione per la rimozione delle colonne è stata implementata usando una funzione ausiliare chiamata `colonna_di_zero`.

### 4.2.1 Colonna\_di\_zero

La funzione `colonna_di_zero` ha come input la matrice su cui si sta effettuando la pre-elaborazione e un indice  $i$  che rappresenta la colonna su cui si sta lavorando. Restituisce come output True o False: True se la colonna è composta solamente da zero, False se esiste almeno un uno nella colonna.

L'implementazione è molto semplice: si guardano tutti gli elementi della colonna  $i$ , se uno di questi è 1 allora la funzione termina immediatamente restituendo False, altrimenti si prosegue e il ciclo termina quando tutti gli elementi della colonna sono stati considerati. Viene quindi restituito il valore True.

La complessità temporale è  $\mathcal{O}(n)$ , dove  $n$  è il numero di righe della matrice, che rappresenta anche il numero di elementi di una colonna.

```

1 def colonna_di_zero(matrice, i):
2     for j in range(len(matrice)):
3         if matrice[j][i]:
4             return False
5     return True

```

**Complessità temporale**  $T(n) = \mathcal{O}(n)$  dove  $n$  è il numero di righe della matrice  $A$ , che rappresenta anche il numero di elementi di una colonna.

**Complessità spaziale**  $S(n) = \mathcal{O}(1)$  dovuta al record di attivazione della funzione.

### 4.2.2 Togli\_colonne

La funzione `togli_colonne` esegue un ciclo: per ogni colonna va a verificare se questa sia una colonna di zeri invocando l'apposita funzione e, nel caso in cui lo fosse, aggiunge l'indice di questa colonna ad un array, il quale verrà utilizzato poi per rimuovere tutte le colonne che soddisfano la condizione. Viene infine restituito l'array degli indici di colonne rimosse.

La complessità temporale è la somma di alcuni contributi:

- il ciclo viene eseguito  $m$  volte, una per ogni colonna
- ogni iterazione va ad invocare la funzione `colonna_di_zeri`,  $\mathcal{O}(n)$
- nel caso in cui `colonna_di_zeri` restituisca `True`, viene aggiunto l'indice  $i$  all'array, quindi  $\mathcal{O}(n)$
- l'operazione di rimozione di una colonna ha una complessità di  $\mathcal{O}(n*m)$ : per ogni riga vado a rimuovere l'elemento in posizione  $i$ , quindi  $\mathcal{O}(m)$ , e questa operazione deve essere fatta  $n$  volte, una per ogni riga
- l'operazione di rimozione di una colonna viene eseguita al più  $m - 1$  volte, per cui si ha  $\mathcal{O}(m^2 * n)$

Complessivamente si ha  $\mathcal{O}(m^2 * n)$

```

1 def togl_colonne(matrice):
2     indici_rimossi = []
3     for i in range(len(matrice[0]) - 1, -1, -1):
4         if colonna_di_zero(matrice, i):
5             indici_rimossi.append(i)
6     for i in indici_rimossi:
7         deque(map(lambda x: x.pop(i), matrice), maxlen=0)
8     indici_rimossi = arr.array('H', indici_rimossi)
9     return indici_rimossi, len(indici_rimossi)

```

**Complessità temporale**  $T(n, m) = \mathcal{O}(m^2 * n)$

**Complessità spaziale**  $S(m) = \mathcal{O}(m)$  dovuta al memorizzazione dell'array che contiene gli indici da rimuovere, che sono al più  $m - 1$

# Capitolo 5

## Sperimentazione

In questo capitolo viene spiegato il modo in cui si sono state analizzate le prestazioni del programma e i risultati ottenuti dall'applicazione dello stesso su una selezione delle matrici fornite.

### 5.1 Metodo di raccolta dati

Per misurare il tempo di esecuzione dell'algoritmo applicato ad una matrice è stato utilizzato il modulo *time*, presente nativamente in Python. All'inizio del codice della funzione *alg\_base* viene ottenuto, tramite la chiamata alla funzione *process\_time*, il timestamp che rappresenta l'inizio dell'esecuzione. Una volta terminata l'esecuzione dell'algoritmo (dopo aver trovato tutti i MHS oppure dopo che l'utente l'ha interrotta) viene ottenuto allo stesso modo il timestamp di fine esecuzione. Viene quindi restituita la differenza tra i due valori.

Per quanto riguarda l'occupazione di memoria registrata, si è utilizzato il modulo *memory-profiler*, che permette di monitorare l'uso di memoria di una funzione o un processo durante tutta la sua esecuzione. Nel nostro caso, è stato utilizzato per registrare la massima occupazione di memoria raggiunta durante l'esecuzione dell'algoritmo (con o senza pre-elaborazione) su una matrice. L'uso di questo modulo comporta un overhead in termini di tempo di esecuzione, dovuto al fatto che per misurare la memoria occupata viene creato un altro processo, il quale si occuperà ad intervalli regolari di richiedere al sistema l'occupazione di memoria del processo padre (che sta eseguendo l'algoritmo).

Durante la realizzazione dell'applicazione è stato notato che esecuzioni successive dell'algoritmo senza accorgimenti portavano a letture sbagliate della massima occupazione di memoria raggiunta. Pertanto, al fine di poter eseguire la versione base seguita da quella con pre-elaborazione, è stato utilizzato il modulo *multiprocessing* di Python. L'esecuzioni dell'algoritmo vengono eseguite in un processo separato, in modo tale che con l'invocazione della funzione *memory\_usage* si possa monitorare correttamente l'uso della memoria. Alla fine dell'esecuzione, le informazioni relative all'elaborazione, oltre alle informazioni sulle prestazioni, vengono passate al processo principale, il quale poi recupera le risorse allocate al processo figlio e procede con l'esecuzione dell'algoritmo preceduto da pre-elaborazione sulla stessa matrice, oppure procede ad una nuova esecuzione su un'altra matrice. In questo modo si è raggiunto un maggior grado di indipendenza tra le esecuzioni successive dell'algoritmo.

### 5.1.1 Formato dei dati raccolti

I dati raccolti dalle esecuzioni dell'algoritmo su varie matrici vengono salvate in un file .csv, costituito da varie colonne, in ordine:

- *nome\_matrice*
- *righe*
- *colonne*
- *esecuzione\_completata\_1*, valore booleano rappresentato come 0 nel caso in cui l'esecuzione dell'algoritmo base senza pre-elaborazione sia stata terminata dall'utente, 1 se sono stati generati tutti i MHS per la matrice in esame
- *tempo\_di\_esecuzione\_1*, il tempo di esecuzione richiesto dall'algoritmo base senza pre-elaborazione, espresso in secondi
- *n\_iter\_1*, il numero di iterazioni del ciclo for più interno, ovvero il numero di insiemi che sono stati valutati come possibili MHS
- *massima\_occupazione\_memoria\_1*, la massima occupazione di memoria registrata durante l'esecuzione dell'algoritmo senza pre-elaborazione, espresso in MiB

- *numero\_mhs\_trovati*
- *cardinalita\_minima*, la dimensione minima dei MHS trovati
- *cardinalita\_massima*, la dimensione massima dei MHS trovati
- *tempo\_pre*, il tempo richiesto dalle operazioni di pre-elaborazione
- *nuovo\_numero\_righe*, dopo l'esecuzione dell'operazione di pre-elaborazione toglì righe
- *nuovo\_numero\_colonne*, dopo l'esecuzione dell'operazione di pre-elaborazione toglì colonne
- *esecuzione\_completata\_2*, valore booleano rappresentato come 0 nel caso in cui l'esecuzione dell'algoritmo con pre-elaborazione sia stata terminata dall'utente, 1 se sono stati generati tutti i MHS per la matrice in esame
- *tempo\_di\_esecuzione\_2*, il tempo di esecuzione in secondi dell'algoritmo con pre-elaborazione
- *n\_iter\_2*, analogo di *n\_iter\_* per l'esecuzione con pre-elaborazione
- *massima\_occupazione\_memoria\_2*, la massima occupazione di memoria raggiunta durante l'esecuzione dell'algoritmo con pre-elaborazione
- *numermio\_mhs\_2*, il numero di MHS trovati dall'esecuzione con pre-elaborazione, se è avvenuta
- *cardinalita\_minima\_2*, la cardinalità minima dei MHS trovati durante l'esecuzione con pre-elaborazione
- *cardinalita\_massima\_2*, la cardinalità massima dei MHS trovati durante l'esecuzione con pre-elaborazione
- *risultati\_uguali*, valore booleano (rappresentato come 0 e 1) per indicare se i risultati ottenuti effettuando la pre-elaborazione e non sono uguali oppure no.

Eccezione fatta per le colonne `nome_matrice`, `righe` e `colonne`, tutte le altre possono presentare, in aggiunta ai valori specificati sopra, anche il valore `'?'`, il quale indica l'assenza di un valore. Ciò si verifica quando viene eseguita una sola delle due versioni dell'algoritmo, oppure quando l'utente interrompe l'esecuzione dell'algoritmo.

## 5.2 Limitazioni riscontrate

Durante l'esecuzione dell'algoritmo su varie matrici sono state riscontrate alcune limitazioni che hanno influenzato i risultati ottenuti, specialmente sul fronte della massima occupazione di memoria registrata.

### 5.2.1 Gestione dell'out of memory

L'interprete Python non prevede la possibilità di limitare la memoria utilizzata dall'applicazione. Quando è necessario creare un nuovo oggetto, l'interprete cercherà di allocare tutta la memoria necessaria per memorizzarlo e, qualora si accorgesse che la memoria rimasta non è sufficiente, l'interprete lancerà una eccezione *MemoryError*. In realtà questa eccezione non viene sempre lanciata, in alcuni casi l'interprete non realizza cosa sta accadendo dal punto di vista della memoria e l'esecuzione dell'applicazione si interrompe.

L'applicazione realizzata ricade nell'ultimo caso e quindi non è stato possibile segnalare per quali matrici l'esecuzione esaurisce la memoria RAM disponibile sulla macchina.

Un altro caso in cui la gestione della memoria diventa critica si ha quando bisogna effettuare i controlli di correttezza, confrontando i risultati ottenuti con l'esecuzione dell'algoritmo base con quello preceduto da pre-elaborazione. Per fare ciò è necessario memorizzare due volte l'insieme di MHS calcolati, con ovvie implicazioni sull'occupazione di memoria nel caso in cui il numero di MHS trovati è grande. Ciò viene evitato utilizzando salvando i risultati delle due esecuzioni dell'algoritmo su disco e facendo il controllo di correttezza su i due file così ottenuti (che contengono tutti i MHS calcolati). L'operazione viene fatta utilizzando il modulo *filecmp* di Python, che permette di confrontare il contenuto di due file per verificare se sono differenti o meno. L'operazione viene fatta in maniera efficiente andando ad utilizzare un buf-

fer per leggere il file (la cui grandezza è 8 KB), per cui non ci sono problemi anche nel caso in cui il numero di MHS trovati sia elevato.

### 5.2.2 Swap

L'applicazione è stata eseguita su una macchina personale del gruppo, le cui specifiche sono riportate nella sezione Risultati. Come visto nella sezione precedente, Python non ha un meccanismo per limitare l'uso della memoria, e quindi l'interprete andrà ad occupare tutta la memoria disponibile sulla macchina, anche quella della di swap. Questo ovviamente comporta un peggioramento delle prestazioni, vista la differente velocità di lettura e scrittura tra RAM e disco. Va anche notato che i moduli presenti in Python per il tracciamento dell'uso della memoria non consentono di monitorare l'uso della memoria di swap da parte di un singolo processo, forniscono solo informazioni relative all'utilizzo di memoria fisica del sistema. Disabilitare l'uso della memoria di swap può avere effetti deleteri sulla stabilità del sistema operativo, per cui si è scelto di non andare ad alterare le impostazioni. A fronte di ciò alcuni risultati (quelli relativi a matrici con un grande numero di MHS trovati oppure con un grande numero di candidati MHS valutati) potrebbero non essere corretti. Al di sotto di una certa soglia dell'occupazione di memoria (stimata in circa 5.5 GiB per la macchina utilizzata durante i test) i risultati invece dovrebbero essere corretti.



## 5.3 Risultati

I risultati che saranno mostrati in seguito sono stati ottenuti eseguendo l'applicazione su una macchina caratterizzata dalle seguenti specifiche:

- **RAM:** 8 GB
- **Processore:** Intel i5-10300H 2.50GHz (fino a 4.50 GHz con Turbo Boost)
- **Sistema operativo:** Windows 11 21H2

Inoltre per eseguire il codice .py è stato utilizzato l'ambiente PyPy, riguardo a questo sono disponibili più informazioni nella sezione PyPy del manuale utente. L'algoritmo (nella variante con e senza pre-elaborazione) è stato eseguito su 386 matrici, le cui dimensioni sono rappresentate graficamente nella prossima figura. I risultati ottenuti sono riferiti a quelle matrici per cui è stato possibile completare sia l'esecuzione dell'algoritmo base che quella con pre-elaborazione.

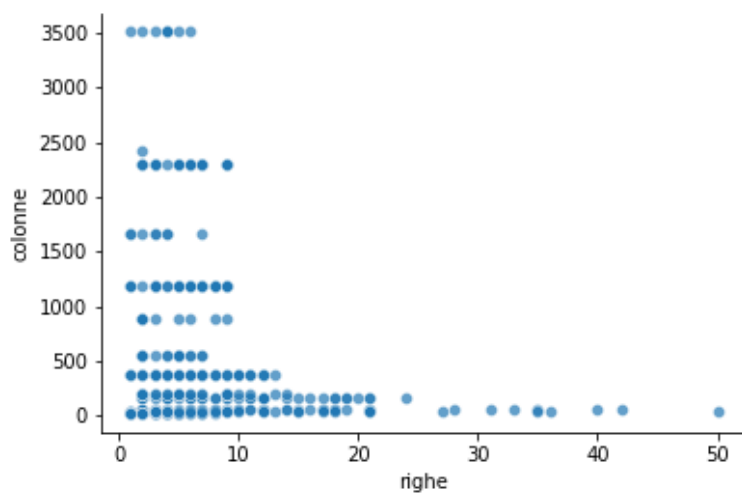


Figura 5.1: Dimensione delle matrici trattate

### 5.3.1 Prestazioni temporali

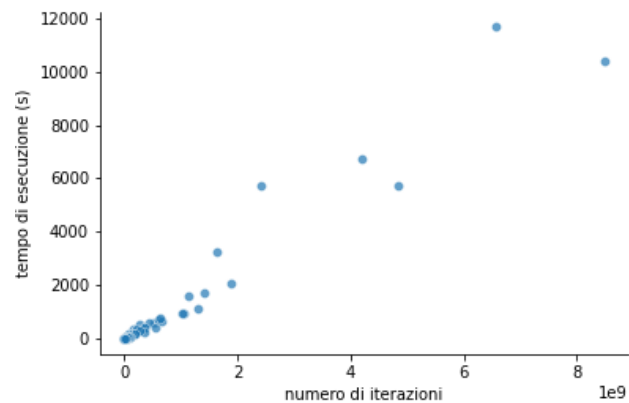


Figura 5.2: Tempo di esecuzione dell'algoritmo base in funzione del numero di iterazioni compiute

La figura sopra mostra i limiti dell'analisi compiuta, in quanto non sono presenti molte matrici che comportano un numero di iterazioni elevato. Si può comunque notare una correlazione tra l'aumento del numero di iterazioni (il numero di insiemi considerati come possibili MHS) e il tempo di esecuzione necessario. Concentrandosi su quelle matrici la cui esecuzione comporta un numero di iterazione inferiore a 1 miliardo, questa ipotesi sembra essere confermata.

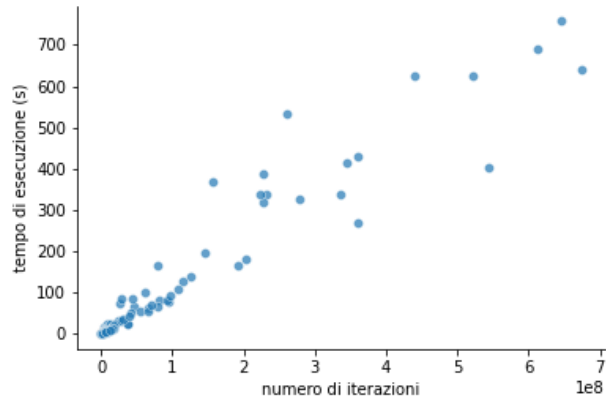


Figura 5.3: Tempo di esecuzione dell'algoritmo base in funzione del numero di iterazioni compiute, numero di iterazioni inferiore a 1 miliardo

Le operazioni di pre-elaborazione hanno un impatto positivo e notevole sulle prestazioni temporali dell'algoritmo. In alcuni casi si ha una riduzione del tempo di esecuzione dell'80-90 % rispetto all'esecuzione del solo algoritmo base, mentre le operazioni di pre-elaborazione stesse richiedono meno di 1 s in tutti i casi. Per esempio, per la matrice su cui l'esecuzione dell'algoritmo ha richiesto più tempo, c499.040, il tempo richiesto è passato da 11662.75 secondi a 2924.59375 secondi, una riduzione del 74.92 % del tempo richiesto.

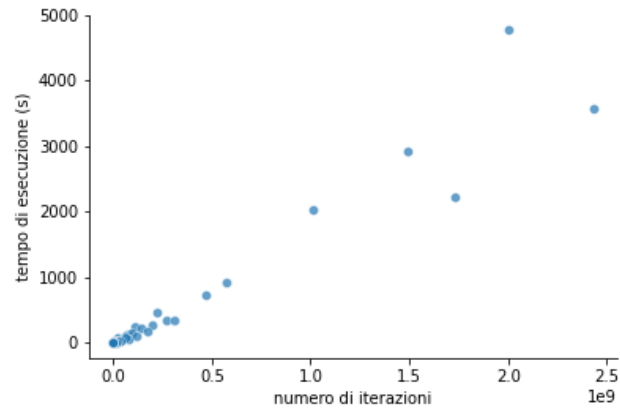


Figura 5.4: Tempo di esecuzione dell'algoritmo con pre-elaborazione in funzione del numero di iterazioni compiute

Come fatto nel caso dell'esecuzione dell'algoritmo base, nella figura successiva si restringe il numero di iterazioni, andando a considerare solo valori inferiori a 250 milioni. Questo mette in evidenza la correlazione che esiste tra il numero di iterazioni e il tempo di esecuzione.

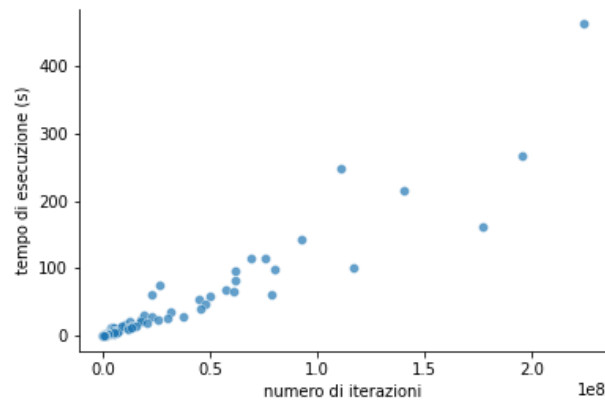


Figura 5.5: Tempo di esecuzione dell'algoritmo con pre-elaborazione in funzione del numero di iterazioni compiute, numero di iterazioni inferiore a 250 milioni

### 5.3.2 Prestazioni spaziali

Le prestazioni spaziali dell'implementazione dell'algoritmo base sono discrete. Nel caso di matrici che comportano un numero di iterazioni basso, l'occupazione di memoria rimane contenuta, mentre con valori più elevati si ha un rapido incremento della memoria utilizzata. La maggior parte delle matrici testate comporta un'occupazione di memoria limitata, al di sotto dei 1000 MiB, e risultano accomunate da un numero di iterazioni inferiore a 250 milioni. Ulteriori test andrebbero eseguiti per verificare l'occupazione di memoria quando il numero di iterazioni supera questo valore.

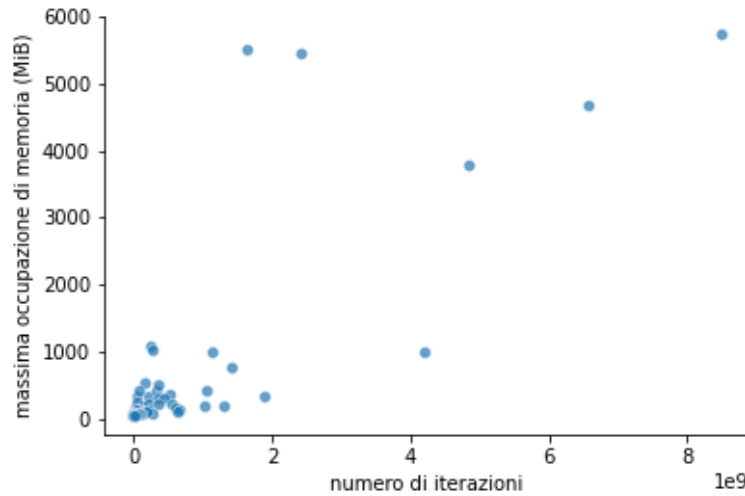


Figura 5.6: Massima occupazione di memoria durante l'esecuzione dell'algoritmo base

A differenza delle prestazioni temporali, l'esecuzione delle operazioni di pre-elaborazione non porta alcun miglioramento per quanto riguarda l'occupazione di memoria. L'operazione `togli_righe` potrebbe avere un impatto sull'occupazione di memoria, perché riducendo il numero di righe viene ridotto anche il numero di celle del vettore rappresentativo, ma la scelta di non memorizzare va già in tale direzione, per cui l'impatto è nullo.

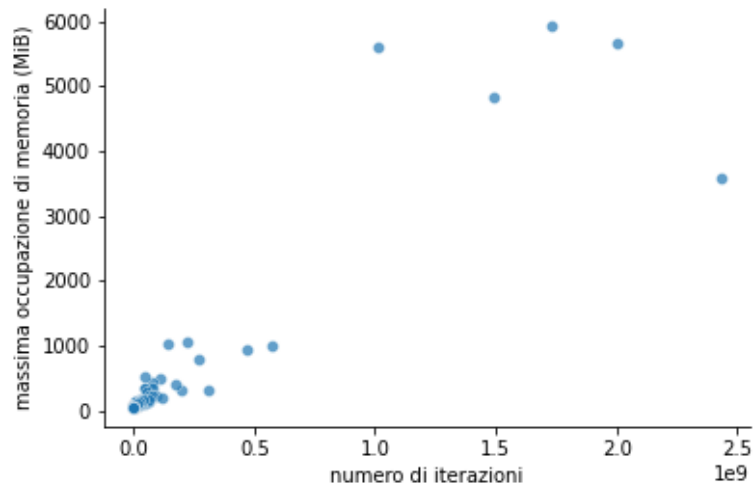


Figura 5.7: Massima occupazione di memoria durante l'esecuzione dell'algoritmo base preceduto da pre-elaborazione

### 5.3.3 Accorgimenti per la riduzione dell'uso della memoria

Nel capitolo 4, oltre all'implementazione normale dell'algoritmo base, è stata anche descritta una versione aggiuntiva, creata al fine di ridurre la massima occupazione di memoria. In questa sottosezione si analizza l'impatto delle modifiche implementate, su alcune matrici.

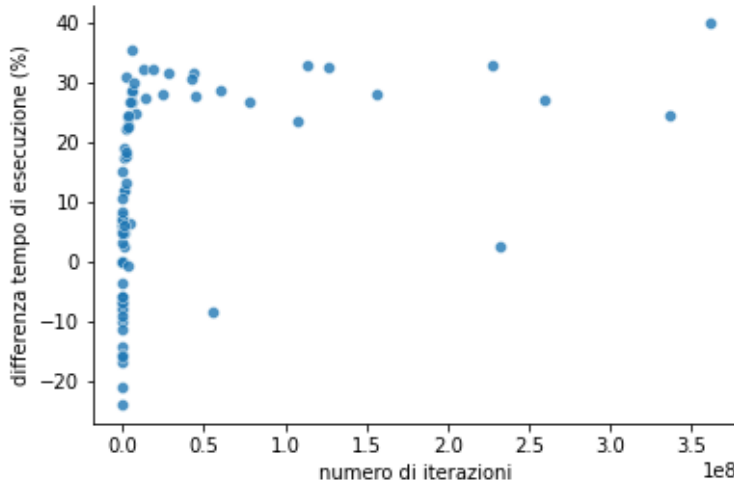


Figura 5.8: Differenza percentuale nel tempo di esecuzione rispetto alla prima versione dell'algoritmo, numero di iterazioni inferiore a 500 milioni. Valori negativi indicano un peggioramento delle prestazioni temporali

Il principale obiettivo delle modifiche introdotte è quello di ridurre l'uso della memoria, tuttavia ciò non deve avvenire a discapito delle prestazioni temporali. Nella figura sopra si nota come si riesca anche ad ottenere un miglioramento delle prestazioni temporali in alcuni casi. Per esempio, con la matrice c499.040 si ha una riduzione del tempo di esecuzione del 40.47 %, passando da 11662.75 s a 6942.796875 s. I peggioramenti delle prestazioni temporali invece si hanno con calcoli relativamente semplice (tempo di esecuzione richiesto inferiore al secondo), e risultano comunque tollerabili considerando il miglioramento delle prestazioni spaziali.

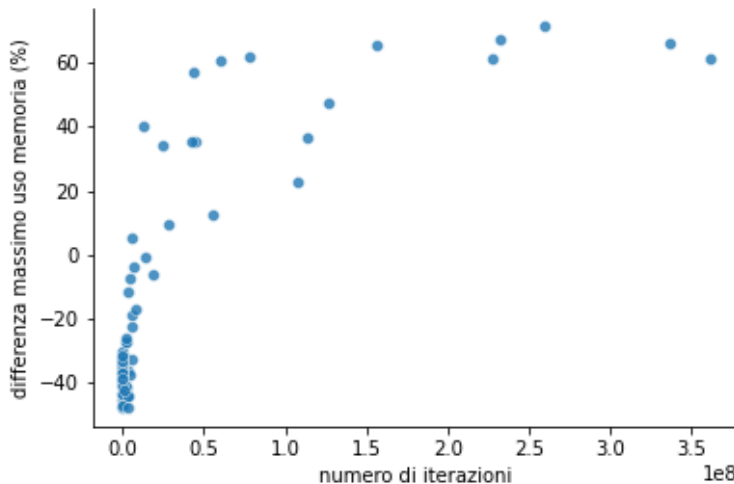


Figura 5.9: Differenza percentuale nell'uso della memoria rispetto alla prima versione dell'algoritmo, numero di iterazioni inferiore a 500 milioni. Valori negativi indicano un peggioramento delle prestazioni spaziali

La figura precedente mostra che per alcune matrici si ha un peggioramento notevole delle prestazioni spaziali, con un massimo del 47.68 % con la matrice c499.010. Tuttavia, si scopre che l'occupazione di memoria per queste matrici è relativamente piccola se confrontata con altre, valori assoluti attorno ai 55-60 MiB. Quindi per queste matrici è tollerabile un incremento dell'occupazione spaziale, almeno su sistemi con un discreto quantitativo di RAM come quello su cui sono stati eseguiti i test.

Proseguendo nell'analisi si nota come, all'aumentare del numero di iterazioni, si passa da un peggioramento delle prestazioni spaziali ad un notevole miglioramento. Per esempio, con la matrice c880.032 si ha un'occupazione massima della memoria con la prima versione di 3785.24609375 MiB, mentre con la seconda versione si ha 913.8828125 MiB, una riduzione del 75.87 %. I risultati ottenuti ci portano quindi a ritenere la seconda versione la migliore. Di seguito sono mostrati alcuni grafici, che rappresentano l'andamento dell'occupazione di memoria nel tempo durante l'esecuzione delle due versioni su due matrici. L'assenza del backend grafico utilizzato per la realizzazione dei grafici nell'ambiente PyPy ha imposto l'utilizzo del normale interprete Python, per cui i tempi di esecuzione sono più lunghi.



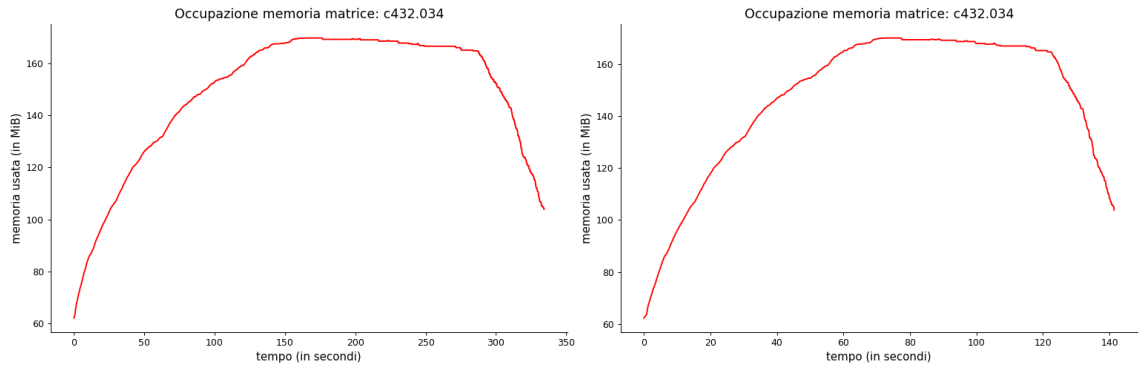


Figura 5.10: Uso della memoria durante l'esecuzione della prima versione sulla matrice c432.034, a sinistra l'esecuzione senza pre-elaborazione, a destra quella con.

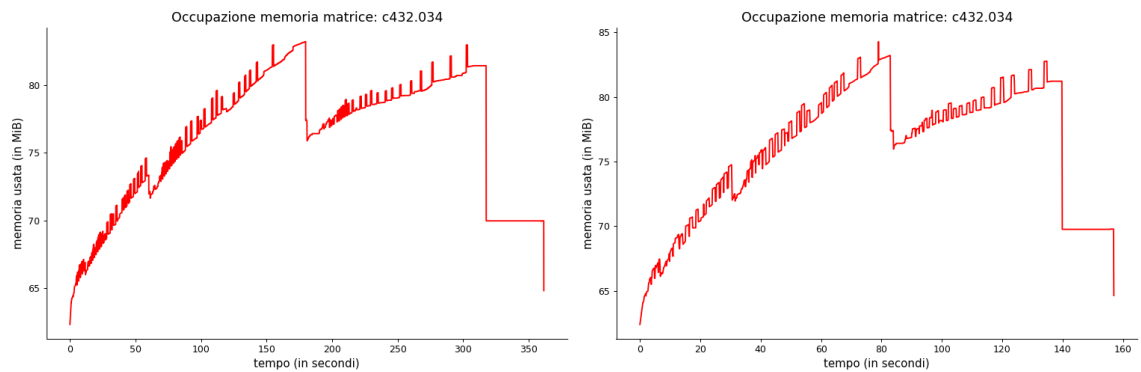


Figura 5.11: Uso della memoria durante l'esecuzione della seconda versione sulla matrice c432.034, a sinistra l'esecuzione senza pre-elaborazione, a destra quella con.

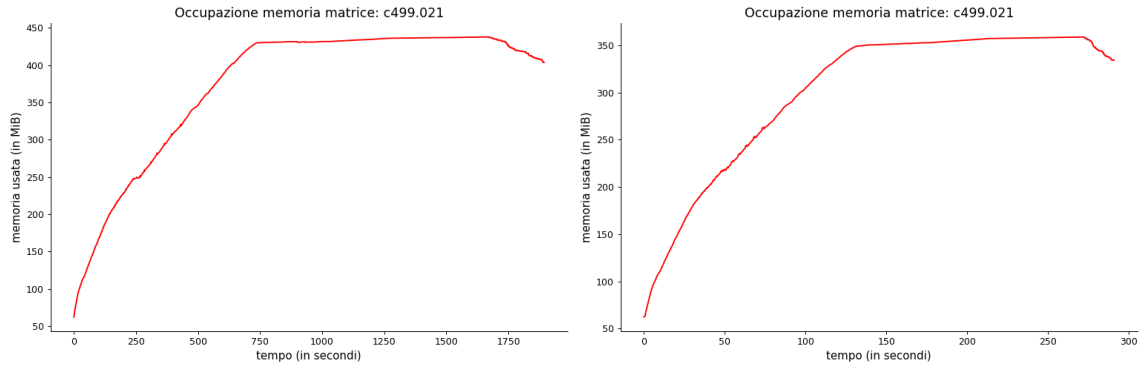


Figura 5.12: Uso della memoria durante l'esecuzione della prima versione sulla matrice c499.021, a sinistra l'esecuzione senza pre-elaborazione, a destra quella con.

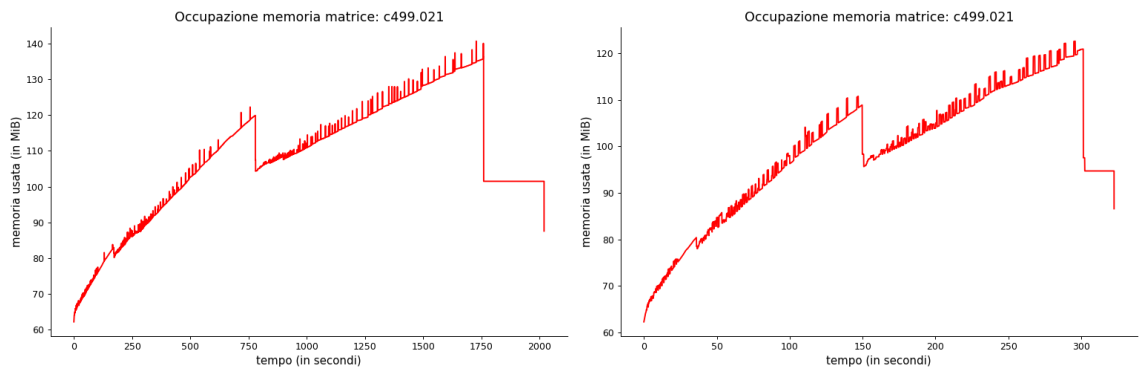


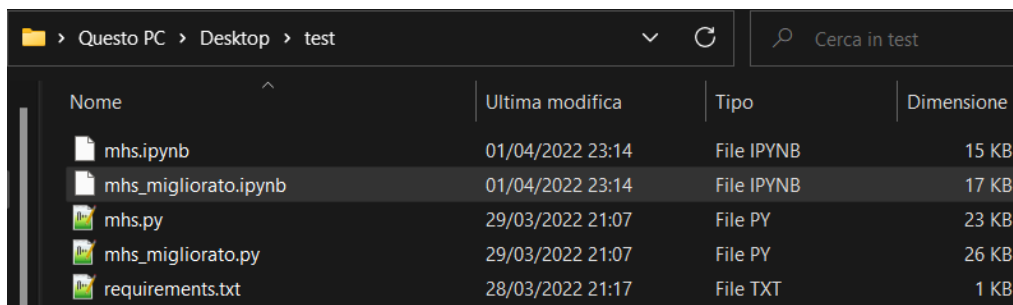
Figura 5.13: Uso della memoria durante l'esecuzione della seconda versione sulla matrice c499.021, a sinistra l'esecuzione senza pre-elaborazione, a destra quella con.

# Capitolo 6

## Manuale utente

### 6.1 Installazione

Di seguito sono presentate le istruzioni per eseguire lo script sulla propria macchina, oppure tramite Google Colab. Colab permette di utilizzare dei runtime Python senza dover installare nulla sulla propria macchina, ed è stato utilizzato durante le prime fasi di realizzazione dell'elaborato. È stato ritenuto utile mantenere questa possibilità una volta terminata la realizzazione dello script .py, anche se in forma limitata. I file .ipynb allegati permettono di eseguire l'algoritmo su una matrice scelta dall'utente, senza però poter salvare i risultati ottenuti su file e senza il tracciamento della memoria utilizzata durante l'esecuzione, a causa di limitazione dei runtime di Colab. L'uso di questi notebook è pertanto consigliato solo in assenza di un runtime Python sulla propria macchina, oppure per verificare velocemente le funzionalità richieste. Al fine di eseguire l'algoritmo sulla propria è necessario avere installato Python, preferibilmente la versione 3.9 sulla quale l'algoritmo è stato testato, anche se non ci dovrebbero essere problemi con altre versioni precedenti, purché siano Python 3. Nella cartella dei file sorgenti, oltre ai già citati file .ipynb, sono presenti altri tre file, mhs.py, mhs\_migliorato.py e requirements.txt. Il primo rappresenta il file sorgente dell'applicazione nella prima versione realizzata, il secondo file è la versione migliorata che adotta l'accorgimento visto in precedenza per ridurre l'occupazione di memoria durante l'esecuzione. Il terzo file costituisce la lista dei moduli richiesti dall'applicazione che non sono inclusi nella distribuzione di Python.



Nome	Ultima modifica	Tipo	Dimensione
mhs.ipynb	01/04/2022 23:14	File IPYNB	15 KB
mhs_migliorato.ipynb	01/04/2022 23:14	File IPYNB	17 KB
mhs.py	29/03/2022 21:07	File PY	23 KB
mhs_migliorato.py	29/03/2022 21:07	File PY	26 KB
requirements.txt	28/03/2022 21:17	File TXT	1 KB

Figura 6.1: Cartella contenente i file sorgenti dell'applicazione

Infatti, oltre all'installazione di Python, sono richiesti altri tre moduli non presenti nativamente, ed essi sono specificati nel file `requirements.txt`. Per installarli è consigliato usare il gestore dei pacchetti di Python, `pip`, che è presente nella maggior parte delle installazioni di Python. Per fare ciò aprire un terminale nella cartella in cui si trova il file `requirements.txt` ed eseguire il comando `pip install -r requirements.txt`. Al termine dell'operazione tutti i moduli richiesti saranno installati e sarà possibile eseguire il file `mhs.py` o il file `mhs_migliorato.py`.

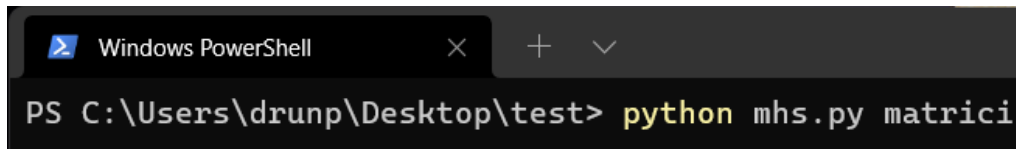
```
PS C:\Users\drunp\Desktop\test> pip install -r requirements.txt
Collecting memory_profiler==0.58.0
  Using cached memory_profiler-0.58.0.tar.gz (36 kB)
  Preparing metadata (setup.py) ... done
Collecting pynput==1.7.5
  Using cached pynput-1.7.5-py2.py3-none-any.whl (87 kB)
Collecting matplotlib~=3.5.1
  Using cached matplotlib-3.5.1-cp39-cp39-win_amd64.whl (7.2 MB)
```

Figura 6.2: Installazione dei moduli necessari

## 6.2 Eseguire il programma

Per eseguire l'applicazione è sufficiente eseguire il comando `python mhs.py` (o `python mhs_migliorato.py`) e un messaggio di aiuto riguardante il funzionamento dell'applicazione verrà mostrato. Il programma realizzato riceve in ingresso il percorso di una cartella (relativo alla posizione del file eseguito o anche un percorso assoluto) e permette di applicare le due versioni

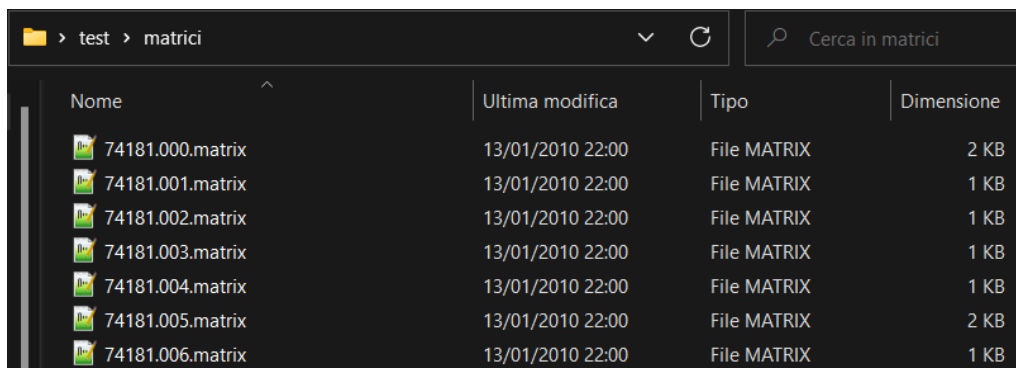
dell'algoritmo (base e con pre-elaborazione) a tutte le matrici (i file con estensione .matrix) che trova nella cartella, oppure di applicare solo una delle due versioni.



```
PS C:\Users\drunp\Desktop\test> python mhs.py matrici
```

Figura 6.3: Come eseguire il programma

Durante l'esecuzione l'algoritmo salva i risultati delle elaborazioni su un file chiamato risultati.csv, contenuto in cartella chiamata risultati che è stata creata appositamente dentro la cartella specificata prima.



Nome	Ultima modifica	Tipo	Dimensione
74181.000.matrix	13/01/2010 22:00	File MATRIX	2 KB
74181.001.matrix	13/01/2010 22:00	File MATRIX	1 KB
74181.002.matrix	13/01/2010 22:00	File MATRIX	1 KB
74181.003.matrix	13/01/2010 22:00	File MATRIX	1 KB
74181.004.matrix	13/01/2010 22:00	File MATRIX	1 KB
74181.005.matrix	13/01/2010 22:00	File MATRIX	2 KB
74181.006.matrix	13/01/2010 22:00	File MATRIX	1 KB

Figura 6.4: La cartella contenente le matrici prima dell'esecuzione

Nome	Ultima modifica	Tipo	Dimensione
risultati	23/03/2022 17:44	Cartella di file	
74181.000.matrix	13/01/2010 22:00	File MATRIX	2 KB
74181.001.matrix	13/01/2010 22:00	File MATRIX	1 KB
74181.002.matrix	13/01/2010 22:00	File MATRIX	1 KB
74181.003.matrix	13/01/2010 22:00	File MATRIX	1 KB
74181.004.matrix	13/01/2010 22:00	File MATRIX	1 KB
74181.005.matrix	13/01/2010 22:00	File MATRIX	2 KB

Figura 6.5: La cartella contenente le matrici dopo l'esecuzione

All'interno della cartella risultati saranno presenti due cartelle:

- *mhs*, la quale contiene i MHS calcolati durante l'esecuzione di una o tutte le varianti dell'algoritmo. Per esempio, nel caso venga eseguito solo l'algoritmo base e non venga effettuata alcuna pre-elaborazione, sarà presente soltanto la cartella *base*, mentre se viene eseguita anche la pre-elaborazione allora sarà presente anche la cartella *pre\_elab*.
- *plot\_memoria*, la quale contiene i grafici relativi all'andamento dell'occupazione di memoria nel tempo durante l'esecuzione dell'algoritmo. Anche questa cartella è divisa come la cartella *mhs* in due sottocartelle, *base* e *pre\_elab*

Nome	Ultima modifica	Tipo	Dimensione
mhs	29/03/2022 21:31	Cartella di file	
plot_memoria	29/03/2022 21:31	Cartella di file	
risultati.csv	29/03/2022 21:34	File con valori sep...	4 KB

Figura 6.6: La cartella contenente i risultati

Nome	Ultima modifica	Tipo
base	26/03/2022 13:26	Cartella di file
pre_elab	26/03/2022 13:28	Cartella di file

Figura 6.7: La cartella contenente i MHS risultanti

Nome	Ultima modifica	Tipo	Dimensione
74181.000.txt	26/03/2022 13:16	File TXT	4 KB
74181.001.txt	26/03/2022 13:16	File TXT	2 KB
74181.002.txt	26/03/2022 13:16	File TXT	1 KB
74181.003.txt	26/03/2022 13:16	File TXT	1 KB
74181.004.txt	26/03/2022 13:16	File TXT	1 KB
74181.005.txt	26/03/2022 13:16	File TXT	1 KB

Figura 6.8: Risultati dell'esecuzione senza pre-elaborazione

Durante l'esecuzione l'algoritmo informa l'utente sulle operazioni che stanno venendo eseguite: inizialmente riporta il nome della matrice trattata, le sue dimensioni e l'inizio dell'esecuzione dell'algoritmo base. Nel caso ci siano errori nel file di ingresso, l'applicazione segnalerà tali errori all'utente, chiedendo se vuole procedere con l'esecuzione dell'algoritmo sulla matrice successiva.

```
Caricamento matrice dal file matrici\74181.001.matrix
Matrice caricata con successo
Matrice: 74181.001
Numero righe: 2
Numero colonne: 65
Inizio esecuzione algoritmo base sulla matrice 74181.001, premi SPAZIO per terminarla
```

Figura 6.9: Informazioni sul caricamento della matrice

Alla fine di ciò, l'utente viene informato del tempo di esecuzione richiesto, la massima occupazione di memoria che si è registrata durante l'esecuzione,

il numero di iterazioni del ciclo for più interno che sono stata eseguite, il numero di MHS trovati, la cardinalità minima e massima di questi MHS.

```
Esecuzione dell'algoritmo base completata

Il tempo richiesto dall'esecuzione base e' stato di 0.015625 s
Il numero di iterazioni compiute e' stato di 750
La massima occupazione di memoria e' stata di 30.8203125 MiB
Sono stati trovati 95 MHS
La cardinalita' minima dei MHS trovati e' 1
La cardinalita' massima dei MHS trovati e' 2
```

Figura 6.10: Riepilogo dopo l'esecuzione dell'algoritmo base

Dopo di che informa l'utente dell'inizio dell'esecuzione con pre-elaborazione dell'algoritmo. Alla fine di questa esecuzione vengono fornite le stesse informazioni riepilogative di prima, precedute dalle informazioni relative alle nuove dimensioni della matrice dopo la pre-elaborazione, gli indici di riga e colonna rimossi. Nel caso sia stato possibile fare il confronto tra i risultati ottenuti con l'esecuzione dell'algoritmo base viene anche riportato se gli MHS trovati sono gli stessi oppure no, altrimenti viene indicato che non è stato possibile fare questo confronto.

```
Il tempo richiesto dalla pre-elaborazione e' stato di 0.0 s
Dopo l'esecuzione della pre-elaborazione, il nuovo numero di righe e' 3
Dopo l'esecuzione della pre-elaborazione, il nuovo numero di colonne e' 21
Gli indici di riga rimossi sono:

Gli indici di colonna rimossi sono: 1 3 4 5 6 7 8 9 13 14 15 16 17 18 19 21 22 23
25 26 29 30 33 34 35 36 37 38 39 40 41 42 43 48 49 50 53 54 55 56 58 59 60 62 -
```

Figura 6.11: Riepilogo relativo alla pre-elaborazione della matrice

Nel caso in cui l'utente interrompa l'esecuzione dell'algoritmo base prima che questa sia terminata, il programma riporta comunque le informazioni parziali ottenute fino a quel momento, poi chiede all'utente se vuole proseguire con l'esecuzione della versione con pre-elaborazione sulla stessa matrice. In caso affermativo il programma procede normalmente, altrimenti chiede all'utente se si vuole proseguire con l'esecuzione sulla prossima matrice. In caso affermativo, viene caricata la matrice successiva (se presente) e l'esecuzione procede normalmente, altrimenti l'esecuzione del programma termina.



```

Inizio esecuzione algoritmo base sulla matrice 74181.032, premi SPAZIO per terminarla
Nota: SPAZIO termina l'esecuzione anche quando si e' in un'altra finestra
Esecuzione terminata dall'utente

Il tempo richiesto dall'esecuzione base e' stato di 54.375 s
Il numero di iterazioni compiute e' stato di 3179271
La massima occupazione di memoria e' stata di 88.1484375 MiB
Sono stati trovati 50348 MHS
La cardinalita' minima dei MHS trovati e' 4
La cardinalita' massima dei MHS trovati e' 7

Vuoi proseguire con l'applicazione della pre-elaborazione sulla matrice corrente? [s/n]

```

Figura 6.12: Terminazione da parte dell'utente durante l'algoritmo base

```

Esecuzione terminata dall'utente

Il tempo richiesto dalla pre-elaborazione e' stato di 0.0 s
Dopo l'esecuzione della pre-elaborazione, il nuovo numero di righe e' 30
Dopo l'esecuzione della pre-elaborazione, il nuovo numero di colonne e' 52
Gli indici di riga rimossi sono: 27 26 15
Gli indici di colonna rimossi sono: 1 2 3 4 6 7 12 13 16 17 22 23 64 -

Il tempo richiesto dall'esecuzione con pre-elaborazione e' stato di 1.34375 s
Il numero di iterazioni compiute e' stato di 121796
La massima occupazione di memoria e' stata di 69.37890625 MiB
Sono stati trovati 0 MHS
La cardinalita' minima dei MHS trovati e' 0
La cardinalita' massima dei MHS trovati e' 0

Almeno una delle due esecuzioni dell'algoritmo e' stata terminata dall'utente, il
controllo dei risultati non puo' essere fatto

Vuoi proseguire l'esecuzione del programma con la prossima matrice? [s/n]
n
Esecuzione programma terminata

```

Figura 6.13: Terminazione da parte dell'utente durante l'algoritmo con pre-elaborazione

Nel caso in cui l'utente scelga di eseguire soltanto una delle due varianti dell'algoritmo (con o senza pre-elaborazione), l'esecuzione avviene nello stesso modo, offrendo all'utente la possibilità di interrompere l'esecuzione. I risultati parziali così ottenuti vengono salvati come descritto prima, semplicemente alcune cartelle saranno mancanti. Per esempio, nel caso si esegua soltanto la versione base senza pre-elaborazione, le cartelle *mhs* e *plot\_memoria* contengono solo la sottocartella *base*.

## 6.3 Opzioni

Sono disponibili alcune opzioni che modificano il comportamento del programma. Per inserire queste opzioni basta inserirle dopo il comando `python mhs.py`. Per esempio `python mhs.py -h` (oppure `python mhs_migliorato.py -h`) mostra il messaggio di aiuto.

- `-h`, `-help`, è l'opzione per mostrare il messaggio di aiuto che illustra le opzioni disponibili.
- `-o`, `-output`, per cambiare il nome del file di output (che ha come valore di default `risultati.csv`)
- `-no-plot`, per disabilitare il salvataggio dell'uso della memoria nel tempo in forma di grafico
- `-v`, `-versione`, per scegliere se eseguire solo l'algoritmo base (opzione 1), eseguire solamente l'algoritmo con pre-elaborazione (opzione 2) oppure eseguire entrambe le versioni (opzione 0, quella scelta di default anche quando l'opzione non è specificata)
- `-m`, `-matrice`, per abilitare il salvataggio dei MHS come la matrice di ingresso (quindi una matrice di M colonne e N righe, dove N è il numero di MHS trovati)

L'ultima opzione è stata introdotta perché in alcuni casi il numero di MHS trovati potrebbe essere molto grande, con conseguenze ovvie sulla dimensione del file. Se l'opzione `-m` non viene inserita, per ogni MHS viene salvata la rappresentazione interna che il programma usa, ovvero vengono salvati gli indici di colonna degli elementi presenti nel MHS.

## 6.4 PyPy

I risultati visti nella sezione sperimentazione non stati ottenuti utilizzando l'interprete Python normalmente installato sulle macchine, ma sono stati ottenuti usando PyPy. PyPy è un'implementazione del linguaggio Python alternativa a CPython (l'implementazione standard che viene usata normalmente). L'utilizzo di PyPy permette di ottenere un miglioramento nelle prestazioni in termini di tempi esecuzione in media di 4.2 volte rispetto a

CPython, questo perché utilizza un compilatore just-in-time. Installando PyPy sulla propria macchina sarà possibile eseguire mhs.py senza dover apportare alcuna modifica al codice dell'applicazione, è necessario solamente reinstallare i due moduli richiesti dall'applicazione. Una guida sul uso di PyPy può essere trovata sul sito <https://www.pypy.org/>. Si è tuttavia constatato come il modulo matplotlib non sia disponibile in PyPy per via di alcune dipendenze mancanti, per cui, nel caso si utilizzi PyPy, il plot della memoria usata nel tempo non è disponibile.

# Capitolo 7

## Conclusioni

In conclusione, il lavoro illustrato in questa relazione ha portato all'implementazione dell'algoritmo richiesto in Python, oltre alle funzionalità di pre-elaborazione. Queste ultime hanno un impatto notevole sul tempo di esecuzione dell'algoritmo, risultando quindi di grande beneficio. Il problema principale però è costituito dall'occupazione di memoria, molto elevata in alcuni casi. Le scelte prese in fase di implementazione hanno permesso di evitare la memorizzazione dei vettori rappresentativi, a discapito delle prestazioni temporali, mentre i miglioramenti introdotti con la seconda versione vanno ad agire sulla memorizzazione degli insiemi, avendo quindi un impatto notevole. Sviluppi futuri dell'applicazione possono andare in diverse direzioni. Si potrebbe modificare il linguaggio d'implementazione dell'algoritmo, andando a scegliere un linguaggio di più basso livello come il C, per cercare di ottenere una maggiore efficienza. Oppure ci si può interrogare su come si possa rappresentare i vettori rappresentativi in maniera più intelligente, al fine di memorizzarli e ottenere un significativo miglioramento delle prestazioni temporali. Infine si potrebbe migliorare ulteriormente la struttura dati utilizzata per memorizzare l'insieme visto che, per come l'abbiamo definita, ci sono molti elementi di insiemi che risultano ripetuti, comportando quindi uno spreco della memoria.