



DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato A.A. 2021/2022

Algoritmi e strutture dati

Problema di Exact Cover

Baselli Giovanni - 718969

INDICE

1. Introduzione	2
1.1 Compiti e obiettivi del lavoro	2
1.2 Problema EC (Exact Cover)	3
1.2.1 Insiemi coinvolti	3
1.2.2 Input	5
1.2.3 Output	6
1.2.4 Matrice di compatibilità	7
1.3 Principi e funzionamento dell'algoritmo	8
2. File di input	13
3. File di output	17
4. Algoritmo base: EC ed ESPLORA	23
5. Algoritmo plus: EC+ ed ESPLORA+	28
6. Implementazione	31
6.1 Linguaggio di programmazione e ambiente di sviluppo	31
6.2 Struttura del notebook ed esecuzione	33
6.2.1 Ordine di esecuzione	36
6.3 Generazione del file di input	38
6.4 Lettura del file di input	44
6.5 Implementazione dell'algoritmo EC	45
6.6 Implementazione dell'algoritmo EC+	52
7. Prove sperimentali	56
7.1 Confronto diretto	57
7.1.1 Esempi di confronto diretto per valore	60
7.2 Sperimentazione multipla	63
7.2.1 Confronto multiplo	64
7.2.2 Creazione delle cartelle	65
7.2.3 Creazione del file di input	66
7.2.4 Esecuzione degli algoritmi	68
7.2.5 Visualizzazione degli andamenti	76
8. Conclusioni e osservazioni finali	91
8.1 Limitazioni e problemi riscontrati	91
8.2 Risultati ottenuti	93

1. Introduzione

1.1 Compiti e obiettivi del lavoro

Il presente elaborato è stato redatto al fine di illustrare il lavoro svolto per la risoluzione del problema di Exact Cover, attenendosi alle definizioni, spiegazioni e richieste fornite nel testo disponibile [qui](#).

In particolare, in questa relazione vengono presentati il problema, le scelte implementative compiute, le prove sperimentali eseguite e la valutazione critica delle stesse. Ogni capitolo è focalizzato su una specifica richiesta e fornisce tutti i dettagli utili a comprendere le decisioni prese, le strutture dati utilizzate, gli sforzi tesi a ridurre il costo temporale della computazione, le indicazioni ritenute utili per consentire l'utilizzo dei programmi software utilizzati e le sperimentazioni, le limitazioni riscontrate nelle prove.

Per adempiere alle richieste software, nello specifico, è stato necessario:

- progettare e sviluppare un programma per l'algoritmo base in grado di risolvere il problema di Exact Cover, sulla base dello pseudocodice fornito in questo documento;
- progettare e sviluppare un programma per un'ulteriore versione dell'algoritmo, chiamata algoritmo plus, sulla base dello pseudocodice fornito nella documentazione;
- progettare e sviluppare un programma per la generazione del file di input;
- progettare e sviluppare un programma per effettuare un confronto tra le uscite calcolate dalle due versioni dell'algoritmo.

Tali soluzioni sono reperibili direttamente dal notebook Colab disponibile [qui](#).

1.2 Problema EC (Exact Cover)

Nell'ambito matematico del calcolo combinatorio, data una collezione N di sottoinsiemi di un insieme M , una **copertura esatta** è definita come l'insieme delle sottocollezioni N^* di N tali che tutti gli elementi di M siano contenuti in esattamente un sottoinsieme di N^* . In altre parole, N^* è una **copertura esatta** o **partizione** di M formata da sottoinsiemi contenuti in N .

In ambito informatico, il *problema di copertura esatta* o *Exact Cover* è un problema di ottimizzazione combinatoria NP-completo ed è uno dei 21 problemi di tipo NP-completo di Karp. In particolare, esso è un problema di decisione che mira a determinare se esistano coperture esatte dati una collezione N e un insieme M .

Molti problemi conosciuti possono essere ridotti a problemi di copertura esatta. Infatti, la risoluzione dei Sudoku, il problema della piastrellatura di una certa area attraverso pentamini e il problema delle otto regine sono solo alcuni degli esempi che possono essere visti come problemi appartenenti a questa categoria.

1.2.1 Insiemi coinvolti

Nella risoluzione del problema di EC sono coinvolti i seguenti tre insiemi:

- M : definito come dominio, è un insieme di elementi distinti. In particolare, in questa soluzione, ciascun insieme è rappresentato come una coppia ordinata lettera-cifra araba (ad esempio, a1, b2, b3, c9). Inoltre, ciascun elemento di M è identificato univocamente da un indice intero appartenente all'intervallo $[1..|M|]$, dove $|M|$ rappresenta la cardinalità dell'insieme M .
- N : è una collezione finita di insiemi finiti (distinti) dove gli elementi di ogni insieme appartengono al dominio M . Ogni elemento di N è univocamente identificato da un indice intero appartenente all'intervallo $[1..|N|]$, dove $|N|$ è la cardinalità dell'insieme N .

- *Partizioni*: è l'insieme di tutte le partizioni (o coperture esatte) di M dove ciascuna parte è un insieme della collezione N . Una singola partizione di M è un (sotto)insieme della collezione N costituito da insiemi tutti reciprocamente disgiunti e tali che la loro unione sia M .

Vediamo ora un **esempio**, a cui si farà più volte riferimento all'interno di questa sezione introduttiva. Qui viene mostrata la struttura degli insiemi appena definiti. Per l'insieme M e la collezione N sono presentati, in colore, gli indici identificativi di ciascun elemento.

```

      1      2      3      4      5      6
M = {a1, b2, b3, c4, a4, d2}

N = { {a1, b2},           1
      {a1, b3, a4},       2
      {b2, c4, d2},       3
      {b3, a4, d2},       4
      {c4},               5
      {b2, c4, b3, a1, a4, d2} 6
      {a1, b3} }          7

Partizioni = { {{a1, b2}, {b3, a4, d2}, {c4}},
               {{a1, b3, a4}, {b2, c4, d2}},
               {{b2, c4, b3, a1, a4, d2}} }

|M| = 6
|N| = 7
|Partizioni| = 3

```

1.2.2 Input

Il problema di Exact Cover prevede in input la collezione N che, come definito precedentemente, è una collezione finita di insiemi finiti (distinti) dove gli elementi di ogni insieme appartengono al dominio M . Si può assumere che M sia l'unione di tutti gli insiemi della collezione N .

È però possibile rappresentare l'input in un ulteriore modo, ovvero attraverso una matrice $A_{|N|, |M|}$, dove il componente $a_{i,j}$ della matrice ha valore:

- **1**, se l'elemento di M di indice j appartiene all'insieme in N di indice i ;
- **0**, altrimenti.

Considerando l'esempio fatto in precedenza, la matrice A che si ottiene è quella rappresentata nel riquadro seguente. In particolare, ogni riga è associata ad un elemento della collezione N , mentre ogni colonna è associata ad un elemento del dominio M . Gli indici rappresentati per ciascuna riga e colonna sono utili a comprendere l'associazione rispetto al valore effettivo degli elementi all'interno dei vari insiemi.

$$A_{|N| \times |M|} = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} & \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

Nelle soluzioni sviluppate in questo progetto è stato adottato l'input in formato matriciale appena definito.

1.2.3 Output

L'output del problema di Exact Cover è l'insieme *Partizioni*, ovvero l'insieme di tutte le partizioni (o coperture esatte) di M dove ciascuna parte è un insieme della collezione N . In particolare, la soluzione fornita rappresenta le partizioni attraverso l'insieme COV , ovvero un insieme per cui ciascuna partizione trovata viene rappresentata come insieme di indici di sottoinsiemi di N . In altre parole, ogni indice è un identificatore del corrispettivo insieme appartenente alla collezione N , e quindi una partizione sarà un insieme di identificatori i cui insiemi di N associati compongono la partizione stessa.

Proseguendo con l'esempio fatto nelle sezioni precedenti, è possibile rappresentare il relativo insieme COV come nel riquadro sottostante. Ad ogni elemento di N contenuto in ciascuna partizione è associato un indice relativo all'insieme N , pertanto il risultato finale sarà una collezione di insiemi di indici.

```
Partizioni = { 1{a1, b2}, 4{b3, a4, d2}, 5{c4}},  
              2{a1, b3, a4}, 3{b2, c4, d2}},  
              6{b2, c4, b3, a1, a4, d2}} }  
  
COV = { {1, 4, 5},  
        {2, 3},  
        {6} }
```

Conoscendo quindi le coppie indice-elemento è possibile ricavare dagli indici dell'output gli insiemi originali, e di conseguenza si può visualizzare e, in caso, verificare adeguatamente che la partizione creata rispetti la definizione di partizione.

1.2.4 Matrice di compatibilità

Un altro concetto importante per la comprensione e la risoluzione del problema di Exact Cover è quello della matrice di compatibilità $B_{|N|, |N|}$, ovvero una matrice simmetrica di dimensione $|N|$ in cui il componente $b_{i,j}$ assume il valore **1** se valgono tutte e tre le seguenti condizioni:

- $i \neq j$
- $A[i] \cap A[j] = \emptyset$
- $A[i] \cup A[j] \neq M$

dove

- $A[i]$: riga i della matrice A , associata all'insieme i -esimo della collezione N ;
- $A[j]$: riga j della matrice A , associata all'insieme j -esimo della collezione N ;
- $A[i] \cap A[j] = \emptyset$ indica che non esistono elementi in comune tra gli insiemi i -esimo e j -esimo di N . In particolare, non esistono colonne per cui la riga i e la riga j di A abbiano entrambe un 1;
- $A[i] \cup A[j] \neq M$ indica che unendo gli insiemi i -esimo e j -esimo non si ottiene il dominio M . In particolare, le righe i e j non possono avere, per ciascuna colonna in comune, sempre valori diversi l'una dall'altra.

Se almeno una delle tre condizioni sopracitate non è valida, il componente $b_{i,j}$ assume il valore **0**.

Essendo B una matrice simmetrica, è possibile riempire solamente le celle che compongono il “triangolo superiore” della matrice, ovvero i $b_{i,j}$ per cui $j > i$.

Quando $b_{i,j} = 1$, allora si può dire che $A[i]$ e $A[j]$ (con $j > i$) sono *insiemi compatibili*. Ciò implica che ad essi è possibile aggregare ulteriori insiemi al fine di formare delle partizioni.

Nel riquadro seguente è possibile osservare la matrice B risultante per il caso in esempio. Il triangolo verde sta ad indicare la mancanza di necessità dell'inserimento di valori nella parte inferiore della matrice B in quanto essa simmetrica.

$$B_{|N| \times |N|} = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

Questa matrice è in particolar modo utile, all'interno dell'algoritmo sviluppato, per verificare quando, per aggregati di cardinalità > 2 , gli insiemi che costituiscono tale aggregato sono compatibili a due a due. Questo aspetto è molto importante dal punto di vista della visita dei nodi di un albero, di cui si parlerà più nello specifico nel prossimo paragrafo.

1.3 Principi e funzionamento dell'algoritmo

Le due versioni dell'algoritmo risolvente del problema di Exact Cover sfruttano l'ordine degli insiemi entro la collezione N , denominato *ordine lessicografico*. Durante la sua esecuzione, l'algoritmo produce in modo incrementale la matrice di compatibilità $B_{|N|, |N|}$, analizzando nel mentre gli aggregati di insiemi della collezione N la cui compatibilità sia già stata appurata.

L'operato dell'algoritmo si può descrivere in termini di *esplorazione di alberi*, dove:

- il *nodo* di un albero rappresenta un aggregato di uno o più insiemi della collezione N . Tale nodo non può presentare un aggregato che ha come elemento un insieme vuoto o un insieme coincidente con M . Ad ogni nodo di un albero è associato l'insieme degli identificatori degli insiemi che

appartengono all'aggregato a cui il nodo si riferisce. Per semplicità, indicheremo un nodo come sequenza di indici (come singola cifra): ad esempio, il nodo 421 è il nodo che rappresenta l'aggregato degli insiemi 4, 2 e 1 della collezione N ;

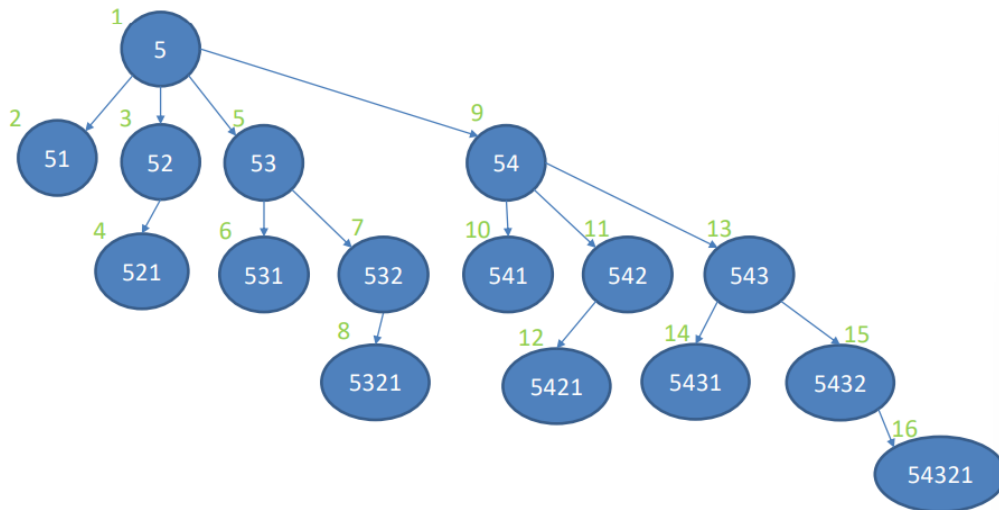
- la *radice* di un (singolo) albero è un singolo insieme distinto della collezione N . A ciascuna delle radici è associato un insieme di identificatori singoletto $\{i\}$, con $1 \leq i \leq |N|$, dove $A[i]$ non è l'insieme vuoto e non coincide con M ; l'albero avente tale radice si dice “radicato in i ”;
- un (singolo) *albero* contiene tutti e soli gli aggregati composti dall'insieme della radice dell'albero e dagli insiemi (non vuoti e non coincidenti con M) che lo precedono secondo l'ordine lessicografico degli insiemi in N .

Gli alberi vengono visitati per valori crescenti degli identificatori delle loro radici (secondo l'ordine lessicografico). Durante la visita dell'albero radicato in i vengono inizializzate le caselle della matrice di compatibilità B della colonna relativa ad i , implicando così la visita di tutti i nodi corrispondenti ad aggregati di cardinalità 2.

Per quanto riguarda, invece, nodi corrispondenti ad aggregati di cardinalità > 2 , la visita di tali nodi viene effettuata solo se gli insiemi che lo costituiscono sono compatibili a due a due basandosi sui valori della matrice B .

Ad esempio, considerando la matrice B dell'esempio si può evincere che gli insiemi di indice 1, 4 e 5 in N siano compatibili fra loro in quanto $b_{1,4}$, $b_{1,5}$ e $b_{4,5}$ sono tutti uguali a 1. Ciò implica che, dato l'aggregato degli insiemi 1, 4 e 5, su di esso si possano cercare nuovi aggregati tali da costruire una nuova partizione o meno. Pertanto, il nodo 541 verrà visitato.

Gli insiemi 1, 2 e 5 sono invece incompatibili in quanto $b_{1,5}$, $b_{2,5} = 1$, ma $b_{1,2} = 0$, ovvero nonostante gli insiemi '1, 5' e '2, 5' siano compatibili a coppie, gli insiemi 1 e 2 non lo sono. Perciò, il nodo 521 non verrà visitato.



Nell'immagine soprastante è rappresentato un esempio di albero. In particolare, esso definisce l'albero radicato nell'insieme di indice 5 in N , assumendo che gli alberi precedentemente esplorati siano quelli radicati negli insiemi con indice da 1 a 4 (con insieme radice né vuoto né coincidente con M) e che l'ordine entro la collezione rispecchi l'ordine dei valori interi. Come già citato, un aggregato, per semplicità, è indicato dalla sequenza di identificatori degli insiemi che lo costituiscono.

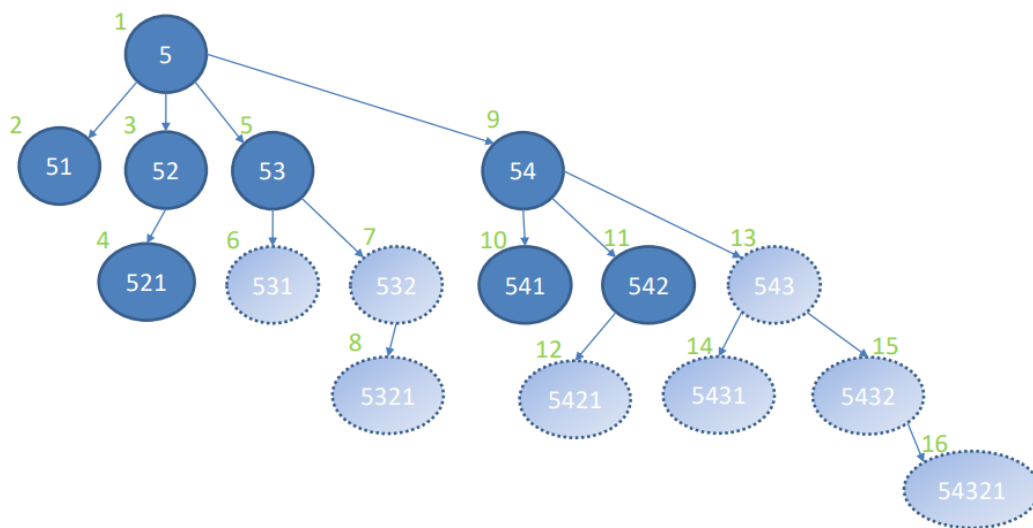
In verde sono riportati i numeri relativi all'ordine di visita dei nodi dell'albero. L'ordine di visita garantisce che vengano inizializzate le caselle $B[1,5]$, $B[2,5]$, $B[3,5]$, $B[4,5]$ della matrice di compatibilità B nell'ordine indicato. Dopo l'inizializzazione di $B[1,5]$ e $B[2,5]$ sarà quindi possibile analizzare l'aggregato 521, in quanto le compatibilità degli aggregati 51, 52 e 21 sono note. In particolare, la compatibilità dell'aggregato 21 è riconosciuta dalla visita dell'albero radicato in 2, precedente a quella dell'albero radicato in 5. Seguendo poi lo stesso ragionamento si prosegue all'interno dell'albero fino al termine.

Tuttavia, non è necessario esplorare tutti i nodi dell'albero.

- Se la visita di un nodo di primo livello di un albero (ovvero un aggregato di cardinalità 2, figlio diretto di una radice) evidenzia che fra i due insiemi che lo compongono vi sia intersezione non nulla o che la loro unione coincida con M , allora i discendenti di tale nodo non verranno esplorati, così come tutti gli aggregati contenenti tali insiemi;

- Se la visita di un nodo di un livello successivo a primo evidenzia che l'aggregato rappresentato da tale nodo sia una partizione, allora i discendenti di tale nodo non verranno esplorati.

Vediamo ora l'esempio precedente aggiornato rispetto a quanto appena detto.



I nodi rappresentati con sfondo più chiaro e contorno tratteggiato rappresentano i nodi che non vengono visitati dall'algoritmo.

- Se gli insiemi 5 e 3 hanno intersezione non nulla o unione coincidente con M , i nodi discendenti del nodo 53 non vengono più visitati, così come non verranno visitati i nodi del sottoalbero radicato in 543;
- Se l'aggregato 542 è una partizione, i suoi nodi discendenti non vengono visitati.

L'algoritmo, osservando gli aspetti appena trattati, è quindi in grado di evitare la visita completa dell'albero, computando solamente su nodi che rispettano le prerogative citate. Ciò implica automaticamente una riduzione del tempo di visita dell'albero e quindi del tempo di esecuzione dell'algoritmo risolvante.

Ciascun albero possiede un totale di 2^i elementi (radice e nodi) possibili, dove i è un numero compreso tra 0 ed $|N|-1$ che identifica il rispettivo albero radicato in $i+1$. Pertanto, sommando il numero di elementi che compongono tutti gli alberi è possibile ottenere la totalità degli elementi che andrebbero visitati per avere un'esplorazione completa degli stessi. Tale valore è presentato nella seguente formula (dove tot_{nodi} racchiude sia i nodi relativi ad aggregati che le radici):

$$tot_{nodi} = \sum_{i=0}^{|N|-1} 2^i = 2^{|N|} - 1$$

Tuttavia, il numero di nodi effettivamente visitati potrebbe risultare ridotto se sono valide le condizioni citate ai punti precedenti. Pertanto, durante l'esecuzione dell'algoritmo, quando lecito, viene aggiornato il numero di nodi visitati, al fine di avere poi una percentuale dei nodi visitati su quelli totali e quindi un'idea di quanto sia stato risparmiato durante l'esecuzione in termini di esplorazione degli alberi.

2. File di input

Come definito nella sezione [Input](#), la matrice $A_{|N|, |M|}$ viene utilizzata come input per la risoluzione del problema di Exact Cover, rappresentando con ciascuna riga un insieme distinto di N e con ciascuna colonna un elemento del dominio M . Entrambi gli algoritmi sviluppati in questa soluzione utilizzano tale matrice, la quale viene generata e scritta su un file di testo (in formato `.txt`) da un programma per poi essere letta dallo stesso file grazie ad un ulteriore programma.

Nella seguente immagine è possibile osservare un esempio di file d'ingresso generato dal primo programma software citato, da cui è possibile evincere il formato interno e le caratteristiche di un qualsiasi tipo di file d'ingresso coinvolto in questo problema. Questo esempio verrà portato avanti anche nelle prossime pagine, ed è reperibile direttamente da [qui](#).

```
1 ;;; M = {h9, a1, l9, c5, x6}
2 ;;; N = {{x6, l9, a1}, {x6, h9, l9}, {a1}, {l9, c5}, {x6, h9}, {l9}, {c5}}
3
4 ;;; A =
5 0 1 1 0 1 -
6 1 0 1 0 1 -
7 0 1 0 0 0 -
8 0 0 1 1 0 -
9 1 0 0 0 1 -
10 0 0 1 0 0 -
11 0 0 0 1 0
12
13 ;;; Cardinality of domain M: 5
14 ;;; M indexes from 1 to 5
15 ;;; Cardinality of collection N: 7
16 ;;; N indexes from 1 to 7
17 ;;; Average cardinality for sets in N (avgN): 1.8571
18 ;;; Aggregate index (avgN/|M|): 0.3714
19 ;;; Sets in N with cardinality 1: 3 => 42.857%
20 ;;; Sets in N with cardinality 2: 2 => 28.571%
21 ;;; Sets in N with cardinality 3: 2 => 28.571%
22 ;;; Sets in N with cardinality 4: 0 => 0.0%
23 ;;; Sets in N with cardinality 5: 0 => 0.0%
24 ;;; Time to build and save A: 0.0013103570008752286s
```

Si può notare, grazie alla presenza delle due righe vuote, una suddivisione del file in tre macroaree:

- la *prima macroarea*, composta dalle prime due righe del file di testo, presenta sulla prima riga il dominio M e gli elementi che lo compongono, e sulla seconda riga la collezione N ed i sottoinsiemi che la compongono. La rappresentazione è utile ad avere un'idea della struttura e dei componenti di tali insiemi. Ciascun insieme e sottoinsieme è racchiuso tra parentesi graffe.
- la *seconda macroarea*, ovvero quella compresa tra le due righe vuote, presenta la matrice A . Ciascuna riga del file composta da sequenze (di uguale lunghezza) di numeri (0 e 1, separati da uno spazio bianco) corrisponde ad una riga della matrice A . Il carattere '-' al termine di ogni riga di testo ha la funzione di carattere separatore rispetto alle righe della matrice, ed è assente solamente nell'ultima riga, ad indicare che non sono presenti ulteriori nuove righe.
- la *terza macroarea*, ovvero l'ultima e seguente l'ultima riga vuota, racchiude tutte le caratteristiche utili ad un'ulteriore comprensione dei dati riportati precedentemente. In particolare, sono resi disponibili:
 - la *cardinalità del dominio M* e gli *indici* dei suoi elementi;
 - la *cardinalità della collezione N* e gli *indici* dei suoi sottoinsiemi;
 - la *cardinalità media degli insiemi presenti in N* , il cui valore è compreso tra 1 e la cardinalità di M ;
 - l'*aggregate index*, un indice ottenuto dividendo la cardinalità media degli insiemi presenti in N per la cardinalità di M . Il suo valore è utile a comprendere, in modo riassuntivo, la distribuzione di cardinalità degli insiemi presenti in N . In particolare, più il suo valore è vicino a 1, più sono presenti sottoinsiemi di N di cardinalità elevata (fino ad un valore massimo equivalente a $|M|$). Di questo valore si discuterà approfonditamente nei capitoli a seguire, soprattutto dal punto di vista dell'andamento dei risolutori;
 - la *distribuzione della cardinalità degli insiemi in N* con la rispettiva *percentuale* sul numero di insiemi in N ;
 - il *tempo trascorso per la generazione e la scrittura di A* .

Tutte le righe del file di testo introdotte da “;;;” sono definite come linee di commento, pertanto la parte di programma che si occupa della lettura del file di testo non baderà a tali righe, ma piuttosto si occuperà di costruire correttamente la matrice A ricavandola dalle righe prive di commento presenti nella seconda macroarea del file. Pertanto, le uniche righe non commentate sono quelle rappresentanti la matrice A .

In questo progetto, i file generati in automatico sono due:

1. il file ‘*input.txt*’, che presenta la struttura appena indicata e viene utilizzato dal programma di lettura per importare la matrice nel programma per l’esecuzione degli algoritmi di Exact Cover;
2. il file ‘*indexes.txt*’, un file di utilità che accompagna il file di input precedentemente generato, e che presenta:
 - in una prima parte, le medesime informazioni raccolte nel file ‘*input.txt*’;
 - in una seconda parte, una lista delle coppie indice-elemento per il dominio M e una lista delle coppie indice-sottoinsieme per la collezione N in ordine lessicografico. Questo aspetto è particolarmente utile per la lettura del contenuto del file di output, in quanto è possibile comprendere e verificare, tramite queste liste associative, se una data partizione in uscita (che si ricorda essere un insieme in COV composto da indici di sottoinsiemi di N costituenti una partizione) sia effettivamente corretta o meno.

Nella seguente immagine viene mostrato il tratto di testo che viene quindi aggiunto al file ‘*input.txt*’ dell’esempio precedente per costruire il file ‘*index.txt*’.


```

25
26 ;;; M indexes:
27 ;;; 1->h9
28 ;;; 2->a1
29 ;;; 3->l9
30 ;;; 4->c5
31 ;;; 5->x6
32
33 ;;; N indexes:
34 ;;; 1->{x6, l9, a1}
35 ;;; 2->{x6, h9, l9}
36 ;;; 3->{a1}
37 ;;; 4->{l9, c5}
38 ;;; 5->{x6, h9}
39 ;;; 6->{l9}
40 ;;; 7->{c5}

```

Si sottolinea che il file “*indexes.txt*” vuole fungere solamente da file di utilità, come specificato in precedenza. Ciò significa che esso non ha alcuna rilevanza ai fini dell’esecuzione dei programmi principali, ma che piuttosto può essere sfruttato per comprenderne al meglio i risultati.

Quanto definito fino ad ora descrive principalmente la struttura di un file prodotto in modo automatico da parte del programma software di generazione dell’input. Tuttavia, vi è comunque la possibilità di creare manualmente un file di testo e di fornire tale file in input al programma, a patto che le righe componenti la matrice A di interesse rispettino il formato definito in questa sezione. In particolare, non viene costruita la matrice A se nel file:

- le righe (non commentate) non presentano alcuna sequenza di valori;
- le righe (non commentate) presentano sequenze con valori diversi da 1 e 0;
- le righe (non commentate) con sequenze di 1 e 0 hanno lunghezza differente tra loro;
- le righe (non commentate) presentano sequenze di soli 0 (insieme vuoto).

Per i file generati in modo manuale, tutti i dettagli presentati nei commenti precedenti non sono presenti in quanto non generati in automatico, pertanto sarà l’utente a dover riconoscere autonomamente le caratteristiche per egli necessarie. Ciò non mina all’esecuzione corretta dell’algoritmo risolutivo di Exact Cover, il quale, inoltre, fornirà ulteriori informazioni utili in uscita, come riassunto nel prossimo paragrafo.

3. File di output

Come definito nella sezione [Output](#), gli algoritmi risolvitori del problema di Exact Cover forniscono in uscita un insieme contenente tutte le coperture esatte del dominio M data la collezione N . In particolare, tale insieme è rappresentato attraverso l'insieme COV , dove ciascuna partizione al suo interno è un elenco di identificatori (indici) degli elementi di N tali da essere copertura esatta di M .

I risultati dell'esecuzione delle soluzioni adottate per questo problema vengono automaticamente inseriti, attraverso il programma che le implementa, all'interno di un file di testo (in formato *.txt*) di uscita. Nell'immagine seguente è possibile osservare il contenuto del file di output generato dall'algoritmo EC sull'input dato come esempio della sezione precedente.

```
1 ;;; Cardinality of domain M: 5
2 ;;; Cardinality of collection N: 7
3
4 ;;; Partitions EC:
5 {3, 4, 5} => cardinality: 3
6 {2, 3, 7} => cardinality: 3
7 {3, 5, 6, 7} => cardinality: 4
8
9 ;;; Time to create partitions: 0.0005213969998294488s
10
11 ;;; Number of partitions produced by EC: 3
12 ;;; Partitions with cardinality 1: 0 => 0.0%
13 ;;; Partitions with cardinality 2: 0 => 0.0%
14 ;;; Partitions with cardinality 3: 2 => 66.667%
15 ;;; Partitions with cardinality 4: 1 => 33.333%
16
17 ;;; Number of visited nodes: 28
18 ;;; Number of possible nodes: 127
19 ;;; Percentage of visited nodes: 22.04724409448819%
20
21 ;;; Average cardinality for sets in N (avgN): 1.8571
22 ;;; Aggregate index (avgN/|M|): 0.3714
```

Il file di output di EC presenta, ordinatamente, le seguenti caratteristiche:

- la *cardinalità del dominio M e della collezione N* ;
- l'*elenco di tutte le partizioni prodotte dall'algoritmo*: viene presentata una partizione per riga, e ciascuna di esse (appartenente all'insieme COV) viene rappresentata come sequenza di indici racchiusa tra parentesi graffe. Il sottoinsieme di N relativo a ciascun indice della partizione è facilmente si può recuperare consultando il file "*indexes.txt*", se disponibile. Vicino ad ogni partizione, inoltre, viene resa nota la cardinalità della partizione stessa.
- il *tempo di esecuzione dell'algoritmo*: questo elemento è particolarmente importante per effettuare la comparazione dei tempi di esecuzione delle due versioni;
- il *numero totale di partizioni prodotte (cardinalità di COV) e distribuzione di cardinalità delle stesse*: su ogni riga relativa alla distribuzione viene mostrata anche la percentuale di partizioni di una determinata cardinalità rispetto al totale delle partizioni;
- il *numero di nodi visitati, il numero di nodi totali possibili e la percentuale di nodi visitati sul totale*: anche questi aspetti sono utili a comprendere quanti nodi sono stati visitati e, di conseguenza, di quanti nodi è stata evitata la visita considerando la totalità delle possibilità.
- la *cardinalità media degli insiemi presenti in N* ;
- l'*aggregate index*. Come citato in [File di input](#), l'*aggregate index* è un indice utile a comprendere, in modo riassuntivo, la distribuzione di cardinalità degli insiemi presenti in N . Permette anche, però, di giustificare l'andamento del risolutore.
 - Se tale valore è vicino a 1 significa che la maggior parte dei sottoinsiemi di N avrà cardinalità tendente ad $|M|$. Ciò implica anche che per il risolutore ci sarà minore probabilità di ottenere aggregati o (se presenti) partizioni di cardinalità ≥ 2 ;
 - Al contrario, per valori vicini a 0, si avrà una maggioranza di insiemi di N con cardinalità bassa (tendente a 1). Di conseguenza, si

avranno tanti insiemi di N in grado di creare aggregati e/o partizioni di cardinalità ≥ 2 .

Esso è particolarmente utile a comprendere maggiormente le motivazioni relative alla presenza o meno di un determinato numero di nodi visitati, del numero di partizioni (in *COV*) ottenute e del tempo relativo all'esecuzione degli algoritmi.

Sfruttare la distribuzione di cardinalità di N vera e propria, resa nota nel file di input, può rendere ancora più evidente, infine, il significato di tale indice.

In seguito viene invece rappresentato il file di output dell'algoritmo EC⁺.

```
1 ;;; Cardinality of domain M: 5
2 ;;; Cardinality of collection N: 7
3
4 ;;; Partitions EC+:
5 {3, 4, 5} => cardinality: 3
6 {2, 3, 7} => cardinality: 3
7 {3, 5, 6, 7} => cardinality: 4
8
9 ;;; Time to create partisions: 0.00047371999971801415s
10
11 ;;; Number of partitions produced by EC+: 3
12 ;;; Partitions with cardinality 1: 0 => 0.0%
13 ;;; Partitions with cardinality 2: 0 => 0.0%
14 ;;; Partitions with cardinality 3: 2 => 66.667%
15 ;;; Partitions with cardinality 4: 1 => 33.333%
16
17 ;;; Number of visited nodes: 28
18 ;;; Number of possible nodes: 127
19 ;;; Percentage of visited nodes: 22.04724409448819%
20
21 ;;; Average cardinality for sets in N (avgN): 1.8571
22 ;;; Aggregate index (avgN/|M|): 0.3714
23 ;;; Sets in N with cardinality 1: 3 => 42.857%
24 ;;; Sets in N with cardinality 2: 2 => 28.571%
25 ;;; Sets in N with cardinality 3: 2 => 28.571%
26 ;;; Sets in N with cardinality 4: 0 => 0.0%
27 ;;; Sets in N with cardinality 5: 0 => 0.0%
```

A meno di errori nella progettazione, entrambi i file di output devono presentare la stessa struttura e le stesse informazioni riguardo alla cardinalità di M ed N, al numero di nodi visitati e totali, alla lista di partizioni trovate, al numero e alla loro distribuzione di cardinalità. Le **principali differenze** stanno:

- nelle diciture “*EC*” ed “*EC+*”, utili ad identificare maggiormente gli algoritmi che hanno fornito tale risultato;
- nel tempo di esecuzione: essendo differenti, gli algoritmi implementati impiegheranno tempi diversi per risolvere il medesimo problema;
- nella presenza della distribuzione di cardinalità dei sottoinsiemi di N, presente solo dell’output di *EC+*.

Esistono **tre ulteriori casi in cui sono presenti delle differenze rispetto ad un output usuale dei due algoritmi**:

1. *Blocco da parte dell’utente*: quando l’utente decide di fermare l’esecuzione del programma per la produzione e scrittura sul file di output delle partizioni, viene inserito nel file di uscita un messaggio simile a quello presente nelle prossime immagini (relative rispettivamente agli algoritmi *EC* ed *EC+*).

```
;;; Partitioning in EC has been stopped by the user after 16.3290712670  
;;; Number of partitions produced by EC: 13998
```

```
;;; Partitioning in EC+ has been stopped by the user after 24.01373550800008s  
;;; Number of partitions produced by EC+: 33587
```

Si possono notare due periodi differenti, rappresentati il lasso di tempo trascorso tra l’inizio dell’esecuzione dell’algoritmo e il momento in cui il programma è stato bloccato. La riga che segue tali valori, ovvero quella relativa al numero di partizioni, è presente a dimostrare il fatto che, nonostante lo stop, i risultati fino a quel momento vengono inseriti comunque all’interno del file di output. L’esempio completo è reso disponibile [qui](#).

2. *Blocco per superamento della durata massima di esecuzione*: quando l'esecuzione del programma di produzione delle partizioni supera il tempo massimo prestabilito, viene presentato il seguente messaggio (sempre rispettivamente ai due algoritmi *EC* ed *EC+*).

```
;;; Time is up. Partitioning in EC has been stopped after 14.999895090999871s  
;;; Number of partitions produced by EC: 18455
```

```
;;; Time is up. Partitioning in EC+ has been stopped after 14.999873732000196s  
;;; Number of partitions produced by EC+: 20029
```

I lassi di tempo rappresentati indicano il tempo trascorso tra l'inizio dell'esecuzione e il momento in cui è stato raggiunto il limite temporale massimo fissato, che in entrambi i casi è di circa 15 secondi. Anche in questo caso, nonostante lo stop automatico, i risultati fino al momento di blocco vengono inseriti comunque all'interno del file di output. L'esempio completo è reso disponibile [qui](#).

Nella risoluzione del problema generale, il tempo massimo che è stato fissato per l'esecuzione di entrambe le versioni dell'algoritmo è di 3600 s.

3. *Numero di nodi totali possibili troppo elevato*: quando il numero di nodi che compongono l'albero è troppo elevato (situazione che si presenta facilmente all'aumentare della cardinalità di N), risulta difficile decifrare tale numero, così come la percentuale di nodi visitati su quelli totali risulta divenire approssimativamente zero. Per far sì che sia più semplice comprendere tali numeri ed evitare problemi nella stampa degli stessi, si è preferito adottare un formato come quello seguente:

```
;;; Number of visited nodes: 6414  
;;; Number of possible nodes: > 1e34  
;;; Exact number: 10384593717069655257060992658440191  
;;; Percentage of visited nodes: around 0%  
;;; Exact percentage: 6.176457331649865e-29%
```

Come si può osservare, il numero di nodi viene approssimato attraverso la notazione esponenziale “ $1eN$ ”, dove $N+1$ rappresenta il numero di cifre che compongono il numero di nodi totali. Per quanto riguarda la percentuale, invece, viene mostrata la dicitura “*around 0%*”, ad indicare il numero molto basso di nodi visitati rispetto al numero di nodi totali.

Se il valore del numero di nodi totali rispetta il range massimo di cifre possibili per una variabile intera stampabili, allora viene stampato comunque anche il valore esatto del numero di nodi e della percentuale dei visitati; se ciò non fosse possibile, viene stampato un messaggio ad indicare che il numero di nodi totali è troppo elevato e non può essere scritto, e che pertanto la percentuale di nodi visitati è ancora approssimativamente 0%.

Tale comportamento è reso visibile in entrambi gli esempi citati ai due punti precedenti.

4. Algoritmo base: EC ed ESPLORA

In questa sezione viene presentato lo pseudocodice relativo alla versione base dell'algoritmo risolvete del problema di Exact Cover. Esso è composto da due *procedure* denominate *EC* ed *ESPLORA*.

- *EC*: questa *procedure* rappresenta il programma principale. Essa riceve in ingresso come parametro la matrice $A_{|N|, |M|}$ ed esegue il ciclo iterativo principale, ovvero quello che permette la visita di tutti gli alberi radicati in i , con $1 \leq i \leq |N|$, per valori crescenti di i (ovvero secondo l'ordine lessicografico). Se l'insieme associato alla radice i non corrisponde né ad un insieme vuoto \emptyset , né al dominio M (caso per cui sarebbe una partizione di M), viene inizializzata la colonna i -esima di B ed eseguito un ulteriore ciclo che rende possibile, se le rispettive condizioni sono verificate, la visita di:
 - nodi associati ad aggregati di cardinalità 2 rispetto all'albero radicato in i , iterando su j con $1 \leq j \leq i-1$ seguendo l'ordine lessicografico. Ciò accade se, oltre alla condizione sopracitata, gli insiemi legati al nodo i -esimo e j -esimo hanno intersezione vuota tra loro (ovvero sono compatibili);
 - nodi associati ad aggregati di cardinalità > 2 discendenti da un aggregato di cardinalità 2 non associato ad un insieme equivalente ad M (l'insieme dell'aggregato padre non è una partizione). Ciò accade se, oltre a quanto detto, vi è intersezione non vuota tra $B[1..j-1, i]$ e $B[1..j-1, j]$, ovvero se vi è compatibilità a due a due tra l'insieme di indice i in N , l'insieme di indice j in N e almeno uno degli insiemi di N con indice compreso tra 1 e $j-1$. Di tali nodi discendenti si occupa la *procedure ESPLORA*.

Quando un insieme associato ad un nodo, aggregato o meno, risulta equivalente al dominio M , allora rappresenta una partizione, e pertanto viene aggiunto all'insieme *COV* delle partizioni come lista di indici che identificano gli insiemi componenti tale partizione.


```

1: procedure EC( $A$ )                                ▷ programma principale,  $A$  è la matrice d'ingresso
2:   for  $i \leftarrow 1$  to  $rows[A]$  do
3:     if  $A[i] == \emptyset$  then
4:       break                                       ▷ break termina l'iterazione  $i$ -ma
5:     if  $A[i] == M$  then
6:       inserire  $\{i\}$  nell'insieme delle partizioni COV, inizialmente vuoto
                                                    ▷ COV è una variabile globale
7:       break
8:       in  $B$  aggiungere la colonna relativa a  $i$ 
9:       for  $j \leftarrow 1$  to  $i - 1$  do
10:        if  $A[j] \cap A[i] \neq \emptyset$  then
11:           $B[j, i] \leftarrow 0$ 
12:        else
13:           $I \leftarrow \{i, j\}$ ,  $U \leftarrow A[i] \cup A[j]$ 
14:          if  $U == M$  then
15:            inserire  $I$  in COV
16:             $B[j, i] \leftarrow 0$ 
17:          else
18:             $B[j, i] \leftarrow 1$ 
19:             $Inter \leftarrow B[1..j - 1, i] \cap B[1..j - 1, j]$ 
20:            if  $Inter \neq \emptyset$  then
21:              ESPLORA( $I, U, Inter$ )

```

Nell'immagine precedente è visibile, in modo dettagliato, il contenuto dello pseudocodice di *EC*. I principali elementi coinvolti sono:

- i : indice della radice, valore su cui viene eseguita l'iterazione principale o esterna;
- j : indice utilizzato per l'iterazione interna, tramite cui è possibile lavorare su aggregati $\{i, j\}$ di cardinalità 2;
- $rows[A]$: numero di righe della matrice A , corrispondenti a $|N|$;
- $A[i]$ ($A[j]$): riga i -esima (j -esima) di A , corrispondente all'insieme i -esimo (j -esimo) in N ; essa è una sequenza di 1 e 0, i cui casi limite sono la sequenza di tutti 0, indicata come \emptyset , e quella di tutti 1, indicata con M ;
- B : matrice di compatibilità (simmetrica di ordine $|N|$);
- $B[j, i]$: elemento sulla riga j e colonna i di B , che presenta un 1 se $A[i]$ ed $A[j]$ sono compatibili, 0 altrimenti;
- I : coppia $\{i, j\}$ rappresentante un aggregato di cardinalità 2 come elenco di indicatori degli insiemi di N che li compongono (i e j);
- U : unione degli insiemi $A[i]$ e $A[j]$, intesa come sequenza di 1 e 0 di cardinalità $|M|$ tale per cui si ha, in una determinata posizione k ($1 \leq k \leq |M|$) della sequenza, un 1 ogni volta in cui l'elemento di

posizione k in $A[i]$ è differente dall'elemento di posizione k in $A[j]$, 0 altrimenti;

- *Inter*: intersezione tra $B[1..j-1, i]$ e $B[1..j-1, j]$, intesa come sequenza di 1 e 0 di cardinalità ' $j-1$ ' tale per cui si ha, in una determinata posizione k ($1 \leq k \leq j-1$) della sequenza, un 1 ogni volta in cui entrambi gli elementi $B[k][i]$ e $B[k][j]$ sono uguali a 1, e 0 altrimenti. La presenza di un 1 implica quindi la compatibilità a due a due tra gli insiemi $A[i]$, $A[j]$ e $A[k]$, e la possibilità di ottenere un aggregato $\{i, j, k\}$ di cardinalità 3;
 - *COV*: insieme delle partizioni in uscita, presentate come elenco tra parentesi graffe di indici di elementi di N componenti ciascuna copertura esatta.
- **ESPLORA**: questa è una *procedure* di tipo ricorsivo che viene chiamata dalla *procedure EC* quando sono stati trovati nodi corrispondenti ad aggregati di cardinalità > 2 composti da insiemi compatibili a due a due. Su questi nodi è possibile verificare ulteriormente la possibilità di trovare nuove partizioni o ulteriori aggregati di cardinalità ancora maggiore. I parametri in input di *ESPLORA*, sono:
- *I*: elenco di indicatori, tra parentesi graffe, di insiemi di N componenti l'aggregato di partenza, ovvero quello a cui andrà unito un nuovo insieme compatibile per verificare se vi sarà o meno una nuova partizione;
 - *U*: unione degli insiemi componenti l'aggregato di partenza;
 - *Inter*: intersezione tra *tutti* i $B[1..x_i-1, y]$ con $1 < x_i \leq y$, dove $x_i \in I$ è uno degli indici contenuti in *I* relativo ad un sottoinsieme di N ($A[x]$), mentre $y \in I$ è l'indice del sottoinsieme di N ultimo per ordine lessicografico ($A[y]$). Dato k ($1 \leq k \leq x-1$) tale che *tutti* i $B[k][x_i]$ e $B[k][y]$ siano uguali a 1, la sequenza di 1 e 0 componente *Inter* avrà in posizione z il valore 1, e pertanto gli insiemi $A[y]$, $A[k]$ e tutti gli $A[x_i]$ saranno compatibili a due a due, e quindi in grado di produrre aggregati di cardinalità maggiore rispetto ad *I*. *Inter*

racchiude, inoltre, tutti i valori di k tali da produrre i nuovi aggregati.

Viene visitata in ordine lessicografico tutta la lista di insiemi compatibili a I , ovvero vengono eseguite ciclicamente delle operazioni per ogni indice k presente in $Inter$. In particolare, per ogni nuovo aggregato generato con I e k viene verificato se si produce una nuova partizione. Se ciò non accade e vi è intersezione non vuota tra $Inter$ e $B[1..k, k]$, ovvero vi è nuovamente compatibilità tra gli insiemi dell'aggregato iniziale e il nuovo insieme legato a k , allora viene effettuata una nuova chiamata ricorsiva di *ESPLORA*, in modo da andare a verificare se vi siano nuove partizioni di cardinalità ancora maggiore o meno.

```
1: procedure ESPLORA( $I, U, Inter$ )
2:   for all  $k \in Inter$ , in ordine lessicografico del valore di  $k$  do
3:      $Itemp \leftarrow I \cup \{k\}$ ,  $Utemp \leftarrow U \cup A[k]$ 
4:     if  $Utemp == M$  then
5:       inserire  $Itemp$  in COV
6:     else
7:        $Intertemp \leftarrow Inter \cap B[1..k - 1, k]$ 
8:       if  $Intertemp \neq \emptyset$  then
9:         ESPLORA( $Itemp, Utemp, Intertemp$ )
```

Nell'immagine precedente è visibile il contenuto dello pseudocodice di *ESPLORA*. I principali elementi coinvolti, oltre a quelli citati in precedenza, sono:

- k : rappresenta un indice di $Inter$, ovvero un elemento associato ad un insieme di N tale da comporre un aggregato con I da analizzare;
- $Itemp$: unione tra I e $\{k\}$, intesa come aggregato sotto forma di elenco di identificatori degli insiemi di N che compongono I e k ;
- $Utemp$: unione tra U e $A[k]$, intesa come sequenza di 1 e 0 di cardinalità $|M|$ tale per cui si ha, in una determinata posizione p ($1 \leq p \leq |M|$) della sequenza, un 1 ogni volta in cui l'elemento di posizione p in U è differente dall'elemento di posizione p in $A[k]$, 0 altrimenti;

- *Intertemp*: intersezione tra *Inter* e $B[1..k-1, k]$, intesa come sequenza di 1 e 0 di cardinalità $k-1$ tale per cui si ha, in una determinata posizione p ($1 \leq p \leq k-1$) della sequenza, un 1 ogni volta in cui entrambi gli elementi $Inter[p]$ e $B[p][k]$ sono uguali a 1, e 0 altrimenti. La presenza di un 1 implica quindi la compatibilità a due a due tra gli insiemi di *Inter*, $A[k]$ e $A[p]$, e la possibilità di ottenere un aggregato di cardinalità ancora maggiore.

Anche in questo caso, se un insieme associato ad un nodo visitato risulta equivalente al dominio M , esso allora rappresenta una partizione, e pertanto viene aggiunto all'insieme *COV* delle partizioni come lista di identificatori componenti la partizione.

Un esempio di esecuzione dell'algoritmo base è disponibile [qui](#). In esso vengono mostrati la matrice $A_{|N|, |M|}$, la matrice $B_{|N|, |N|}$ e la visita di tutti gli alberi radicati in i con $1 \leq i \leq |N|$, compresa la costruzione di *COV*.

5. Algoritmo plus: EC^+ ed $ESPLORA^+$

In questa sezione viene presentato lo pseudocodice relativo alla seconda versione dell'algoritmo risolvante del problema di Exact Cover, chiamato per semplicità algoritmo “plus”. Questa nuova versione si basa sulla medesima logica di esplorazione degli alberi di quella precedente, visitando esattamente gli stessi nodi, e ne ricalca la struttura di controllo. La differenza sostanziale sta nel fatto che, in caso di aggregati di insiemi, non si va più a verificare se l'unione di tali insiemi coincida con il dominio M , ma piuttosto se la somma delle cardinalità degli stessi coincida con $|M|$. Infatti, se più insiemi sono compatibili e la loro unione ha cardinalità $|M|$, si avrà a che fare con una nuova partizione da aggiungere a COV .

In questa versione, pertanto, viene evitata la computazione dell'unione di insiemi, la quale viene sostituita dal calcolo e dalla memorizzazione della cardinalità di ciascun insieme della collezione.

I due controlli, quello effettuato dalla prima versione e quello effettuato dalla seconda, si equivalgono per aggregati di 3 o più insiemi, dal momento che, per costruzione, coinvolgono solo insiemi tutti reciprocamente disgiunti. In caso di aggregati di cardinalità 2, il controllo viene effettuato solo dopo aver appurato che essi siano disgiunti.

Esso è composto da due *procedure* denominate EC^+ ed $ESPLORA^+$. La funzione di ciascuna di esse si mantiene indifferente rispetto alla versione base dell'algoritmo, mentre le differenze a livello di pseudocodice sono evidenziate nelle immagini sottostanti.

In particolare, per EC^+ (presentato nell'immagine sottostante) abbiamo:

- $card[i]$: cardinalità dell'insieme $A[i]$, indicata come $|A[i]|$, dove i è l'indice dell'elemento i -esimo della collezione N . Questa cardinalità è equivalente al numero di 1 presenti nella riga i -esima di A ;
- $cardU$: cardinalità dell'unione tra l'insieme $A[i]$ e l'insieme $A[j]$, dove i e j sono gli indici degli insiemi di N che compongono l'aggregato di cardinalità

2. Essendo stata verificata precedentemente la presenza di un'intersezione nulla tra i due insiemi, l'unione si può rappresentare come somma delle cardinalità dei due insiemi, e quindi come somma del numero di 1 presenti nella riga i -esima e j -esima di A . Questa operazione va a sostituirsi alla computazione dell'unione tra i due insiemi, eseguendo così solamente una somma di elementi. Inoltre, se tale somma corrisponde a $|M|$, l'aggregato di cardinalità 2 è una partizione, e pertanto va aggiunto a COV ;

```

1: procedure  $EC^+(A)$  ▷  $A$  è la matrice d'ingresso
2:   for  $i \leftarrow 1$  to  $rows[A]$  do
3:     if  $A[i] == \emptyset$  then
4:       break ▷ break termina l'iterazione  $i$ -ma
5:     if  $A[i] == M$  then
6:       inserire  $\{i\}$  nell'insieme delle partizioni  $COV$ , inizialmente vuoto
▷  $COV$  è una variabile globale
7:     break
8:      $card[i] \leftarrow |A[i]|$ 
9:     in  $B$  aggiungere la colonna relative a  $i$ 
10:    for  $j \leftarrow 1$  to  $i - 1$  do
11:      if  $A[j] \cap A[i] \neq \emptyset$  then
12:         $B[j, i] \leftarrow 0$ 
13:      else
14:         $I \leftarrow \{i, j\}$ ,  $cardU \leftarrow card[i] + card[j]$ 
15:        if  $cardU == |M|$  then
16:          inserire  $I$  in  $COV$ 
17:           $B[j, i] \leftarrow 0$ 
18:        else ▷ se si esegue questa sezione è sicuramente  $cardU < |M|$ 
19:           $B[j, i] \leftarrow 1$ 
20:           $Inter \leftarrow B[1..j - 1, i] \cap B[1..j - 1, j]$ 
21:          if  $Inter \neq \emptyset$  then
22:             $ESPLORA(I, \text{cardU}, Inter)$ 

```

Invece, per quanto riguarda $ESPLORA^+$, rappresentato nell'immagine qui sotto, abbiamo:

- $cardU$: parametro di ingresso della *procedure* $ESPLORA^+$ (sostituto di U in $ESPLORA$ della versione base), che rappresenta la somma delle cardinalità di tutti gli insiemi componenti l'aggregato di partenza, e quindi la somma di tutti gli 1 presenti nelle righe di A relative agli insiemi di tale aggregato;
- $cardtemp$: somma tra $cardU$ e $|A[k]|$, dove $A[k]$ è l'insieme relativo all'indice k , dove $k \in Inter$ rappresenta uno degli indici degli insiemi

compatibili a quelli contenuti in I e quindi tali da costituire un nuovo aggregato con gli elementi di I . Questo valore va a sostituirsi alla costruzione di $Utemp$ in *ESPLORA* della versione base, evitando così la computazione dell'unione e sfruttando solamente un'operazione di somma. Anche in questo caso, se $cardtemp$ risulta equivalente a $|M|$ significa che l'aggregato composto dagli insiemi di I e l'insieme indicato da k è una partizione, e che quindi va aggiunto a COV .

```

1: procedure ESPLORA+( $I$ ,  $cardU$ ,  $Inter$ )
2:   for all  $k \in Inter$ , in ordine lessicografico del valore di  $k$  do
3:      $Itemp \leftarrow I \cup \{k\}$ ,  $cardtemp \leftarrow cardU + |A[k]|$ 
4:     if  $cardtemp == |M|$  then
5:       inserire  $Itemp$  in  $COV$ 
6:     else
7:        $Intertemp \leftarrow Inter \cap B[1..k-1, k]$ 
8:       if  $Intertemp \neq \emptyset$  then
9:         ESPLORA( $Itemp$ ,  $cardtemp$ ,  $Intertemp$ )

```

Un esempio di esecuzione dell'algoritmo plus è disponibile [qui](#). Come per il caso base, in esso vengono mostrati la matrice $A_{|N|, |M|}$, la matrice $B_{|N|, |N|}$ e la visita di tutti gli alberi radicati in i con $1 \leq i \leq |N|$, compresa la costruzione di COV .

6. Implementazione

In questa sezione vengono presentate tutte le scelte implementative effettuate per la risoluzione del problema di Exact Cover e, in generale, per adempiere a tutti i requisiti e a tutti i compiti richiesti per la costruzione del software. Sono presenti, inoltre, informazioni utili a comprendere la scelta del linguaggio di programmazione, l'ambiente di sviluppo utilizzato e tutte le istruzioni utili ad eseguire i programmi forniti.

6.1 Linguaggio di programmazione e ambiente di sviluppo

Il linguaggio di programmazione utilizzato per lo sviluppo del software di risoluzione del problema di Exact Cover e di tutti i programmi ad esso associati è *Python 3.7.15*. Oltre ad essere uno dei principali linguaggi di programmazione utilizzati oggi, *Python* è stato scelto in quanto:

- utilizza un codice molto intuitivo e leggibile, risultando così moderno e semplice da apprendere;
- è completamente gratuito, utilizzabile e distribuibile senza restrizioni di copyright;
- è multi-paradigma, supporta sia la programmazione procedurale che la programmazione ad oggetti, oltre ad ulteriori elementi quali iteratori e generatori;
- è un linguaggio interpretato, quindi lo stesso codice può essere eseguito su qualsiasi piattaforma purché abbia l'interprete *Python* installato. Nonostante ciò, i programmi vengono automaticamente compilati in formato bytecode prima di essere eseguiti, quindi in un formato compatto ed efficiente che garantisce elevate prestazioni;
- è un linguaggio di alto livello al tempo stesso semplice e potente, la cui sintassi, i diversi moduli e funzioni già inclusi sono consistenti, intuitivi e facili da imparare;

- è ricco di librerie, tra cui quelle standard già incluse nell'installazione e i moduli aggiuntivi creati e mantenuti da una comunità molto attiva;
- adotta un meccanismo di garbage collection che si occupa automaticamente dell'allocazione e del rilascio della memoria. Ciò permette al programmatore di usare variabili liberamente, senza preoccuparsi di dichiararle e di allocare e rilasciare spazi di memoria manualmente.

Inoltre, il linguaggio *Python* utilizza la tipizzazione dinamica delle variabili, ovvero non rende necessario dichiarare il tipo di variabili (numerico, alfanumerico, ecc.) in quanto l'interprete è in grado di riconoscerlo automaticamente dal suo contenuto durante l'assegnazione. Così facendo, è permesso cambiare il tipo di dato di una variabile semplicemente tramite una nuova assegnazione, fattore che (se sfruttato correttamente e adeguatamente) permette di riutilizzare spazio di memoria occupato da variabili non più utilizzate.

L'ambiente di sviluppo che è stato scelto per sfruttare il linguaggio *Python* e sviluppare i software è *Colaboratory*. Chiamato anche in breve "*Colab*", esso è un prodotto di *Google Research* che permette a chiunque di scrivere ed eseguire codice Python arbitrario tramite il browser. In particolare, *Colab*:

- è un *IDE web-based* che sfrutta browser quali *Google Chrome* e *Mozilla Firefox* per essere utilizzato;
- è un prodotto *Jupyter Notebook-like*, ovvero permette di creare e condividere documenti testuali interattivi contenenti oggetti quali equazioni, grafici e codice sorgente eseguibile, come accade per i notebook *Jupyter*, ma limitato alla programmazione in *Python*; ciò significa che è possibile scrivere ed eseguire codice direttamente all'interno dei blocchi componenti il notebook, visualizzare subito i risultati ottenuti, produrre grafici e aggiungere blocchi di testo come un blocco appunti, per la buona presentazione di progetti da supervisionare;
- è ospitato su cloud, pertanto non è necessario installare nulla sul proprio dispositivo, come non è necessario aggiornare l'hardware per soddisfare i

requisiti di carico di lavoro di CPU/GPU di *Python*. Essendo un servizio di *Google*, permette facilmente di salvare e condividere il notebook tramite *Google Drive*. Inoltre, permette un'integrazione facile con *GitHub*;

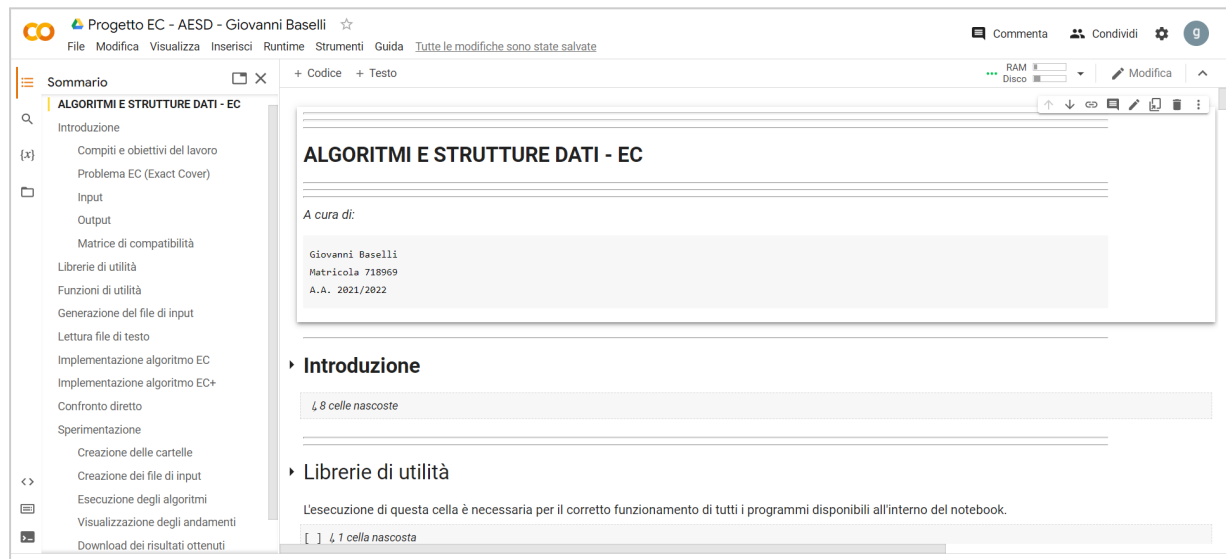
- offre un accesso libero e gratuito ad un'intera infrastruttura informatica composta da storage, memoria, capacità di elaborazione, GPU e TPU. Tuttavia, esistono delle limitazioni per quanto riguarda tali risorse, come definito [qui](#). Gli utenti, in generale, possono eseguire i notebook per un massimo di 12 ore;
- viene interamente eseguito da una Virtual Machine con sistema operativo Linux, e il runtime della GPU viene fornito con CPU Intel Xeon a 2,20 GHz, 13 GB di RAM, acceleratore Tesla K80 e 12 GB GDDR5 VRAM (secondo quanto fornito dalle specifiche).

Tutti questi aspetti rendono *Colab* uno strumento potente ed efficiente per la programmazione. In particolare, per il caso in esame è stato possibile sfruttare il notebook Colab per raccogliere in un unico documento tutti i software richiesti, dalla generazione del file di input, alla lettura ed esecuzione dei programmi per l'algoritmo base e plus di Exact Cover, fino alla sperimentazione finale.

6.2 Struttura del notebook ed esecuzione


Il notebook presenta il formato '*ipynb*', il cui nome completo è *IPython Notebook Format*, ovvero il formato utilizzato per costruire i notebook classificati come *Jupyter-like*. Questo tipo di file può essere aperto tramite *Jupyter Notebook* (cross-platform), *Jupyter Notebook Viewer* (web), *Cantor* (Linux) o *Google Colaboratory* (web). Per quanto riguarda la **visualizzazione** del contenuto del notebook, è possibile utilizzare uno qualsiasi dei programmi sopra specificati. Tuttavia, per l'**esecuzione** dei blocchi di codice è **necessario** utilizzare direttamente *Google Colab*, in quanto il codice presenta funzioni appartenenti al modulo '*google.colab*' (quali il download diretto dei file sul proprio dispositivo) e funzioni disponibili solo su sistema operativo Linux (come la funzione *signal.alarm()* per fermare l'esecuzione del codice dopo un tempo prestabilito).

Oltre a tutti gli aspetti su cui ci si è appena soffermati, Colab rende molto semplice la condivisione dei notebook grazie a Google Drive. Infatti, è possibile accedere alla soluzione di questo progetto dalla cartella Drive disponibile [qui](#) facendo doppio click sul file ‘Progetto EC - AESD - Giovanni Baselli’, oppure direttamente a questo [link](#).



Nell'immagine sopra si può osservare la schermata completa di Colab relativa al progetto. Sulla sinistra si nota in particolare il *somario* del notebook, il quale delinea la struttura di tutto il documento. Sulla destra è invece presente il notebook effettivo. Per rendere visibile il contenuto di ciascuna sezione è necessario premere il triangolo nero affianco a ciascun titolo, oppure premere il riquadro con il numero di celle nascoste. Così facendo, si espande la sezione ed è possibile consultare tutti i blocchi disponibili al suo interno.

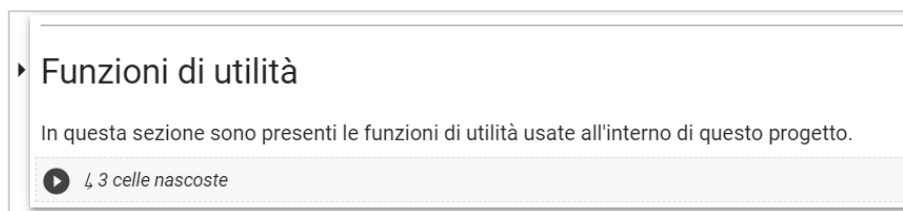
I *blocchi* (o *celle di testo*) permettono di scrivere e visualizzare del testo. Tramite queste è possibile costruire adeguatamente un documento consultabile, ben strutturato e commentato.

Le *celle di codice* contenute in ogni sezione possono essere eseguite una alla volta premendo sul simbolo  che si mostra quando è presente il mouse sull'area della cella. Nell'immagine è presente un esempio indicativo.



Una cella correttamente eseguita è contrassegnata da ✓, mentre lo stop per errore da ✗. Se è presente la stampa di eventuali risultati, direttamente al di sotto della cella sarà possibile visualizzare gli stessi. Ciò può accadere, ad esempio, in presenza della funzione *print()* o della stampa di grafici.

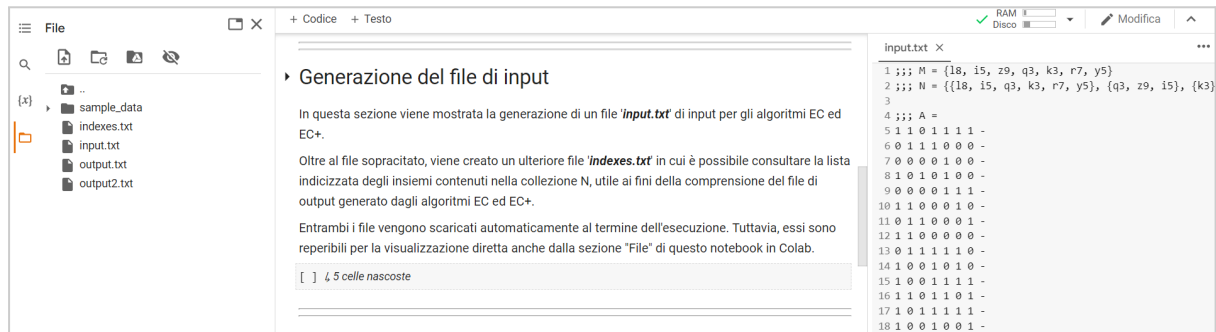
Per evitare di premere una cella alla volta per eseguire i blocchi di una sezione, è possibile premere direttamente ▶ nel riquadro con il numero di celle nascoste quando la sezione è ancora compressa, come si può vedere nella seguente immagine. Ciò rende molto più veloce l'esecuzione di blocchi che mirano ad uno stesso obiettivo. Nel caso d'esempio dell'immagine, infatti, si effettua l'esecuzione automatica di tutti i blocchi per la definizione delle funzioni di utilità presenti nella sezione compressa “*Funzioni di utilità*” premendo il suddetto pulsante.



I programmi sviluppati coinvolgono, inoltre, una serie di file che vengono generati automaticamente all'interno del notebook. Questi file sono direttamente consultabili grazie al file manager 📁 presente sulla sinistra della schermata di Colab. Qui è possibile visualizzare i file di testo (a meno di file di dimensione consistente), scaricarli sul proprio dispositivo, eliminarli dal runtime o caricare file esterni (ad esempio, file di input *.txt* scritti manualmente o generati e scaricati da un'esecuzione precedente dei programmi). Inoltre, utilizzando i

metodi del modulo ‘*google.colab*’ è possibile effettuare il download automatico dei file direttamente sul dispositivo che si sta utilizzando.

Nell’immagine sottostante sono visibili sia la sezione “*File*” (sulla sinistra), sia un esempio di file di input aperto direttamente in Colab (sulla destra).




Tutto ciò rende quindi evidente la semplicità e la comodità della piattaforma Google Colab per produrre documenti completi, ben strutturati, con codice eseguibile e con risultati semplici da consultare.

6.2.1 Ordine di esecuzione

Grazie al sommario del notebook del progetto, si rendono evidenti le sezioni che compongono la risoluzione software del problema. Tale suddivisione in “capitoli” non rappresenta solamente la struttura del codice, ma anche l’ordine logico di esecuzione del codice. In particolare, togliendo la sezione introduttiva testuale utile alla comprensione del problema generale di Exact Cover, si hanno i seguenti passi *da eseguire in ordine*:

1. **Librerie di utilità:** *import* di tutte le librerie utili all’interno del notebook;
2. **Funzioni di utilità:** contiene tutte le funzioni non di libreria utilizzate all’interno del codice;
3. **Generazione del file di input:** rappresenta il programma che si occupa della generazione pseudo-casuale del file di input, come richiesto dalla consegna del progetto;

4. **Lettura del file di testo:** contiene tutte le operazioni per la lettura del file di testo, che sia esso generato in automatico dal programma o aggiunto manualmente dall'utente;
5. **Implementazione algoritmo EC:** rappresenta il programma che si occupa dell'implementazione delle procedure *EC* ed *ESPLORA*, oltre che della generazione del file di output della versione base dell'algoritmo, che utilizza come input il file letto dalla sezione precedente;
6. **Implementazione algoritmo EC+:** rappresenta il programma che si occupa dell'implementazione delle procedure *EC*⁺ ed *ESPLORA*⁺, oltre che della generazione del file di output della versione plus dell'algoritmo, che utilizza come input il medesimo file letto al *punto 4* ed utilizzato dall'algoritmo al *punto 5* di questa lista;
7. **Confronto diretto:** rappresenta il programma che si occupa del confronto diretto dei risultati ottenuti in seguito all'esecuzione degli algoritmi base e plus su uno stesso input, producendo una tabella riassuntiva basata sulle differenze tra i valori di variabili generati dagli algoritmi stessi;
8. **Sperimentazione:** rappresenta il programma che si occupa della sperimentazione. In particolare, al suo interno vengono create le cartelle contenenti i risultati, vengono creati i file di input ed eseguiti gli algoritmi utilizzando le procedure implementate ai punti precedenti (i cui file generati vengono inseriti nelle rispettive cartelle), vengono visualizzati i grafici delle differenze presenti nei due algoritmi e infine scaricati tutti i risultati in un unico file *.zip*.

Per la corretta esecuzione di tutti i programmi è pertanto importante seguire l'ordine stabilito da questo elenco. Ogni elemento della lista identifica una sezione compressa del notebook; essendo ciascuna di esse composta da codice eseguibile, è possibile eseguire direttamente tutte le celle contenute tramite il pulsante  del riquadro con la dicitura “# celle nascoste” (vedasi l'esempio nell'immagine seguente).



In questo modo è possibile effettuare l'esecuzione di tutti i punti della lista indicata andando semplicemente ad eseguire in ordine le sezioni comprese del notebook il cui titolo è il medesimo di uno degli elementi della lista.

Nei capitoli seguenti di questo elaborato si entrerà più nello specifico per quanto riguarda le scelte implementative effettuate per i programmi principali, sia dal punto di vista delle strutture dati e variabili utilizzate che dal punto di vista delle operazioni svolte al loro interno.

6.3 Generazione del file di input

Una delle richieste per la stesura di questo progetto è la realizzazione di un *programma per la generazione del file di input* per il problema di Exact Cover, dove il file segue quanto riportato nelle sezioni [Input](#) e [File di Input](#). In particolare, si vuole sviluppare un programma software che costruisca tale file di input, per entrambe le versioni dell'algoritmo, in modo pseudo-casuale, dove ciascun file generato deve contenere la rappresentazione di collezioni aventi dimensioni e caratteristiche diverse in termini di $|N|$ ed $|M|$, distribuzione di cardinalità degli insiemi e quant'altro. Questi aspetti sono utili a rilevare gli andamenti delle prestazioni dei due risolutori al variare di tali caratteristiche.

Il codice del programma che si occupa della produzione del file di input è reperibile alla sezione “**Generazione del file di input**” del notebook Colab del progetto. Esso può essere suddiviso e descritto in tre parti, presentate di seguito.

1. Generazione del dominio M e della collezione N

La cardinalità del dominio M viene scelta in modo pseudo-casuale. Per la generazione del dominio M è stato utilizzato un **set**, in quanto i set sono collezioni che non accettano duplicati, aspetto che impedisce la presenza in M di elementi uguali fra loro. In particolare, ciascun elemento di M , come precedentemente specificato, è rappresentato da una coppia lettera-cifra araba,

dove la lettera (dalla a alla z) e la cifra (da 1 a 9) sono anch'esse scelte in modo pseudo-casuale.

Anche per la cardinalità di N viene generato un numero pseudo-casuale. Tuttavia, in questo caso viene utilizzato il concetto di *combinazione semplice*. In genere, una combinazione semplice (o senza ripetizione) è un raggruppamento di k elementi distinti selezionati tra n elementi distinti, con $k \leq n$ e nell'ipotesi che l'ordine con cui gli elementi si susseguono sia irrilevante. Nel caso in esame, si vuole che un insieme appartenente ad N sia esattamente una combinazione semplice di k elementi distinti selezionati all'interno di M ; in particolar modo, per insiemi della medesima cardinalità k si vuole che differiscano tra loro per almeno un elemento, e non per l'ordine. Infatti, l'ordine degli elementi di un insieme è irrilevante, e pertanto insiemi che presentano il medesimo contenuto ma in ordine differente sono classificati comunque come coincidenti, e quindi vanno esclusi dall'insieme N .

Il numero di combinazioni semplici di classe k di n elementi distinti si indica con $C_{n,k}$ ed è dato dal coefficiente binomiale di n su k , come rappresentato nella formula seguente:

$$C_{n,k} = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Il numero massimo di combinazione semplici di n possibili è dato dalla somma del numero di combinazioni semplici di n di classe i che va da 1 a n , come indicato nella formula:

$$\sum_{i=1}^n C_{n,i} = \sum_{i=1}^n \binom{n}{i}$$

Per non superare il numero massimo di possibilità di combinazioni semplici tra gli elementi di M dati, viene scelto come cardinalità di N un numero pseudo-casuale che sia compreso tra 1 e il numero massimo di combinazioni semplici di n , dove $n = |M|$, ovvero il numero massimo di combinazioni semplici

degli elementi del dominio M di tutte le classi da 1 a $|M|$ al fine di generare sottoinsiemi distinti di N .

Anche la collezione N è un **set**, in quanto tutti gli insiemi al suo interno devono essere distinti tra loro. Ciascun elemento da aggiungere ad N , essendo una combinazione semplice di elementi di M , è estratto come campione (di M) di dimensione pseudo-casuale k . Si è scelto di estrarre solamente campioni di dimensione compresa tra 1 ed $|M|$, in modo da escludere automaticamente l'insieme vuoto che, nelle partizioni in uscita, su richiesta delle specifiche del progetto andrebbe omissis.

Per fare sì che vi sia una distribuzione di cardinalità dei campioni equilibrata, ovvero che vengano generati insiemi di elementi di M che coprano adeguatamente più o meno tutte le cardinalità comprese tra la quella minima (1) e quella massima ($|M|$), si producono per un *prima parte* della dimensione totale di N solamente insiemi di cardinalità compresa tra $\frac{1}{2}|M|$ ed $|M|$, mentre per la *seconda parte* campioni di cardinalità compresa tra 1 e $\frac{1}{2}|M|$. Nel contempo, per far sì che si ottengano distribuzioni differenti tra un'esecuzione e l'altra, si è scelto di fare in modo che le due parti distinte di N che vengono costruite varino sempre nel numero di elementi che le compongono, sfruttando nuovamente la pseudocasualità: ogni volta in cui si vuole aggiungere un nuovo valore al set N , viene estratto a sorte un numero intero compreso tra un minimo di $\frac{1}{3}|N|$ (presente nella variabile $lmin$) e un massimo di $|N|$ (nella variabile $lmax$), e questo valore andrà ad essere confrontato con la lunghezza del set N nella sua fase di costruzione per spartire i nuovi insiemi da aggiungere tra quelli di cardinalità $[\frac{1}{2}|M|, |M|]$ e quelli di cardinalità $[1, \frac{1}{2}|M|]$. Per far sì che vi siano ulteriori possibilità per distribuire la cardinalità degli insiemi in N , è possibile modificare i limiti sopracitati a proprio piacimento all'interno del codice: diminuendo adeguatamente il limite minimo ($lmin$) e il limite massimo ($lmax$) si avrà un maggior numero di insiemi di N di cardinalità più bassa (cardinalità media in N più bassa); aumentandone il valore, invece, si avrà un maggior numero di insiemi di N di cardinalità più alta (cardinalità media in N più alta).

Ciascun elemento di N deve essere differente dagli altri, sia dal punto di vista del contenuto che dell'ordine. La collezione N andrebbe considerata quindi come un **set di set**, ovvero un insieme di insiemi distinti e in cui ogni insieme è composto da elementi distinti di M . Per poter costruire tale struttura, non è possibile aggiungere semplicemente dei nuovi set al set N , perché un set richiede degli oggetti immutabili al suo interno, e un set non è considerato immutabile per la possibilità di aggiungere o eliminare elementi dallo stesso. Pertanto, si è deciso di ricorrere ai **frozenset**, ovvero dei set immodificabili. Utilizzare il frozenset permette in automatico di evitare la presenza di set uguali (seppur in ordine differente) in quanto un frozenset eredita dai set la mancanza di un ordinamento dei valori al suo interno; quindi, durante l'aggiunta al set N del nuovo frozenset, la funzione `add()` non valuterà tale frozenset come idoneo se N contiene già un altro frozenset identico nel contenuto.

In seguito alla costruzione di tutti i set, si è scelto di trasformare tali set in **tuple**. Queste strutture dati hanno la caratteristica di essere immodificabili ed ordinate; in particolare, quest'ultimo aspetto è importante in quanto ogni elemento all'interno degli insiemi M ed N deve essere identificato da un indice che va a specificare un ordine (cosiddetto lessicografico). Si è preferito optare per tale struttura rispetto ad una lista in *Python* in quanto, a parità di contenuto, le tuple sono più veloci quando si ha a che fare con cicli e iterazioni, sono più appropriate per operazioni di accesso al loro contenuto e occupano meno memoria. Inoltre, la tipizzazione dinamica di *Python* non rende necessarie nuove dichiarazioni: il passaggio da set a tupla viene effettuato assegnando la tupla generata (dal set) alla medesima variabile, evitando di occupare ulteriore spazio di memoria per una nuova variabile.

Subito dopo viene costruita la distribuzione di cardinalità degli insiemi di N . In particolare, viene utilizzata una **lista** i cui indici rappresentano la cardinalità dell'insieme, mentre l'elemento contenuto nella cella relativa ad un certo indice rappresenta il numero di ricorrenze di insiemi di N che possiedono la cardinalità identificata da tale indice. In questo caso si è scelta la lista in quanto struttura

dati ordinata e mutabile, entrambi aspetti necessari in questo caso. Tale lista viene costruita ed aggiornata grazie alla funzione di utilità *cardDistribution*.

2. Scrittura del file *'input.txt'*

Il file d'ingresso *'input.txt'* viene generato passo per passo dal programma. Esso viene creato e aperto in scrittura: se *'input.txt'* è già presente, tale file viene totalmente sovrascritto, altrimenti ne viene creato uno nuovo.

Ricordando quanto definito nella sezione [File di input](#), vengono inseriti nel file, in ordine, la prima, la seconda e la terza macroarea. In particolare, per evitare di occupare troppa memoria, non viene costruita (e quindi salvata in memoria) l'intera matrice A per poi essere scritta sul file, ma prodotta e poi stampata una riga di A alla volta. Ogni riga è pertanto una **lista**, la quale viene formata attraverso una *List Comprehension*, ovvero una tecnica di costruzione delle liste in Python che traduce un'iterazione tradizionale che utilizza il ciclo *for* in una semplice formula scritta in un'unica riga di codice. Questa tecnica è particolarmente efficiente quando lo sviluppo della nuova lista dipende dai valori di altre liste o tuple, e lo è sia dal punto di vista temporale che spaziale, soprattutto a confronto con l'usuale utilizzo dei cicli *for*.

Nell'esempio sottostante viene mostrata precisamente la costruzione della riga di A sfruttando la List Comprehension. In questo caso risulterà più efficiente rispetto al ciclo *for* generico in quanto la riga i -esima di A viene costruita in base ai valori presenti in M e nell'insieme i -esimo di N .

```
A_row=[1 if (tupleM[j] in tupleN[i]) else 0 for j in range(lengthM)]
```

In seguito alla definizione della riga di A (ovvero *A_row*), tale riga viene stampata su una riga del file di testo nel formato corretto. Anche tutti i dettagli aggiuntivi vengono inseriti all'interno di *'input.txt'* fino all'esaurimento degli stessi, a cui segue la chiusura del file.

3. Scrittura del file *'indexes.txt'*

Il file *'indexes.txt'* è un file di utilità, aggiuntivo, ma di estrema rilevanza per la comprensione dell'uscita fornita dal file di output generato dai due algoritmi di Exact Cover. Esso viene generato solo in seguito alla costruzione del file *'input.txt'*: inizialmente viene fatta solamente una copia del contenuto del file di input, e subito dopo vengono inserite tutte le righe che rappresentano un'associazione tra l'indicatore e il rispettivo elemento all'interno dell'insieme a cui si riferisce. Vengono mostrate, una per riga, prima le coppie indice→elemento di M , e poi le coppie indice→sottoinsieme di N .

Per avere un'idea del tempo che l'applicazione impiega per svolgere tutte le operazioni di creazione di M , N e di stampa su file, viene sfruttato il modulo *timeit*. All'inizio del codice viene chiamata la funzione *default_timer()* per registrare l'istante di tempo di inizio dell'esecuzione del codice; nel momento in cui viene effettuata la stampa su file del tempo di conclusione di tutte le operazioni precedenti, viene richiamata nuovamente la funzione *default_timer()* ed effettuata la differenza tra l'istante questo nuovo istante e quello inizialmente salvato, ottenendo così il tempo di esecuzione del task. Questo valore è utile a comprendere le differenze in termini di tempo impiegato per costruire il file di input in base a valori diversi di M ed N .

Al termine della sezione “*Generazione del file di input*” del notebook sono inoltre presenti:

- la stampa diretta sul notebook dei dettagli principali sull'input costruito;
- il download di *'input.txt'* e *'indexes.txt'* sul dispositivo che si sta utilizzando.

Entrambi i file risulteranno comunque direttamente consultabili nell'area *File* di Google Colab.

6.4 Lettura del file di input

La parte del notebook a cui ci riferisce ora ha il titolo “*Lettura file di testo*”.

Il processo di lettura del file di testo di input, necessario per l'esecuzione di entrambi i programmi che risolvono il problema di Exact Cover, viene eseguito solamente se è presente, nella cartella del progetto (accessibile dall'area *File* in Colab), il file d'ingresso *input.txt*, che sia esso stato generato automaticamente dal programma al punto precedente o generato manualmente dall'utente.

Il file d'ingresso viene aperto in sola lettura, e da esso viene estratto tutto il contenuto della matrice A all'interno di una **lista** sfruttando nuovamente la *List Comprehension*. In particolare, dal file vengono estratte solamente le linee che non presentano né l'indicatore di commento “;”, né le linee vuote. Subito dopo, la lettura del file viene terminata.

Da qui in poi, il lavoro si sofferma sulla sistemazione del contenuto del file. Si vuole in particolare che ogni elemento della lista letta, rappresentato ancora come stringa alfanumerica, venga trasformato prima in una **lista** di caratteri, che vengano rimossi tutti i caratteri aggiuntivi (come il simbolo “-” che indica il termine di una riga di A , se presente) e che infine venga tramutato in una **tupla** per i medesimi motivi citati nella sezione [Generazione del file di input](#). Pertanto, al termine di questi passaggi, viene ottenuta una matrice A sotto forma di **tupla di tuple**, più veloce da consultare rispetto ad una lista di liste.

Per evitare che vengano importate matrici che non rispettano il formato dichiarato in [File di input](#), durante la fase di sistemazione del contenuto vengono effettuati dei controlli:

- Tramite una *List Comprehension* vengono inseriti, per ogni riga relativa ad A importata dal file, solo gli elementi della riga che contengono un 1 o uno 0; nel caso in cui uno degli elementi della lista non fosse un valore intero tra 1 o 0, viene stampato in Colab un messaggio di errore ad indicare tale problema, e la matrice non viene importata;

- Se due righe presentano lunghezza diversa (ad esempio, è stato creato un file di input con un diverso numero di elementi per riga, o alcuni valori non sono stati accettati nella sistemazione della riga di A), viene stampato in Colab un messaggio di errore relativo a tale problema, e la matrice non viene importata;
- Se viene rilevata una stringa di tutti zeri associata ad un insieme vuoto in N , il quale risulta essere un insieme che si vuole escludere dal problema, viene stampato in Colab un messaggio di errore relativo a tale problema, e la matrice non viene importata.

Se tutti i controlli vengono superati e la matrice correttamente importata, in Colab viene stampato direttamente il contenuto della matrice appena letta, in modo da verificare se vi sia corrispondenza tra quanto scritto nel file e quanto effettivamente importato.

6.5 Implementazione dell'algoritmo EC

Nel notebook è presente un'intera sezione che vuole rappresentare il programma implementativo dell'algoritmo base, dal titolo “**Implementazione algoritmo EC**”. All'interno di questa sezione vengono sviluppate a livello software entrambe le procedure *EC* ed *ESPLORA* presentate nella sezione [Algoritmo base: EC ed ESPLORA](#), e ci si occupa della scrittura dei risultati sul file di uscita.

Soffermandosi su quest'ultimo aspetto, si è optato per una scrittura diretta, sequenziale, del file di output, denominato ‘*output.txt*’. Esso viene infatti aperto in scrittura all'inizio del programma e mantenuto in tale stato fino al termine dell'esecuzione dello stesso programma. Questa scelta permette di evitare la costruzione di una variabile per *COV* (insieme delle partizioni di M trovate dall'algoritmo), risparmiando così memoria, e di evitare di rimandare la scrittura al termine dell'intera esecuzione. Ogni nuova partizione viene, pertanto, inserita direttamente nel file nell'esatto momento in cui viene rilevata dall'algoritmo.

Ovviamente, la scrittura tiene conto del formato del file di uscita specificato in [File di output](#).

Per permettere l'esecuzione dell'algoritmo principale, uno degli elementi essenziali, oltre alla matrice A di input precedentemente importata con la lettura del file d'ingresso, è la matrice di compatibilità B . La struttura dati utilizzata per il salvataggio di B è la **matrice sparsa LIL**. In genere, una matrice sparsa è utilizzata quando si ha a che fare con un numero elevato di zeri (indicativamente almeno il triplo rispetto al numero di elementi diversi da zero); per evitare un'elevata occupazione di memoria, tale struttura dati permette di memorizzare solamente gli elementi con valore diverso da 0 attraverso una tripletta (x, y, z) , dove x e y sono le coordinate (riga e colonna della matrice) e z è il valore non nullo corrispondente. La matrice B , essendo quadrata di $|N|$ righe, contiene un totale di $|N|^2$ elementi, ma gli elementi non nulli, ovvero quelli con valore 1, saranno sempre pochi rispetto alla totalità, soprattutto a causa delle tre condizioni a cui si deve sottostare affinché vi sia un 1 alla coordinata (x, y) ; ciò implica che il numero di zeri sarà sempre molto più elevato del numero di uni, e che quindi la matrice sparsa sia quella più adatta. L'utilizzo della matrice sparsa, però, va a scapito della velocità di esecuzione, in quanto l'accesso ai dati implica l'osservazione di tutte le triplette di elementi non nulli e, di conseguenza, una perdita dal punto di vista temporale.

In questo caso specifico, si è scelta una *lil_matrix*, resa disponibile dalla libreria *scipy.sparse*. Una matrice *LIL* (*List of Lists*) è una matrice sparsa con una struttura simile ad una lista di liste, tipicamente utilizzata per essere costruita in modo incrementale. La matrice B , infatti, viene costruita incrementalmente durante l'esecuzione dell'algoritmo, e ciò quindi ne giustifica la scelta. Inoltre, le matrici *LIL* sono di tipo *row-based*, ovvero presentano la lista delle triple ordinate rispetto all'indice della riga. L'unico confronto che viene eseguito tra due elementi di B è tra celle appartenenti alla stessa riga, e ciò rende ancora più utile la presenza di tale struttura dati in quanto i rispettivi valori verranno recuperati più velocemente.

Oltre alla variabile relativa alla matrice B , vengono inizializzate tutte delle variabili utili al fine della scrittura dei dettagli riassuntivi finali della soluzione del problema di Exact Cover. In particolare, si hanno:

- *visited*: variabile intera che rappresenta il numero di nodi effettivamente visitati durante l'esplorazione degli alberi, ovvero durante l'esecuzione del programma. Tale valore viene incrementato ogni qual volta viene esplorata una radice (rappresentante un insieme non vuoto e che sia esso stesso partizione o meno), un nodo relativo ad un aggregato di cardinalità 2 o (se presente) un nodo di un aggregato di cardinalità > 2 composto da insiemi tutti compatibili tra loro. Essendo utilizzata all'interno delle procedure *EC* ed *ESPLORA*, necessita di essere richiamata al loro interno come *variabile globale*;
- *tot_nodes*: variabile intera che rappresenta il numero totale dei nodi possibili, inteso come somma dei nodi (compresa la radice) di tutti gli alberi radicati in tutti gli i , dove $1 \leq i \leq |N|$ e $A[i]$ rappresenta l'insieme i -esimo di N . Viene calcolato con la formula specificata nella sezione [Principi e funzionamento dell'algoritmo](#);
- *COVcard*: variabile intera che rappresenta la cardinalità di *COV*, ovvero l'insieme delle partizioni di M risultanti. Non essendo salvati i valori di *COV* all'interno di una variabile specifica, ma salvati direttamente all'interno del file di uscita, è necessario incrementare il valore di *COVcard* in presenza di ogni nuova partizione trovata. Essendo utilizzata all'interno delle procedure *EC* ed *ESPLORA*, necessita di essere richiamata al loro interno come *variabile globale*;
- *COVcardDistrib*: lista ordinata in cui l'indice i di una cella rappresenta la cardinalità $i+1$ di una partizione di *COV*, mentre l'elemento nella posizione i è un numero intero che rappresenta il numero di partizioni di *COV* di cardinalità $i+1$. La cella di posizione 0 presenterà il numero di partizioni di *COV* di cardinalità 1, in quanto non possono esistere partizioni di cardinalità 0. Questa lista è utile a rappresentare la distribuzione di cardinalità delle partizioni di M . Tale distribuzione viene

costruita chiamando, ad ogni nuova partizione trovata, la funzione di utilità *cardDistribution*;

- *NcardSum*: variabile intera che rappresenta la somma della cardinalità di tutti i sottoinsiemi di N . Il suo valore viene aggiornato ogni volta in cui viene ispezionato un insieme $A[i]$, aggiungendo alla variabile la cardinalità di tale insieme. Attraverso questo valore è possibile prima calcolare la cardinalità media di tali insiemi su tutta la collezione ($avgN$) e poi l'*aggregate index* al termine dell'esecuzione. Essendo utilizzata all'interno delle procedure *EC*, necessita di essere richiamata al suo interno come *variabile globale*.

In ordine, sono presentate all'interno del notebook, in due blocchi in sequenza, prima la funzione *ESPLORA* e in seguito la funzione *EC*, in quanto la funzione *ESPLORA* è chiamata da *EC* e deve, pertanto, essere definita precedentemente. Entrambe le funzioni seguono passo per passo lo pseudocodice delle procedure fornite in [Algoritmo base: EC ed ESPLORA](#), pertanto in questa sezione ci si riferirà principalmente alle scelte effettuate dal punto di vista delle strutture dati utilizzate e, in caso, delle funzioni principali utilizzate e variabili aggiunte.

Il parametro di ingresso della funzione *EC* è la matrice A che, considerato quanto detto nel sottocapitolo precedente, è rappresentata attraverso una **tupla di tuple**. Per far sì che si possano aggiornare i valori delle variabili relative ai nodi visitati (*visited*) e alla cardinalità di COV (*COVcard*), esse vengono richiamate come variabili globali attraverso la keyword *global*.

All'interno dell'intero codice, per valutare il contenuto di una tupla o di una lista (generiche) dal punto di vista del numero di 1 e del numero di 0 viene utilizzato il metodo *count()* fornito dalle suddette strutture dati. In particolare, quando una riga $A[i]$ non ha alcuno 0 (e quindi $A[i].count(0)==0$), significa che è composta da tutti 1 e che di conseguenza l'insieme associato è una partizione; quando invece non ha alcun 1 (e quindi $A[i].count(1)==0$), sarà composta da tutti 0 e pertanto corrisponderà ad un insieme vuoto. Questa logica "inversa" è stata utilizzata per evitare la ripetitiva chiamata della funzione *len()* per riconoscere la lunghezza di $A[i]$ da confrontare con il valore in *return* dalla funzione

$A[i].count()$. Lo stesso discorso vale per la lista U , rappresentativa dell'unione di due insiemi di N in un aggregato di cardinalità 2.

In *Python*, la keyword *continue* viene utilizzata per passare direttamente all'iterazione successiva (se presente) quando ci si trova all'interno di un ciclo *for*. Pertanto, i *break* specificati nello pseudocodice sono implementati grazie a *continue*, e ciò accade quando l'insieme radice dell'albero rappresenta un insieme vuoto o un insieme coincidente con M . Al contrario, il *break* presente all'interno del codice implementato implica la terminazione del ciclo *for* in cui è inserito; infatti, esso viene utilizzato appena viene rilevata la presenza di elementi coincidenti, e quindi intersezione non vuota, tra due insiemi che di conseguenza non potranno formare un aggregato di cardinalità 2.

Per entrambe le variabili I e U presentate nello pseudocodice vengono utilizzate delle **liste**, in quanto vi è la possibilità che vengano utilizzate nuovamente all'interno del codice andando ad aggiungere nuovi elementi (ad esempio, all'interno della funzione *ESPLORA* quando si è in presenza di aggregati di cardinalità maggiore di 2). Per U viene utilizzato, piuttosto che un ciclo *for*, nuovamente una *List Comprehension* per velocizzare le operazioni di costruzione di una lista basandosi su altre liste (o, in questo caso, tuple) e sintetizzare il tutto su un'unica e semplice riga.

La variabile *Inter* presente nello pseudocodice viene invece suddivisa e implementata attraverso **due differenti liste**:

- *Inter_index*: una lista che viene utilizzata per inserire gli indici degli insiemi di N compatibili con l'aggregato (i,j) di ordine 2 che si sta analizzando in quel momento (i cui indici sono presenti in I); tali indici vengono inseriti all'interno del relativo ciclo *for* per r che va da 0 a j , dove r si riferisce quindi ad un insieme $A[r]$ che venga prima in ordine lessicografico rispetto ad $A[j]$;
- *Inter*: una lista in cui viene inserito un 1 se si ha la suddetta compatibilità, o uno 0 altrimenti.

Se la lista *Inter_index* non è vuota, ovvero è presente al suo interno almeno un indice relativo ad un insieme compatibile con l'aggregato definito da *U*, allora sarà possibile chiamare ed eseguire la funzione *ESPLORA*. I parametri in ingresso a tale funzione sono la matrice d'ingresso *A*, le liste *I* e *U*, e infine *Inter*, il quale è composto dalla coppia di liste “*Inter_index* - *Inter*” proveniente da *EC*, o dalla coppia “*Intertemp_index* - *Intertemp*” di *ESPLORA*. Anche questa funzione ricalca quanto specificato nello pseudocodice della *procedure ESPLORA* fornito.


Le variabili citate nello pseudocodice, ovvero *Itemp* ed *Utemp*, sono anche in questo caso rappresentate attraverso delle **liste**, per i medesimi motivi di *I* e *U* in *EC*. *Itemp* viene costruito sulla base della lista di ingresso *I*, e ad esso viene aggiunto tramite *append()* un indice *k* che rappresenta uno degli insiemi compatibili con l'aggregato definito da *I*. *Utemp* viene invece costruito tramite *List Comprehension* basandosi su confronti tra *U* e la riga *k*-esima di *A*.

Anche *Intertemp* viene separato in **due differenti liste**:

- *Intertemp_index*: come per *Inter_index* in *EC*, questa lista contiene gli indici degli elementi compatibili con l'aggregato rappresentato da *Itemp*, se esistono. Gli insiemi analizzati sono gli $A[K]$ per valori *K* che vanno da 0 a *k*, ovvero tutti gli insiemi precedenti per ordine lessicografico all'insieme $A[k]$;
- *Intertemp*: come per *Inter* in *EC*, questa lista presenta un 1 quando si ha la compatibilità suddetta, o uno 0 altrimenti.

Anche in questo caso, se *Intertemp_index* non è vuoto, e quindi esistono insiemi compatibili agli insiemi di *Itemp*, allora è possibile effettuare la chiamata ricorsiva di *ESPLORA* con i nuovi parametri *Itemp*, *Utemp*, *Intertemp_index* e *Intertemp*, oltre alla matrice *A*.

Al termine della sezione [File di output](#) sono già stati introdotti i concetti di *blocco da parte dell'utente*, *blocco per superamento della durata massima di esecuzione* e il problema del *numero di nodi totali possibili troppo elevato*. Innanzitutto, viene

utilizzato il modulo *signal* per la gestione di entrambe le interruzioni: per il blocco automatico viene impostato un timeout dopo *3600 secondi (1 ora)* di esecuzione dell'algoritmo ancora non conclusa, mentre per il blocco manuale è bastevole premere il pulsante  per far sì che si fermi.

Eseguendo il blocco contenente la chiamata principale alla funzione *EC*, oltre alla registrazione dell'istante di inizio (come accadeva per il programma di creazione del file di input), inizia la vera e propria esecuzione dell'algoritmo base. Il parametro d'ingresso è la matrice *A* precedentemente acquisita grazie al programma di lettura del file. L'esecuzione delle operazioni di *EC* (e di *ESPLORA* al suo interno, se necessarie) prosegue fino al suo normale termine (ovvero, con la visita dell'ultimo nodo visitabile dell'albero radicato nell'ultimo elemento di *N*), a meno che non vengano rilevati dal metodo *signal.signal* un *TimeoutError* (raggiungimento del timeout fissato) o un *KeyboardInterrupt* (blocco da parte dell'utente). I messaggi relativi vengono opportunamente stampati all'interno del file, ad avvisare l'utente della presenza di uno stop anomalo. In ogni caso, la scrittura prosegue con l'inserimento di tutte le informazioni utili e riassuntive del comportamento dell'algoritmo, come specificato in [File di output](#).

Una tra queste informazioni di maggiore rilevanza da inserire nel file è l'*aggregate index*. Il suo valore viene calcolato dividendo la cardinalità media degli insiemi di *N* (*avgN*) per la cardinalità del dominio *M*. La cardinalità media, in particolare, è ottenuta sfruttando il contenuto di *NcardSum* (che contiene la somma della cardinalità di tutti gli insiemi di *N* ispezionati nell'esecuzione) e dividendolo per la cardinalità di *N*. Così facendo si ottiene un numero che va a riassumere, in un unico valore compreso tra 0 e 1, la distribuzione di cardinalità di *N* utile per permettere, in seguito, la valutazione degli andamenti dell'algoritmo.

Ai fini del confronto tra i risultati ottenuti dall'algoritmo base e dall'algoritmo plus, vengono salvati all'interno di una struttura dati **dizionario** (chiamata *EC_data_for_comparison*) tutti i valori delle variabili maggiormente utili per

verificare l'andamento dei due algoritmi sullo stesso input. Questi valori sono, in particolare:

- *ended*: variabile usata come *flag* per comprendere se è stata portata a termine correttamente l'esecuzione di tutto il programma (*ended=True*) oppure se ci sono stati degli stop utente o per timeout raggiunto (*ended=False*);
- *la cardinalità del dominio M*;
- *la cardinalità della collezione N*;
- *l'aggregate index*;
- *visited*: il numero di nodi visitati;
- *COVcard*: la cardinalità di COV;
- *COVdistribution*: la distribuzione di cardinalità degli insiemi di COV;
- *execution_time*: il tempo di esecuzione dell'algoritmo.

Al termine della sezione “*Implementazione algoritmo EC*” del notebook è inoltre presente il download di ‘*output.txt*’ sul dispositivo che si sta utilizzando. Il file risulterà comunque direttamente consultabile nell’area *File* di Google Colab.

6.6 Implementazione dell'algoritmo EC^+

Nel notebook è presente un'intera sezione che si riferisce al programma implementativo dell'algoritmo plus, dal titolo “***Implementazione algoritmo EC^+*** ”. All'interno di questa sezione vengono sviluppate a livello software entrambe le procedure EC^+ ed $ESPLORA^+$ presentate nella sezione [Algoritmo plus: \$EC^+\$ ed \$ESPLORA^+\$](#) , e ci si occupa della scrittura dei risultati sul file di uscita.

Anche in questo caso, come per la versione base dell'algoritmo, si è optato per una scrittura sequenziale del file di output, stavolta denominato ‘*output2.txt*’. Esso viene infatti aperto in scrittura all'inizio del programma e mantenuto in tale stato fino al termine dell'esecuzione dello stesso programma. La scelta risulta ancora la stessa, ovvero evitare la costruzione di una variabile per COV,

risparmiando così memoria, ed evitare di rimandare la scrittura al termine dell'intera esecuzione. Ogni nuova partizione viene quindi inserita direttamente nel file nell'esatto momento in cui viene rilevata dall'algoritmo. La scrittura tiene conto del formato del file di uscita specificato in [File di output](#).

Le funzioni principali implementate sono denominate *EC_PLUS* ed *ESPLORA_PLUS*. Dal punto di vista dei parametri in ingresso, solo *ESPLORA_PLUS* cambia, sostituendo la variabile *cardU* alla lista *U*.

Le variabili presenti in questa versione dell'algoritmo per gran parte del codice sono le medesime. Per evitare inutili ripetizioni, ci si soffermerà ora sugli aspetti che distinguono le due versioni e sulle variabili che sono state aggiunte o, al contrario, rimosse.

Come richiesto dai requisiti funzionali, in questa versione vi è la necessità di avere in output, oltre a quanto già dichiarato, la distribuzione delle cardinalità degli insiemi della collezione N , esattamente come accade quando viene generato il file di input. Tale distribuzione è salvata all'interno della **lista** *NcardDistrib*, che viene costruita durante l'esecuzione della funzione *EC+* ad ogni iterazione del ciclo principale, in quanto attraverso tale ciclo vengono osservati uno alla volta tutti i sottoinsiemi di N (corrispondenti alle righe di A). Anche qui, l'indice i di un elemento rappresenta la cardinalità $i+1$, mentre l'elemento nella posizione i rappresenta il numero di insiemi di N aventi cardinalità $i+1$. Anche questa distribuzione viene costruita sfruttando la funzione di utilità *cardDistribution()*.

L'algoritmo plus si appoggia principalmente sul fatto di voler evitare di effettuare l'unione di insiemi, il cui risultato viene inserito, nell'algoritmo base, nella lista U di *EC* per aggregati di cardinalità 2 e nella lista *Utemp* di *ESPLORA* per aggregati di cardinalità > 2 . Piuttosto che utilizzare le liste, si sfrutta la cardinalità degli insiemi: infatti, basta salvare il numero di 1 presenti in ciascuna riga di A e, in caso fosse necessaria l'unione, effettuare la somma della cardinalità degli insiemi che si vogliono unire. Se la cardinalità relativa ad un aggregato dato dall'unione di due o più insiemi di N è esattamente uguale ad $|M|$, allora si avrà a che fare con una nuova partizione.

Si passa perciò dall'utilizzare delle liste, più dispendiose sia dal punto di vista dello spazio di memoria occupato che dal punto di vista del tempo per la loro costruzione, all'utilizzo di tre semplici **variabili intere** su cui si effettuano solamente operazioni di somma:

- *cardi*: variabile di *EC_PLUS* che rappresenta la cardinalità dell'insieme relativo alla radice *i* dell'albero, ovvero il numero di 1 presenti in $A[i]$. Viene inizializzato all'inizio del ciclo primario dell'algoritmo EC^+ , in modo che venga utilizzato sia per verificare se l'insieme radice sia vuoto o già coincidente con M , sia per effettuare le somme con le cardinalità dei possibili aggregati;
- *cardU*: variabile di *EC_PLUS* che rappresenta la cardinalità dell'unione tra due insiemi componenti un aggregato di cardinalità 2, i cui indici identificatori sono presenti in I come accade per il caso base. Viene calcolato semplicemente come somma di *cardi* e della cardinalità dell'insieme associato al secondo indice di I , ovvero uno degli insiemi compatibili con $A[i]$ e precedenti allo stesso per ordine lessicografico; se *cardU* equivale ad $|M|$, si ha a che fare con una partizione $\{I\}$.
- *cardTemp*: variabile di *ESPLORA_PLUS* che rappresenta la cardinalità dell'unione tra due insiemi componenti un aggregato di cardinalità > 2 , i cui indici identificatori sono presenti in $Itemp$ come accade per il caso base. Viene calcolato semplicemente come somma di *cardU* e della cardinalità dell'insieme associato a k , uno degli indici di $Inter$, ovvero uno degli insiemi compatibili con gli insiemi associati all'aggregato di I in ingresso; se *cardtemp* equivale ad $|M|$, si ha a che fare con una partizione $\{Itemp\}$.

Per il resto, il codice rimane esattamente il medesimo, ad eccezione di piccole variazioni durante la stampa sul file di uscita per identificare l'algoritmo che si sta utilizzando (consultare [File di output](#) per maggiori dettagli).

Anche qui viene utilizzato un **dizionario** per il salvataggio delle variabili principalmente utili al confronto tra gli algoritmi base e plus, dal nome *ECplus_data_for_comparison*. La struttura è la medesima del dizionario usato

nell'algoritmo base, tuttavia presenterà, ovviamente, i valori recuperati dall'esecuzione di *EC_PLUS* ed *ESPLORA_PLUS*.

Al termine della sezione “*Implementazione algoritmo EC+*” del notebook è inoltre presente il download di ‘*output2.txt*’ sul dispositivo che si sta utilizzando (ad esempio, un pc). Il file risulterà comunque direttamente consultabile nell’area *File* di Google Colab.

7. Prove sperimentali

L'obiettivo di questo capitolo è quello di mostrare le scelte implementative e i risultati che si sono ottenuti dal punto di vista della *sperimentazione*. Per far ciò, è necessario sottoporre lo stesso input ai due risolutori del problema di Exact Cover, rispettivamente coloro che implementano l'algoritmo base e l'algoritmo plus; tale input è recuperato automaticamente da un unico file di input. I risultati che si ottengono, sia con la somministrazione ai due algoritmi di un unico file di input che nel caso di input multipli, possono essere confrontati al fine di verificare e valutare vari aspetti, quali la possibile presenza di errori, le prestazioni sia dal punto di vista temporale che da quello spaziale e la variazione dei risultati al variare dei parametri in ingresso (N ed M).

Il lavoro è suddiviso in *due sezioni sperimentali*:

1. **Confronto diretto** (sperimentazione singola): questa sezione mira principalmente alla verifica della presenza di errori (logici o di programmazione) all'interno del codice di uno o di entrambi i risolutori implementati, oltre che ad una prima valutazione delle prestazioni rispetto ad un singolo input;
2. **Sperimentazione multipla**: questa seconda parte mira ad evidenziare graficamente l'andamento di alcuni dei risultati di maggiore rilevanza ottenibili eseguendo più istanze del problema, soffermandosi in particolar modo su quelli dal punto di vista temporale al variare dei parametri d'ingresso, ovvero il dominio M , la collezione N e il valore delle rispettive cardinalità. Essa stessa implementa, quindi, molteplici casi al suo interno con i relativi confronti diretti, al fine di avere un'idea completa di tutti gli elementi su cui la sperimentazione vuole indagare.

7.1 Confronto diretto

Ci si vuole ora soffermare sulla parte di codice relativa alla sezione “**Confronto diretto**” presente all’interno del notebook Colab. Si è scelto di implementare, oltre alla sperimentazione di cui si tratterà nel prossimo sottocapitolo, anche un programma che si occupi di produrre una *sperimentazione singola*. Nella pratica, si tratta di un semplice *confronto tra i risultati che si ottengono dall’esecuzione di entrambi gli algoritmi (base e plus) relativamente ad uno stesso input* (recuperato, a sua volta, dal medesimo file d’ingresso). Il fine è quello di recuperare informazioni utili a verificare la presenza di eventuali discrepanze tra i valori in uscita, oppure direttamente tra i file di output generati, e a valutare il comportamento dei due algoritmi rispetto al medesimo input.

Il confronto diretto viene effettuato seguendo due modalità:

1. *Confronto per valore*: in questo primo caso, il confronto viene effettuato utilizzando i valori delle principali variabili impiegate durante l’esecuzione dei due algoritmi. Tali valori sono salvati all’interno dei **due dizionari** forniti dopo la stampa sui rispettivi file di output; in particolare, ciascun dizionario è costituito da coppie *key-value*, in cui la chiave (*key*) è in formato stringa e identifica (come se fosse un titolo) il significato del contenuto di *value*. Nella sezione [Implementazione dell’algoritmo EC](#) è specificato con maggiore dettaglio l’effettivo contenuto dei dizionari.

Sfruttando il formato delle stringhe in Python, è stato possibile costruire una tabella in grado di racchiudere i valori dei dizionari e di confrontarne il contenuto. Essa è composta da *quattro colonne*:

- *variable*: l’elemento variabile su cui si sta effettuando il confronto, rappresentato dal valore di *key* nei dizionari;
- *EC*: rappresenta il valore (*value*) relativo a *variable* rispetto all’algoritmo base, recuperato dal relativo dizionario;
- *EC+*: rappresenta il valore (*value*) relativo a *variable* rispetto all’algoritmo plus, recuperato dal relativo dizionario;

- *Different?*: questa colonna vuole rendere evidente la differenza tra i valori relativi a *variable* rispetto ai due algoritmi. In particolare, in presenza di valori uguali viene scritto “yes”, mentre per valori differenti viene stampato “no”. Per le variabili di tipo numerico (*int* e *float*), oltre a stampare la stringa “no”, viene scritto tra parentesi quadre il risultato dell’operazione di differenza tra i due. Il valore assoluto della differenza rappresenta quanto effettivamente si discostano tra loro i valori di una certa variabile, mentre il segno sta ad indicare se il risultato di *EC* è minore del risultato di *EC+* (segno “-”) o viceversa (segno “+”).

Sulle righe della stessa tabella sono quindi inseriti, uno sotto l’altro e rispettando la suddivisione delle colonne, i valori recuperati dai due dizionari e i risultati dei confronti e delle differenze effettuate.

Per avere un’idea anche della differenza dal punto di vista della distribuzione di cardinalità, vengono stampate le distribuzioni costruite da entrambi gli algoritmi: se uguali, viene stampato un messaggio che ne riferisce l’uguaglianza, altrimenti stampa un messaggio ad indicare tale discrepanza e i rispettivi valori calcolati attraverso l’operazione di differenza.

La tabella viene stampata direttamente dopo l’esecuzione del blocco relativo, ma nel contempo viene scritta all’interno di un file di testo “*comparison.txt*”, in modo da avere automaticamente una traccia salvata del confronto effettuato. Tale file di testo è reperibile direttamente dalla sezione *File*, oltre ad essere scaricato automaticamente sul dispositivo che si sta utilizzando al termine dell’esecuzione dello stesso blocco.

A titolo esemplificativo, nell’immagine seguente viene mostrata la struttura di una tabella generata da un caso di confronto (recuperata direttamente dal file “*comparison.txt*” scritto).

1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	5	5	no
5 N cardinality	12	12	no
6 Aggregate index	0.4333	0.4333	no
7 visited nodes	73	73	no
8 COV cardinality	5	5	no
9 execution time	0.000717 s	0.000608 s	yes [0.000109 s]
10			
11 COV distribution for EC: [0, 1, 4]			
12 COV distribution for EC+: [0, 1, 4]			
13 COV distributions are the same.			

Il relativo caso viene reso disponibile alla consultazione [qui](#).

2. *Confronto tra file*: il secondo blocco implementato è quello del confronto tra file. I file di output degli algoritmi base e plus, al di là dei casi delle differenze citate in [File di output](#), dovrebbero essere identici, sia dal punto di vista delle variabili riassuntive elaborate dagli algoritmi, sia dal punto di vista della lista delle partizioni trovate. Pertanto, per controllare che ciò sia vero, vengono aperti entrambi i file di output in lettura e viene eseguito un ciclo che si occupa iterare tra le righe e di verificare che, per medesime righe, vi siano medesimi valori. Se ciò non avviene, viene stampato su colab il numero della riga del file in cui è presente un'incongruenza, e il relativo contenuto per entrambi i file.

Il problema principale di questo tipo di controllo è che utilizzare il ciclo e il confronto con la lettura di due file potrebbe essere dispendioso dal punto di vista temporale, soprattutto in presenza di file di dimensione particolarmente elevata.

Nell'immagine seguente è presente il risultato della stampa del confronto tra file in Colab, riferita al medesimo esempio del confronto tra valori al punto precedente.

```
Line 11:
Output EC: ;;; Time to create partitions: 0.000717753999997786s
Output EC+: ;;; Time to create partisions: 0.0006085190000090279s
```

Quanto scritto sta ad indicare che l'unica riga che presenta una differenza tra i due file è la numero 11, ovvero quella che si riferisce al tempo di esecuzione dei due algoritmi. Ciò è giustificato dal fatto che le due versioni dell'algoritmo implementate operano in modi e tempi differenti.

7.1.1 Esempi di confronto diretto per valore

Prima di passare alla sperimentazione multipla, vengono svolti una serie di casi singoli, tutti differenti tra loro, al fine di verificare il funzionamento degli algoritmi, effettuare un confronto diretto per valore tra i risultati degli stessi e, in caso, individuare per quali valori di input si ottengono risultati incompleti, esecuzioni troppo lunghe o errori. Le tabelle di confronto ottenute sono presentate direttamente in seguito, mentre gli esempi completi sono disponibili [qui](#). Ciascun esempio è contenuto in una singola cartella, il cui nome è il medesimo dei titoli che si possono visionare al di sopra delle singole tabelle riportate in seguito.

$M = 5$

1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	5	5	no
5 N cardinality	31	31	no
6 Aggregate index	0.5161	0.5161	no
7 visited nodes	523	523	no
8 COV cardinality	52	52	no
9 execution time	0.009057 s	0.005086 s	yes [0.003971 s]
10			
11 COV distribution for EC: [1, 15, 25, 10, 1]			
12 COV distribution for EC+: [1, 15, 25, 10, 1]			
13 COV distributions are the same.			

$M = 10 (1)$

1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	10	10	no
5 N cardinality	854	854	no
6 Aggregate index	0.5101	0.5101	no
7 visited nodes	877820	877820	no
8 COV cardinality	84607	84607	no
9 execution time	34.874839 s	33.817124 s	yes [1.057715 s]
10			
11 COV distribution for EC:	[1, 357, 5541, 21780, 31438, 19266, 5445, 733, 45, 1]		
12 COV distribution for EC+:	[1, 357, 5541, 21780, 31438, 19266, 5445, 733, 45, 1]		
13 COV distributions are the same.			

$M = 10 (2)$

1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	10	10	no
5 N cardinality	854	854	no
6 Aggregate index	0.5258	0.5258	no
7 visited nodes	711622	711622	no
8 COV cardinality	52789	52789	no
9 execution time	26.827454 s	26.008834 s	yes [0.81862 s]
10			
11 COV distribution for EC:	[1, 343, 4051, 13252, 18299, 12260, 3941, 600, 41, 1]		
12 COV distribution for EC+:	[1, 343, 4051, 13252, 18299, 12260, 3941, 600, 41, 1]		
13 COV distributions are the same.			

$M = 15 (1)$

1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	15	15	no
5 N cardinality	1903	1903	no
6 Aggregate index	0.5862	0.5862	no
7 visited nodes	68823763	68823763	no
8 COV cardinality	2346274	2346274	no
9 execution time	3577.756464 s	3387.424667 s	yes [190.331797 s]
10			
11 COV distribution for EC:	[1, 129, 1580, 10180, 47947, 183250, 470242, 693179, 575278, 274316, 76511, 12461, 1145, 54, 1]		
12 COV distribution for EC+:	[1, 129, 1580, 10180, 47947, 183250, 470242, 693179, 575278, 274316, 76511, 12461, 1145, 54, 1]		
13 COV distributions are the same.			

$M = 15$ (2)

comparison.txt ×			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	no	no	no
4 M cardinality	15	15	no
5 N cardinality	9202	9202	no
6 Aggregate index	0.1671	0.1706	yes [-0.0035]
7 visited nodes	71213763	72513782	yes [-1300019]
8 COV cardinality	1640407	1697158	yes [-56751]
9 execution time	3599.999803 s	3599.999763 s	yes [4e-05 s]
10			
11 COV distribution for EC:	[0, 212, 8017, 112693, 466998, 639711, 335863, 71209, 5593, 111]		
12 COV distribution for EC+:	[0, 222, 8443, 118572, 486514, 660984, 344302, 72379, 5631, 111]		
13 COV distributions are different.			
14 Difference:	[0, -10, -426, -5879, -19516, -21273, -8439, -1170, -38, 0]		

$M = 20$

1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	no	no	no
4 M cardinality	20	20	no
5 N cardinality	8320	8320	no
6 Aggregate index	0.246	0.2466	yes [-0.0006]
7 visited nodes	78982615	80100519	yes [-1117904]
8 COV cardinality	68591	71055	yes [-2464]
9 execution time	3599.999943 s	3599.999924 s	yes [1.8e-05 s]
10			
11 COV distribution for EC:	[0, 36, 171, 577, 1543, 3550, 6892, 13516, 18451, 15090, 6931, 1665, 167, 2]		
12 COV distribution for EC+:	[0, 37, 174, 588, 1593, 3648, 7105, 14004, 19158, 15666, 7190, 1721, 169, 2]		
13 COV distributions are different.			
14 Difference:	[0, -1, -3, -11, -50, -98, -213, -488, -707, -576, -259, -56, -2, 0]		

Si può osservare da questi primi esempi come la struttura tabellare permetta di visualizzare in modo semplice e intuitivo le principali differenze tra le esecuzioni delle due versioni dell'algoritmo implementate. Inoltre, la lettura del file “*comparison.txt*” permette di accedere ancora più facilmente ai risultati di maggiore rilevanza per il confronto, rendendo non necessaria l’apertura dei file di input o di output a tal fine, soprattutto quando questi ultimi hanno dimensioni particolarmente grandi (per cui bisognerebbe scorrere fino al termine della lista delle partizioni trovate per ottenere le informazioni volute).

È già possibile notare alcuni aspetti all’interno dei test effettuati. In particolar modo, si osserva una crescita dei tempi di esecuzione all’aumentare del numero di insiemi nel dominio M e nella collezione N , del numero di nodi visitati, della cardinalità di COV . A questi si aggiungono la variabilità dei valori di *aggregate*

index, la distribuzione di cardinalità delle partizioni presenti in *COV* e, infine, la versione dell'algoritmo utilizzato.

Si osserva anche che per valori di $|M| \geq 15$ e per valori di $|N|$ con ordine di grandezza ≥ 3 difficilmente si riescono ad ottenere esecuzioni che non superino il timeout massimo (impostato, per scelta, a un'ora). Tuttavia, anche in questi casi non risultano solo essere tali valori a portare al raggiungimento di una soluzione non ottima (ovvero ad un insieme non completo delle partizioni possibili), ma anche, ad esempio, la cardinalità degli insiemi contenuti in N e l'implementazione stessa dell'algoritmo.

Per entrare maggiormente nel merito di tutti questi fattori è presentata la sperimentazione multipla nel capitolo a seguire; essa, attraverso i grafici che si ottengono in uscita, permette di dare maggiore risalto a ciascuna delle osservazioni appena fatte e di fornire risposte in modo più intuitivo e dettagliato.

7.2 Sperimentazione multipla

L'ultima sezione all'interno del notebook Colab, a cui ci si riferirà d'ora in avanti, è intitolata “***Sperimentazione***”. Nonostante la parte di sperimentazione inizi di fatto con il confronto diretto, si è scelto di dare tale nome a questa sezione del lavoro in quanto essa stessa racchiude una serie di casi su cui vengono fatti tutti i confronti possibili. Il resoconto finale, disponibile in seguito all'interno di questo elaborato, fornisce tutti i risultati al fine della valutazione degli aspetti che risultano rendere evidente la differenza tra il comportamento dell'algoritmo base e il comportamento dell'algoritmo plus. Ad essere valutati, nello specifico, sono:

- i tempi di esecuzione di entrambi gli algoritmi rispetto ad input differenti;
- tutti gli altri risultati forniti dalle uscite degli algoritmi, rappresentate e messe a confronto attraverso il “confronto diretto”, rispetto ad input differenti.

Il blocco “*Sperimentazione*” è suddiviso in ulteriori sottocapitoli, di cui si discuterà ora sia dal punto di vista delle scelte implementative che dei risultati ottenuti. **Per il resto del capitolo ci si riferirà a due casi di sperimentazioni multipla, entrambi reperibili [qui](#).** Essi sono presi come esempio per permettere di mettere in luce le principali differenze tra gli algoritmi al variare di alcuni parametri, e si suddividono in [Caso 1](#) e [Caso 2](#).

Ciò non toglie che eseguendo il blocco “*Sperimentazione*” sia possibile realizzare anche nuove sperimentazioni e valutare automaticamente, grazie ai risultati ottenuti e reperibili direttamente in Colab, l’andamento su nuovi casi sperimentali.

7.2.1 Confronto multiplo

Questa sezione si riferisce al capitolo introduttivo alla sperimentazione multipla presente nel notebook. Si è scelto di eseguire un totale di **6 singole sperimentazioni**, le quali differiscono l’una dall’altra per il valore di cardinalità del dominio M . Nello specifico, si sono scelte cardinalità sempre di più crescenti, inserite all’interno di una lista $MlenList$. **Le scelte appena riportate si riferiscono ad entrambi gli esempi di sperimentazione che sono stati analizzati e riportati in questo elaborato**, ma ciò non toglie che per sperimentazioni nuove si possano scegliere sia un numero differente di sperimentazioni, sia valori di $|M|$ differenti.

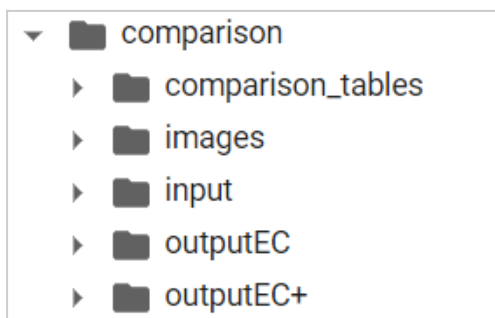
Vengono, pertanto, inizializzate tutte le variabili che permetteranno effettuare la valutazione al termine di tutti i confronti. In particolare, ognuna di esse assume la struttura di una **lista**, in modo che durante il codice, quando necessario, venga aggiunto un nuovo risultato in coda alla stessa. Ciò implica, di conseguenza, che ad ogni cella di indice i della lista corrisponde un valore che si riferisce all’ i -esima sperimentazione.

Tali liste sono:

- *MlenList*: la lista ordinata relativa alle diverse cardinalità di M su cui si vogliono effettuare gli esperimenti. Si è scelto di valutare rispetto ai seguenti valori: [4, 5, 6, 8, 10, 12];
- *NlenList*: lista relativa alle diverse cardinalità di N costruite sulla base della cardinalità di M tramite il programma di costruzione del file di input;
- *aggregate_indexes*: lista relativa agli *aggregate index* di ciascun esperimento effettuato;
- *COVlenList_EC*: lista relativa alle diverse cardinalità di COV ottenute al termine dell'esecuzione dell'algoritmo base al variare dati in ingresso;
- *COVlenList_ECplus*: lista relativa alle diverse cardinalità di COV ottenute al termine dell'esecuzione dell'algoritmo plus al variare dati in ingresso;
- *execution_time_table_EC*: lista relativa ai diversi tempi di esecuzione dell'algoritmo base al variare dei dati in ingresso;
- *execution_time_table_ECplus*: lista relativa ai diversi tempi di esecuzione dell'algoritmo plus al variare dei dati in ingresso.

7.2.2 Creazione delle cartelle

Questo passaggio permette la creazione delle cartelle in cui vengono inseriti tutti i risultati ottenuti durante l'esecuzione della sperimentazione multipla. La struttura è la seguente:



La cartella **comparison** è quella principale e si occupa di racchiudere tutto il materiale utilizzato nella sperimentazione nelle seguenti sottocartelle:

- **input**: contiene tutti i file di input e i file di utilità “*indexes*” generati durante la sperimentazione. Per essere distinti gli uni dagli altri, i file di input hanno il nome “*input_Mi.txt*”, mentre per quelli di utilità si ha “*indexes_Mi.txt*”, dove *i* è l'indice associato all'(*i*+1)-esima sperimentazione singola effettuata. Ad esempio, “*input_M0.txt*” rappresenta l'input della prima sperimentazione effettuata, in cui $|M|=MlenList[0]=4$;
- **outputEC**: contiene tutti i file di output di *EC* generati durante la sperimentazione. Per essere distinti gli uni dagli altri, i file di output hanno il nome “*output_Mi.txt*”, dove # assume lo stesso significato di quanto detto per l'input.
- **outputEC+**: contiene tutti i file di output di *EC+* generati durante la sperimentazione. Per essere distinti gli uni dagli altri, i file di output hanno il nome “*output_Mi_plus.txt*”, dove # assume lo stesso significato di quanto detto per l'input.
- **comparison_tables**: contiene tutti i file del confronto diretto tra *EC* ed *EC+* generati durante la sperimentazione. Per essere distinti gli uni dagli altri, i file di output hanno il nome “*comparison_Mi.txt*”, dove # assume lo stesso significato di quanto detto per l'input.
- **images**: contiene tutte le immagini relative ai grafici ottenuti al termine della sperimentazione. In base al significato del grafico, i file “.png” al suo interno sono nominati in modo univoco e riconoscibile, come specificato nella sezione [Visualizzazione degli andamenti](#) a seguire.

7.2.3 Creazione del file di input

La parte relativa alla creazione dell'input ricalca esattamente quanto svolto e spiegato in [Generazione del file di input](#). Tale codice viene eseguito, piuttosto che una singola volta, un totale di 6 volte, ovvero una per ogni sperimentazione che si vuole effettuare, sfruttando un ciclo *for*. Seguendo l'ordine delle cardinalità

presenti nella lista *MlenList*, si va a costruire, ad ogni iterazione *i* del ciclo, l'*i*-esimo file di input ("*input_Mi.txt*") sulla base dell'*i*-esimo valore di *MlenList*, e in seguito il rispettivo file di utilità "*indexes_Mi.txt*". La variabile associata alla cardinalità di *M* (*lengthM*) otterrà quindi questo valore da *MlenList*, e da lì in poi opererà seguendo il normale algoritmo di generazione dell'input.

Fissando tutte le cardinalità di *M*, viene tolto il carattere di pseudo-casualità che si aveva nel codice originario. Mantenere tale variabilità nel calcolo della cardinalità della collezione *N* provocherebbe casi sempre diversi e talvolta poco consoni per avere un'idea del comportamento degli algoritmi, pertanto si è optato di togliere la pseudo-casualità anche in questo caso, fissando $|N|$ alla metà di *max_comb*, ovvero la variabile che, nel codice per la singola generazione, rappresenta il numero totale di tutte le combinazioni semplici di classe compresa tra 1 e $|M|$. Ad ogni iterazione viene quindi aggiunto un valore di cardinalità di $|N|$ ad *NlenList*; la lista finale, per i casi di esempio scelti, presenterà quindi i seguenti valori: [7, 15, 31, 127, 511, 2047].

Tutti i file prodotti da questo blocco di codice vengono direttamente inseriti e raccolti nella sottocartella *input* di *comparison*. **Per i casi di esempio, tali file sono disponibile nella cartella *input* delle due sperimentazioni multiple accessibili da [qui](#).**

Durante la fase di creazione dei file d'ingresso viene inoltre aggiornata la lista *aggregate_indexes* andando ad aggiungere alla stessa, ad ogni iterazione del ciclo *for*, il valore dell'*aggregate index* calcolato in seguito alla creazione della relativa collezione *N*. Tale lista sarà essenziale, al termine della sperimentazione multipla, per la comprensione dell'andamento dei risolutori.

7.2.4 Esecuzione degli algoritmi

Questo blocco di codice racchiude in un unico ciclo *for* tutte le operazioni che si occupano dell'esecuzione completa degli algoritmi base e plus sulla base dei file di input contenuti all'interno della cartella *input* di *comparison*. Il ciclo viene eseguito 6 volte, ed ogni iterazione *i* contiene i seguenti passi:

1. *Lettura del file di input*: all'inizio dell'iterazione viene eseguita, un'unica volta, la lettura del file "*input_Mi.txt*" con i relativi controlli, esattamente come specificato in [Lettura del file di input](#);
2. *Esecuzione di EC*: il primo algoritmo ad essere eseguito è l'algoritmo base, attraverso la chiamata della funzione *EC*. Perché l'esecuzione vada a buon fine, è **necessario** che prima siano stati eseguiti i blocchi in cui sono presenti le funzioni *EC* ed *ESPLORA* nella sezione "*Implementazione algoritmo EC*" del notebook. Anche in questo caso, le operazioni sono le medesime della sezione appena citata, i cui dettagli implementativi sono presentati in [Implementazione dell'algoritmo EC](#). A queste ultime, vanno aggiunte le operazioni di aggiornamento delle liste relative all'algoritmo base: *COVlenList_EC*, *execution_time_table_EC*.

Il file "*output_Mi.txt*" viene generato dal codice e salvato direttamente nella sottocartella *outputEC* di *comparison*. **Per i casi di esempio, tali file sono disponibili nella cartella *outputEC* delle due sperimentazioni multiple accessibili da [qui](#).**

3. *Esecuzione di EC⁺*: il secondo algoritmo ad essere eseguito è l'algoritmo plus, attraverso la chiamata della funzione *EC_PLUS*. Perché l'esecuzione vada a buon fine, è **necessario** che prima siano stati eseguiti i blocchi in cui sono presenti le funzioni *EC_PLUS* ed *ESPLORA_PLUS* nella sezione "*Implementazione algoritmo EC⁺*" del notebook. Anche in questo caso, le operazioni sono le medesime della sezione appena citata, i cui dettagli implementativi sono presentati in [Implementazione dell'algoritmo EC[±]](#). A queste ultime, vanno aggiunte le operazioni di aggiornamento delle liste

relative all'algoritmo plus, ovvero le variabili *COVlenList_ECplus* ed *execution_time_table_ECplus*.

Il file “*output_Mi_plus.txt*” viene generato dal codice e salvato direttamente nella sottocartella *outputEC+* di *comparison*. **Per i casi di esempio, tali file sono disponibili nella cartella *outputEC+* delle due sperimentazioni multiple accessibili da [qui](#).**

4. *Stampa del confronto diretto*: al termine del ciclo viene effettuata la stampa su file della tabella di confronto tra i risultati delle soluzioni fornite dai due algoritmi implementati ai punti precedenti, con le istruzioni e il formato specificato al punto *Confronto per valore* di [Confronto diretto](#).

Il file “*comparison_Mi.txt*” viene generato dal codice e salvato direttamente nella sottocartella *comparison_tables* di *comparison*. **Per i casi di esempio, tali file sono disponibili nella cartella *comparison_tables* delle due sperimentazioni multiple accessibili da [qui](#).**

Al di là della possibilità di consultare in modo diretto i file che sono stati prodotti, il focus principale della sperimentazione è quello di effettuare un confronto al fine di individuare eventuali discrepanze tra i risultati dei due algoritmi e comprendere le differenze dal punto di vista delle prestazioni temporali e spaziale degli stessi. **A tal fine, divengono particolarmente utili i file di confronto diretto presenti nella cartella *comparison_tables* dei due esempi portati avanti fino ad ora, di cui viene mostrato il contenuto nelle immagini sottostanti.** Ciascuna tabella è identificata dal nome del relativo file “*comparison_Mi.txt*”, dove *i* va sempre ad indicare l'esperimento ad esso associato.

Per tutti gli **esperimenti del Caso 1**, sono disponibili le seguenti tabelle:

comparison_M0.txt ×			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	4	4	no
5 N cardinality	7	7	no
6 Aggregate index	0.4643	0.4643	no
7 visited nodes	19	19	no
8 COV cardinality	3	3	no
9 execution time	0.000335 s	0.000308 s	yes [2.7e-05 s]
10			
11 COV distribution for EC: [1, 1, 1]			
12 COV distribution for EC+: [1, 1, 1]			
13 COV distributions are the same.			

comparison_M1.txt ×			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	5	5	no
5 N cardinality	15	15	no
6 Aggregate index	0.48	0.48	no
7 visited nodes	131	131	no
8 COV cardinality	12	12	no
9 execution time	0.000997 s	0.000885 s	yes [0.000112 s]
10			
11 COV distribution for EC: [1, 3, 4, 3, 1]			
12 COV distribution for EC+: [1, 3, 4, 3, 1]			
13 COV distributions are the same.			

comparison_M2.txt ×			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	6	6	no
5 N cardinality	31	31	no
6 Aggregate index	0.4678	0.4678	no
7 visited nodes	629	629	no
8 COV cardinality	52	52	no
9 execution time	0.007153 s	0.006099 s	yes [0.001055 s]
10			
11 COV distribution for EC: [1, 8, 22, 18, 3]			
12 COV distribution for EC+: [1, 8, 22, 18, 3]			
13 COV distributions are the same.			

comparison_M3.txt ×			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	8	8	no
5 N cardinality	127	127	no
6 Aggregate index	0.4685	0.4685	no
7 visited nodes	15071	15071	no
8 COV cardinality	1245	1245	no
9 execution time	0.302279 s	0.293693 s	yes [0.008585 s]
10			
11 COV distribution for EC:	[1, 30, 229, 513, 376, 90, 6]		
12 COV distribution for EC+:	[1, 30, 229, 513, 376, 90, 6]		
13 COV distributions are the same.			

comparison_M4.txt ×			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	10	10	no
5 N cardinality	511	511	no
6 Aggregate index	0.4759	0.4759	no
7 visited nodes	515582	515582	no
8 COV cardinality	56752	56752	no
9 execution time	14.928147 s	14.329868 s	yes [0.598279 s]
10			
11 COV distribution for EC:	[1, 134, 2439, 12397, 21359, 14990, 4704, 683, 44, 1]		
12 COV distribution for EC+:	[1, 134, 2439, 12397, 21359, 14990, 4704, 683, 44, 1]		
13 COV distributions are the same.			

comparison_M5.txt ×			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	12	12	no
5 N cardinality	2047	2047	no
6 Aggregate index	0.4717	0.4717	no
7 visited nodes	22953130	22953130	no
8 COV cardinality	2915956	2915956	no
9 execution time	1913.442763 s	1865.225212 s	yes [48.217552 s]
10			
11 COV distribution for EC:	[1, 527, 26613, 286808, 872669, 1015167, 542774, 148017, 21622, 1691, 66, 1]		
12 COV distribution for EC+:	[1, 527, 26613, 286808, 872669, 1015167, 542774, 148017, 21622, 1691, 66, 1]		
13 COV distributions are the same.			

La prima osservazione che si può fare su questi risultati è che, in tutti gli esperimenti del *Caso 1* di sperimentazione multipla, l'esecuzione è andata sempre a buon fine. Inoltre, ad eccezione della variabile *execution time*, si sono ottenuti sempre valori equivalenti tra i risultati dei due algoritmi *EC* ed *EC⁺*,

fattore che implica la mancanza di errori durante l'esecuzione degli stessi. Soffermandosi sugli aspetti principali, si nota che i nodi visitati sono esattamente gli stessi (*i due algoritmi devono effettuare la visita degli alberi allo stesso modo, osservando e scartando gli stessi nodi*), così come lo sono le cardinalità di *COV* (*i due algoritmi devono ottenere le stesse partizioni in uscita*) e le relative distribuzioni di cardinalità.

L'unica differenza sta nel tempo di esecuzione. Infatti, leggendo i valori di *execution time* tra i vari casi, si può osservare come l'algoritmo plus sia in grado di risolvere il medesimo problema di Exact Cover in un tempo minore rispetto all'algoritmo base; ciò si può notare con maggiore evidenza per i casi con cardinalità di *COV* più elevata. Ciò può essere giustificato dalla mancata computazione dell'unione di insiemi per comporre aggregati, che in *EC_PLUS* ed *ESPLORA_PLUS* viene sostituita da una semplice somma tra le singole cardinalità dei papabili insiemi da unire in aggregato.

Maggiori osservazioni sono reperibili dei grafici della sezione [Visualizzazione degli andamenti](#) a seguire, in grado di rendere più veloce ed evidenti gli aspetti legati ai tempi di esecuzione per i problemi di questo primo caso in esame.

Ora sono invece presentati i risultati degli **esperimenti del Caso 2**:

comparison_M0.txt ×			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	4	4	no
5 N cardinality	7	7	no
6 Aggregate index	0.6429	0.6429	no
7 visited nodes	16	16	no
8 COV cardinality	2	2	no
9 execution time	4e-05 s	3.5e-05 s	yes [5e-06 s]
10			
11 COV distribution for EC: [1, 1]			
12 COV distribution for EC+: [1, 1]			
13 COV distributions are the same.			

comparison_M1.txt ×			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	5	5	no
5 N cardinality	15	15	no
6 Aggregate index	0.6267	0.6267	no
7 visited nodes	93	93	no
8 COV cardinality	5	5	no
9 execution time	0.000388 s	0.000183 s	yes [0.000204 s]
10			
11 COV distribution for EC: [1, 4]			
12 COV distribution for EC+: [1, 4]			
13 COV distributions are the same.			

comparison_M2.txt ×			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	6	6	no
5 N cardinality	31	31	no
6 Aggregate index	0.6398	0.6398	no
7 visited nodes	465	465	no
8 COV cardinality	6	6	no
9 execution time	0.000397 s	0.000343 s	yes [5.4e-05 s]
10			
11 COV distribution for EC: [1, 5]			
12 COV distribution for EC+: [1, 5]			
13 COV distributions are the same.			

comparison_M3.txt ×			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	8	8	no
5 N cardinality	127	127	no
6 Aggregate index	0.6211	0.6211	no
7 visited nodes	145	145	no
8 COV cardinality	19	19	no
9 execution time	0.005053 s	0.004798 s	yes [0.000255 s]
10			
11 COV distribution for EC: [1, 18]			
12 COV distribution for EC+: [1, 18]			
13 COV distributions are the same.			

comparison_M3.txt ✕			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	8	8	no
5 N cardinality	127	127	no
6 Aggregate index	0.6211	0.6211	no
7 visited nodes	7913	7913	no
8 COV cardinality	19	19	no
9 execution time	0.00561 s	0.005515 s	yes [9.5e-05 s]
10			
11 COV distribution for EC: [1, 18]			
12 COV distribution for EC+: [1, 18]			
13 COV distributions are the same.			

comparison_M4.txt ✕			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	10	10	no
5 N cardinality	511	511	no
6 Aggregate index	0.6084	0.6084	no
7 visited nodes	130462	130462	no
8 COV cardinality	75	75	no
9 execution time	0.448206 s	0.438148 s	yes [0.010058 s]
10			
11 COV distribution for EC: [1, 62, 8, 3, 1]			
12 COV distribution for EC+: [1, 62, 8, 3, 1]			
13 COV distributions are the same.			

comparison_M5.txt ✕			
1 Variable	EC	EC+	Different?
2 -----	-----	-----	-----
3 execution ended	yes	yes	no
4 M cardinality	12	12	no
5 N cardinality	2047	2047	no
6 Aggregate index	0.5992	0.5992	no
7 visited nodes	2096505	2096505	no
8 COV cardinality	372	372	no
9 execution time	10.362953 s	10.242891 s	yes [0.120062 s]
10			
11 COV distribution for EC: [1, 267, 55, 44, 5]			
12 COV distribution for EC+: [1, 267, 55, 44, 5]			
13 COV distributions are the same.			

Anche per il Caso 2 si osservano sia esecuzioni andate tutte a buon fine, sia assenza di discordanze tra le variabili. L'unico valore che si distingue tra le due

versioni dell'algoritmo è il tempo di esecuzione, dove il lasso di tempo relativo ad EC^+ risulta essere comunque più breve rispetto a quello di EC (seppur di pochi secondi, stando soprattutto ai primi esperimenti).

Andando a commentare le *differenze tra i due interi casi d'esempio*, che presentano entrambi la medesima cardinalità dal punto di vista di M ed N nei vari esperimenti, si possono notare alcune variazioni:

- *aggregate index*: nel *Caso 1*, i valori degli *aggregate index* sono tutti più bassi rispetto a quelli del *Caso 2*;
- *visited nodes*: in generale, i nodi visitati nel *Caso 1* sono molti di più rispetto a quelli visitati nel *Caso 2*;
- *COV cardinality*: le partizioni in *COV*, risultati in output dagli algoritmi, sono in più nel *Caso 1* che nel *Caso 2*;
- *execution time*: i tempi di esecuzione degli algoritmi, soprattutto per l'ultimo esperimento di entrambi i casi, sono più elevati nel *Caso 1* rispetto al *Caso 2*;
- *COV distribution*: si può notare come, nel *Caso 1*, le partizioni ottenute siano distribuite su più valori di cardinalità, a differenza del *Caso 2* che, invece, possiede partizioni con cardinalità limitate a pochi valori (e relativamente bassi).

La presenza di una *correlazione* tra tali valori viene indagata nella prossima sezione, in cui attraverso i grafici si riesce ad osservare con maggiore semplicità ed efficacia il comportamento dei risolutori rispetto ai casi d'esempio.

7.2.5 Visualizzazione degli andamenti

Per andare a completare la sperimentazione multipla, si passa infine ad effettuare una visualizzazione degli andamenti di tutti i casi presi in esame nei punti precedenti. Sfruttando le liste che si sono costruite durante l'esecuzione del codice presentato fino ad ora (citate in [Controllo multiplo](#)) è possibile infatti realizzare dei *grafici che mostrino, in modo intuitivo, come si muovono alcuni dei valori registrati in base ai diversi input dati*, ovvero le diverse cardinalità del dominio M (presentate nella lista $MlenList$) e, conseguentemente, quelle relative alla collezione N (nella lista $NlenList$)

Il modulo Python utilizzato per la stampa dei grafici è *pyplot*, contenuto a sua volta in *matplotlib*, una libreria ampiamente utilizzata per la visualizzazione di risultati per scopi analitici. Essa dà la possibilità di costruire tali grafici basandosi sul contenuto di liste fornite, visualizzare direttamente in *Colab* i risultati ottenuti e, inoltre, salvare gli stessi in file immagine. A tal proposito, si è deciso che le immagini risultanti fossero inserite, ad ogni stampa, nella sottocartella di *comparison* dal nome *images*.

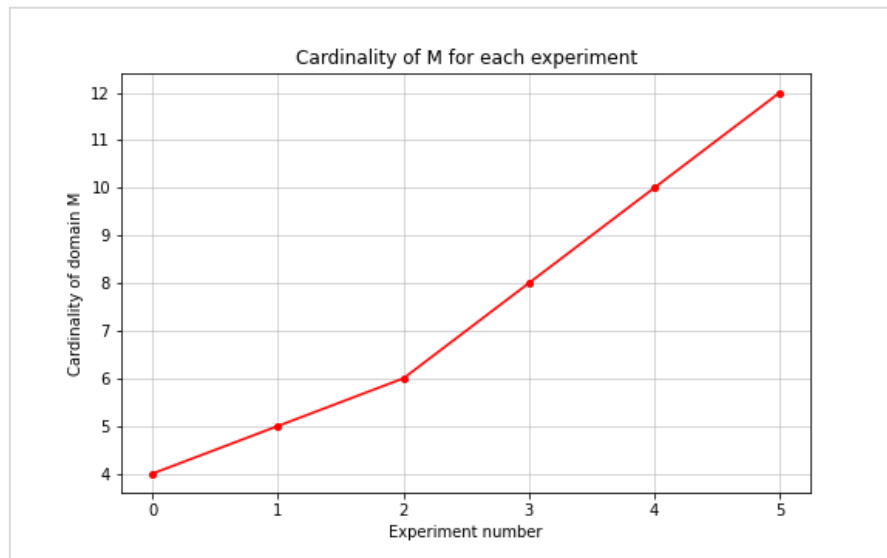
Qui di seguito sono presentati gli aspetti su cui ci si è voluti soffermare per avere un'idea del comportamento degli algoritmi all'interno della sperimentazione multipla. **Ognuno di questi aspetti è accompagnato dal grafico che si è ottenuto seguendo nuovamente gli esempi portati avanti fino ad ora. Gli stessi grafici sono disponibili nella cartella *images* delle due sperimentazioni multiple accessibili da [qui](#).**

1. Evoluzione della cardinalità del dominio M

Il primo grafico realizzato permette di avere un'idea delle associazioni tra il numero dell'esperimento che viene ispezionato all'interno della sperimentazione multipla e il valore della cardinalità di M fissato per quello stesso esperimento. Sull'asse delle ascisse si hanno quindi dei numeri che vanno da 0 a $|MlenList| - 1$ ad indicare gli esperimenti, e sulle ordinate il valore di $|M|$. Il numero

dell'esperimento permette di capire quali file sono ad esso associati: ad esempio, come specificato in precedenza, il file “*input_M0.txt*” sarà direttamente associato al primo esperimento, quello indicato con il valore *0* sull'asse delle ascisse, così come accade per i file “*output_M0.txt*”, “*output_M0_plus.txt*” e “*comparison_M0.txt*”.

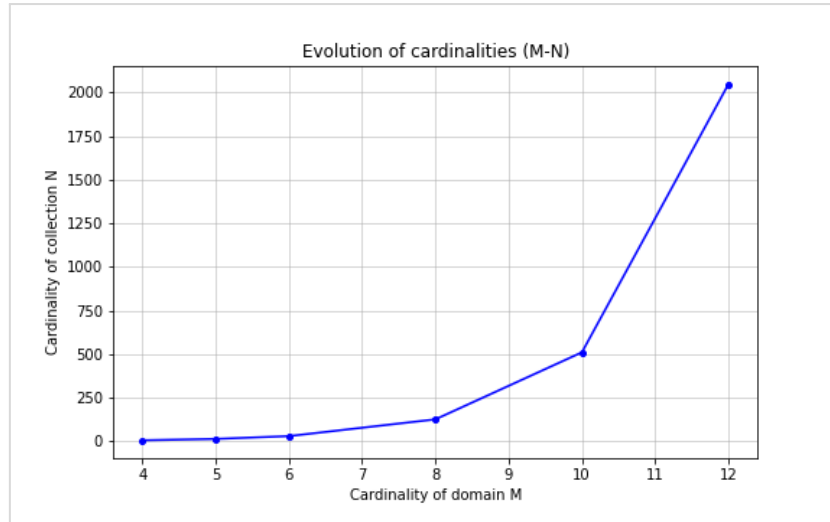
Per entrambi gli **esempi** riportati, si ottiene tale grafico:



2. Evoluzione della cardinalità della collezione *N* al variare della cardinalità del dominio *M*

Questo grafico permette di visualizzare la crescita del valore di *N* all'aumento del valore della cardinalità di *M*. In particolare, seguendo quanto definito nelle specifiche riportate in precedenza, tali valori sono salvati nelle liste *MlenList* ed *NlenList*. Il valore della cardinalità di *N* è calcolato sulla base della cardinalità di *M* del medesimo esperimento associato; pertanto, il grafico presenta sull'asse delle ascisse le diverse cardinalità di *M*, e sull'asse delle ordinate le relative cardinalità di *N*.

Per entrambi i casi in **esempio**, si ottiene il grafico a seguire.



Si nota quindi come, in questo caso, i valori di $|N|$ crescano sempre più all'aumentare di $|M|$. Da ciò si può supporre che, per ciascun esperimento, il numero di iterazioni all'interno di *EC* ed *EC_PLUS* aumenterà, in quanto l'algoritmo dovrà visitare come minimo la radice e tutti i nodi di primo livello di ciascun albero radicato negli insiemi di N . Nei prossimi punti sarà possibile osservare maggiormente ciò che questo comportamento potrebbe causare.

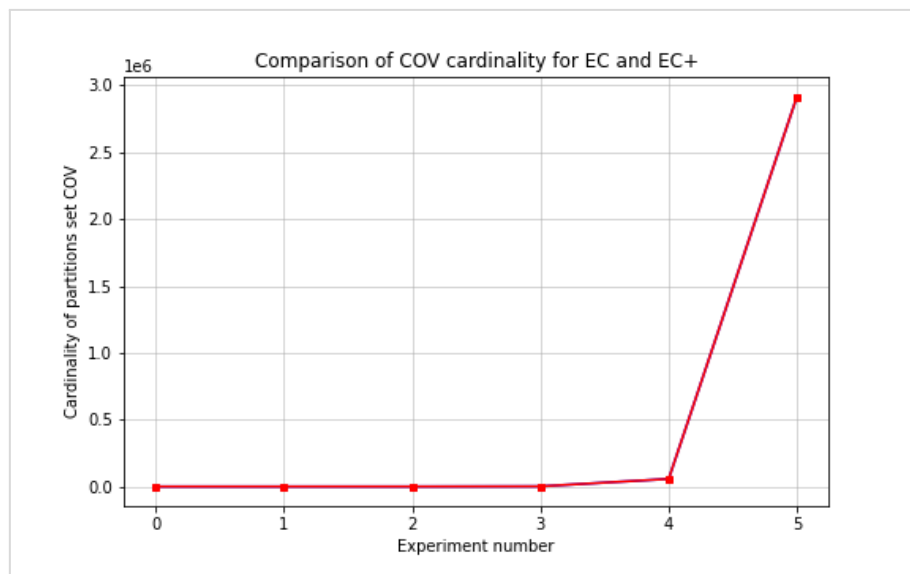
3. Evoluzione della cardinalità dell'insieme delle partizioni COV

Ci si sofferma ora sul grafico relativo all'evoluzione della cardinalità dell'insieme contenente le partizioni in uscita, ovvero *COV*. Esso presenta sull'asse delle ascisse il numero dell'esperimento associato, come accadeva per il grafico delle cardinalità di M al *punto 1*, e sull'asse delle ordinate la cardinalità di *COV* associata. Tramite questo grafico è possibile avere una visione delle variazioni di $|COV|$ tra gli esperimenti prodotti. Oltre a questo aspetto, grazie alla stessa rappresentazione viene resa automaticamente nota la presenza di errori durante l'esecuzione. Infatti, i due algoritmi dovrebbero essere in grado di produrre esattamente lo stesso numero di partizioni: ad ogni linea presentata nel grafico è associato il comportamento di uno dei due algoritmi, e in presenza di linee non coincidenti si ha evidenza di un'inconsistenza. Individuando il numero dell'esperimento per cui è presente tale inconsistenza, e recuperando il file di

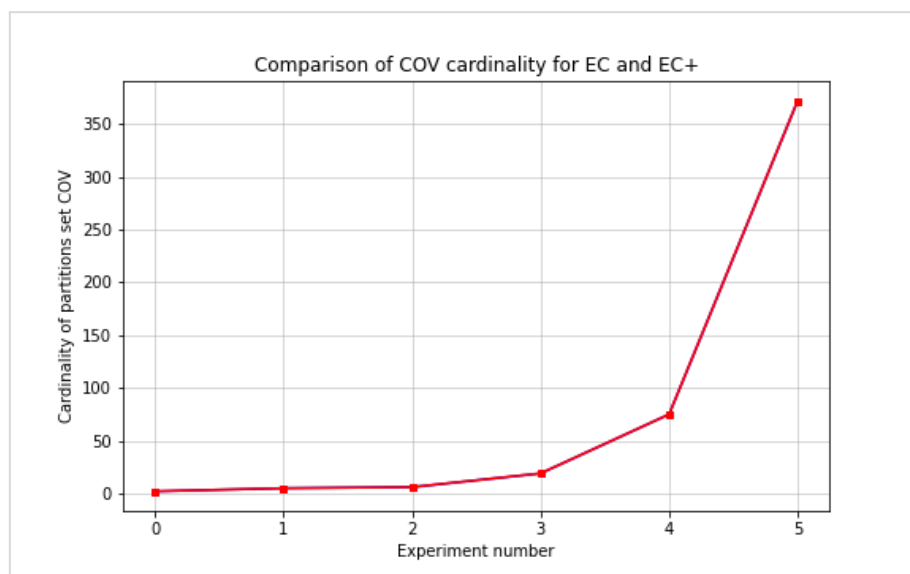
confronto diretto associato a tale valore, è possibile avere idea delle motivazioni di tale comportamento (come, ad esempio, la conclusione anticipata della computazione dovuta ad un blocco utente).

Per i due casi in **esempio**, si ottengono i seguenti grafici, dove il primo si riferisce al *Caso 1*, mentre il secondo al *Caso 2*.

Caso 1



Caso 2



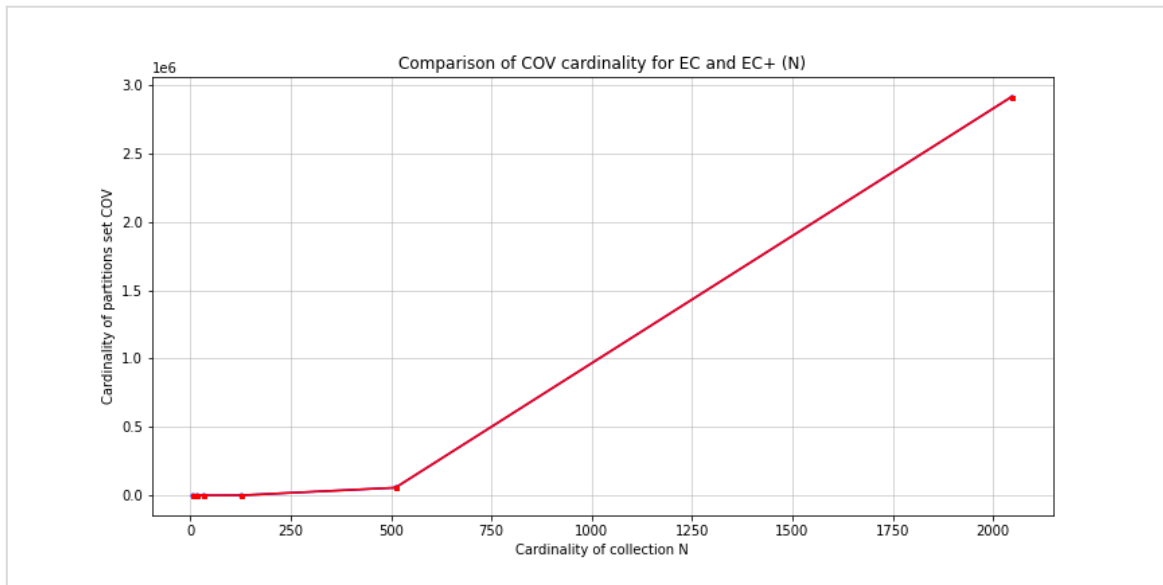
Innanzitutto, si nota come non vi siano inconsistenze o errori tra i due risolutori per entrambe le sperimentazioni multiple (vi è un'unica linea per ciascun grafico). Inoltre, si osserva che il valore della cardinalità di COV cresce avanzando con gli esperimenti, e ciò dipende dalla variazione degli input, ovvero il valore di $|M|$ e, soprattutto, il valore di $|N|$. Tuttavia, si può notare una netta differenza tra il primo caso e il secondo: infatti, nel *Caso 1* il numero di partizioni presenti in COV aumenta drasticamente avanzando con gli esperimenti rispetto a quanto accade per il *Caso 2*. Questa variazione è un effetto delle diverse distribuzioni di cardinalità degli insiemi di N che dipendono dal caso. Per avere maggiori dettagli riguardo agli aspetti sopracitati, si rimanda a quanto viene detto nelle sezioni a seguire.

4. Evoluzione della cardinalità dell'insieme delle partizioni COV al variare della cardinalità della collezione N

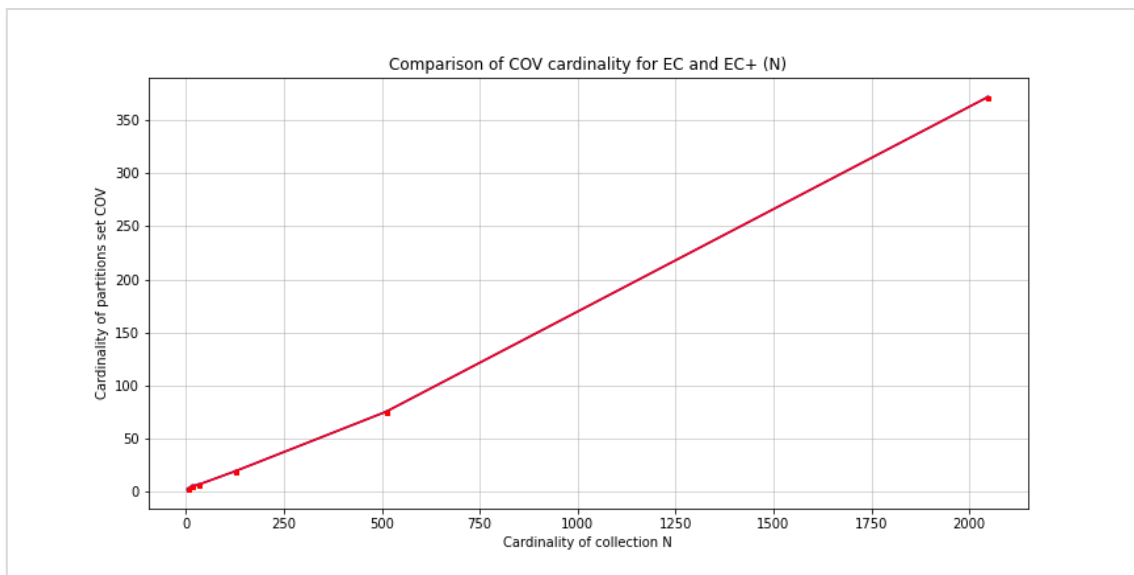
La cardinalità di COV dipende fortemente dagli insiemi che sono coinvolti all'interno della collezione N . Infatti, aumentando la cardinalità di N aumenta anche la probabilità che all'interno di tale insieme siano presenti sottoinsiemi in grado di produrre nuove partizioni di N , soprattutto nel caso in cui tale insieme abbia una buona distribuzione di cardinalità (ovvero sia composto da insiemi che coprano adeguatamente tutte le possibili cardinalità comprese tra 1 e $|M|$, come specificato in [Generazione del file di input](#)). Il grafico ottenuto in questo caso presenta, quindi, sull'asse delle ascisse la cardinalità della collezione N di ciascun esperimento analizzato, e sull'asse delle ordinate la cardinalità associata all'insieme delle partizioni COV generato per quello stesso caso.

Per i due casi in **esempio**, si ottengono i seguenti due grafici:

Caso 1



Caso 2



È reso evidente quanto la cardinalità di COV sia impattata fortemente dal numero di insiemi componenti N : per $|N|$ bassi, la cardinalità di COV non varia troppo da un caso all'altro, ma al crescere di $|N|$, soprattutto quando si passa da un ordine di grandezza per $|N|$ ad uno più elevato, si ha un netto cambiamento.

Infatti, aumentando tale valore, aumenta la probabilità di avere insiemi in grado di creare aggregati e, di conseguenza, anche di creare delle nuove partizioni. Ciò, tuttavia, non dipende solamente dal numero di insiemi in N , ma anche dalla distribuzione della loro cardinalità. Tale aspetto è quindi indagato al punto seguente, sfruttando i valori degli *aggregate index*.

5. Valori degli *aggregate index* in base all'esperimento in analisi

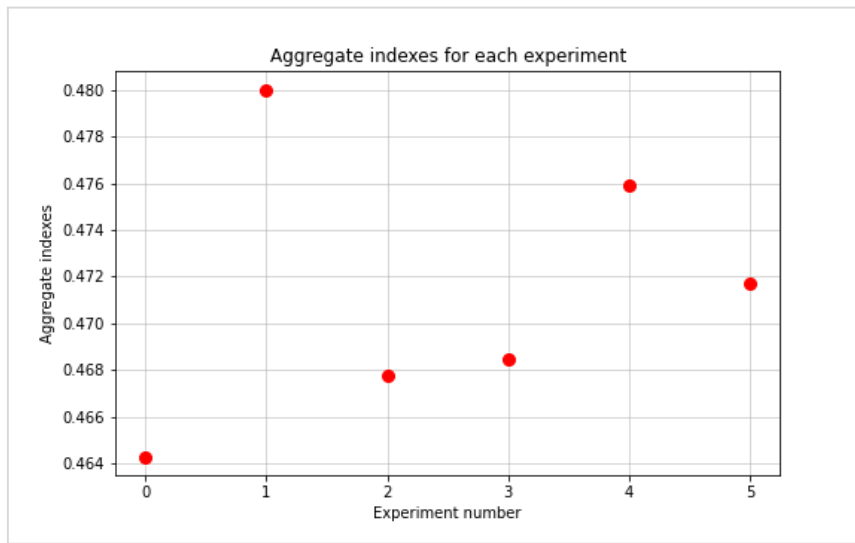
Il grafico di questa sezione vuole presentare, attraverso dei semplici punti, il valore dell'*aggregate index* (presente sull'asse delle ordinate) di ciascuno degli esperimenti effettuati (indicato dal numero sull'asse delle ascisse). Come già specificato, l'*aggregate index* è un indice utile a comprendere, in modo riassuntivo, la distribuzione di cardinalità degli insiemi presenti in N .

- Se tale valore è vicino a 1 significa che la maggior parte dei sottoinsiemi di N avrà cardinalità tendente ad $|M|$. Ciò significa anche che durante l'esecuzione dell'algoritmo ci sarà minore probabilità di ottenere aggregati o (se presenti) partizioni di cardinalità ≥ 2 ;
- Al contrario, per valori vicini a 0, si avrà una maggioranza di insiemi di N con cardinalità bassa (tendente a 1). Di conseguenza, si avranno tanti insiemi di N in grado di creare aggregati e/o partizioni di cardinalità ≥ 2 .

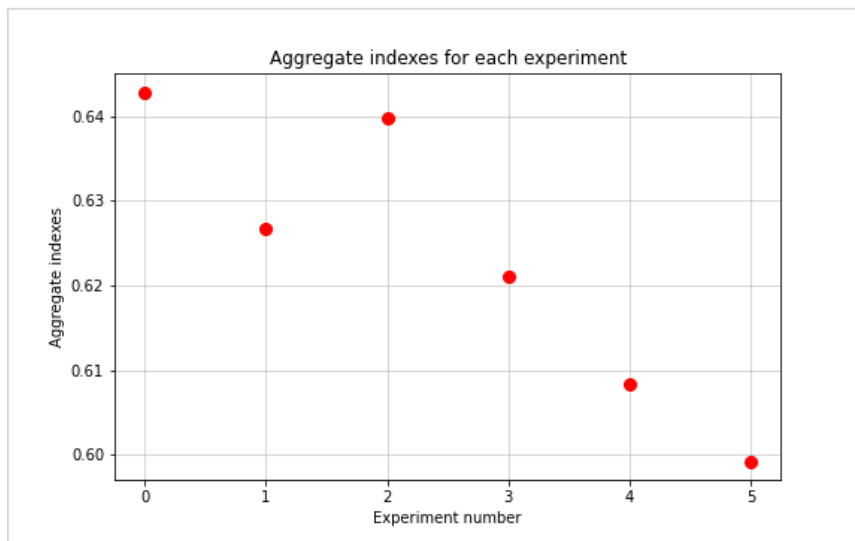
L'*aggregate index* può essere utile a comprendere maggiormente le motivazioni relative alla presenza o meno di un determinato numero di nodi visitati, del numero di partizioni (in *COV*) ottenute e del tempo relativo all'esecuzione degli algoritmi.

Per i due casi in **esempio**, si ottengono i seguenti grafici, dove il primo si riferisce al *Caso 1*, mentre il secondo al *Caso 2*.

Caso 1



Caso 2



Per il *Caso 1*, può notare che tutti i valori dell'indice appartengono all'intervallo $[0.464, 0.480]$. Ciò significa che gli insiemi di N , mediamente, presentano una cardinalità poco minore della metà di $|M|$ del relativo esperimento, e che quindi vi sarà buona probabilità di ottenere nuovi aggregati, anche associati a partizioni di cardinalità ≥ 2 . Ciò va quindi a giustificare la presenza di un così alto numero di partizioni trovate, soprattutto quando si ha la disponibilità di una maggiore quantità di insiemi su cui trovarle (ovvero, per cardinalità di N elevate).

Per il *Caso 2*, invece, si nota invece che i valori di *aggregate index* sono compresi nell'intervallo $[0.5992, 0.6429]$. Tali valori implicano che gli insiemi di N abbiano, mediamente, una cardinalità che tende a quella massima ($|M|$ per il relativo dominio M) e che quindi vi sarà una minore probabilità di creare aggregati e, di conseguenza, partizioni di cardinalità ≥ 2 . Infatti, sottoinsiemi di N di cardinalità elevata possono formare partizioni con sottoinsiemi di N di cardinalità bassa, e nel caso di *aggregate index* alti la quantità di questi ultimi insiemi risulta essere bassa rispetto ai primi, e conseguentemente sarà basso il numero di aggregati e/o partizioni ottenibili. Ecco perché, nel *Caso 2*, il numero di partizioni in uscita ai suoi esperimenti è così basso rispetto a quello del *Caso 1*.

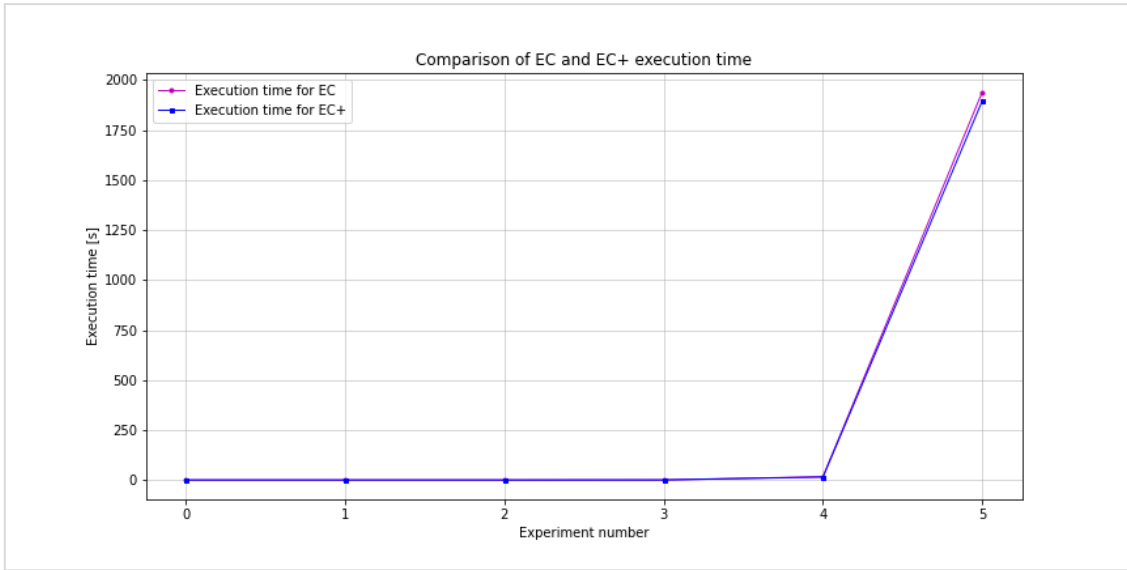
6. *Evoluzione del tempo di esecuzione di EC e di EC⁺*

Uno dei principali aspetti su cui si vuole effettuare un'indagine è quello delle *prestazioni temporali* degli algoritmi implementati. Infatti, avanzando con i casi sperimentali si possono notare quelli che sono i cambiamenti più o meno evidenti del lasso di tempo che viene impiegato dall'algoritmo base e dall'algoritmo plus per risolvere il problema di Exact Cover.

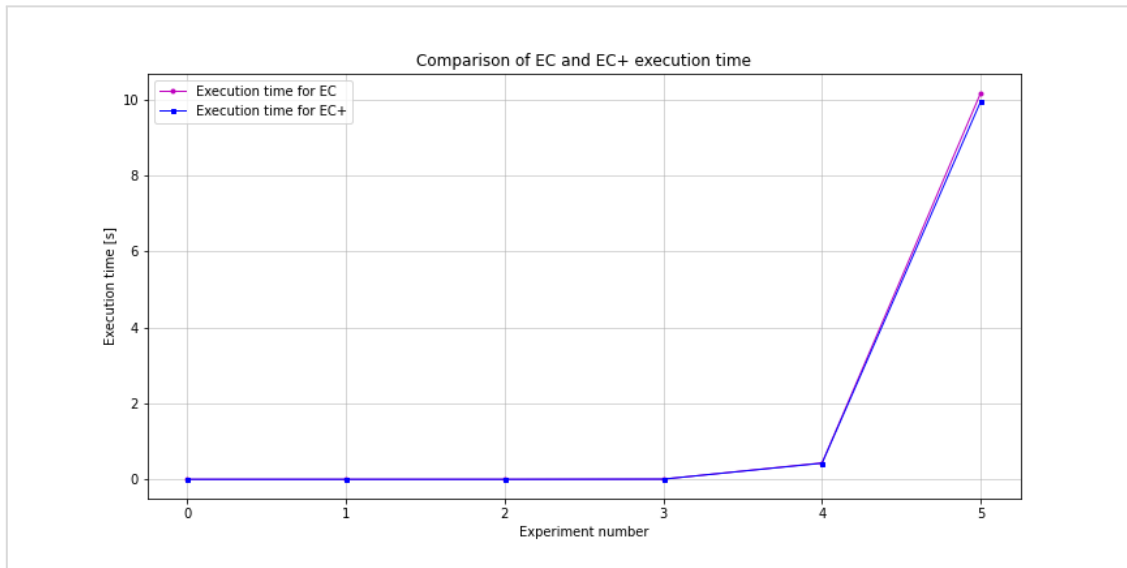
Il grafico che si ottiene presenta sulle ascisse il numero che identifica i diversi esperimenti effettuati, e sulle ordinate i valori del tempo (in *secondi*) impiegato durante l'esecuzione dell'esperimento associato. Dalla sua interpretazione è possibile non solo visualizzare la variazione di tali valori, ma anche rendere noto il comportamento di un algoritmo rispetto all'altro: infatti, essendo diverse alcune delle operazioni che vengono effettuate al loro interno, vi è la possibilità che uno sia più veloce ad ottenere le partizioni in uscita rispetto all'altro, e ciò viene evidenziato dal valore dell'ascissa delle due linee relative ai due comportamenti fissato un certo numero sull'ordinata (quindi, scelto un esperimento da osservare).

Qui di seguito, i due grafici associati ai casi di *esempio*.

Caso 1



Caso 2



In entrambe le sperimentazioni multiple effettuate si può osservare una crescita nel tempo di esecuzione degli algoritmi. Questo primo aspetto è giustificato dal fatto che, durante la sua esecuzione, ciascun esperimento lavora con un numero sempre crescente di sottoinsiemi di N , sia per l'algoritmo EC che per l'algoritmo EC^+ , e quindi dovrà effettuare, come minimo, la visita di tutti le radici e tutti i

nodi di primo livello degli alberi che hanno come radice tali insiemi (valore che, ovviamente, cresce all'aumentare di $|N|$).

Un altro aspetto che si osserva per entrambi i casi è che l'algoritmo EC^+ risulta essere più efficiente rispetto ad EC . Infatti, il tempo che viene impiegato dall'algoritmo plus a costruire le medesime partizioni costruite dall'algoritmo base è minore, e ciò è principalmente dovuto alla mancanza della computazione dell'unione degli insiemi (effettuata, nel codice implementato, lavorando con liste e confronti tra variabili) per cui viene valutata la presenza o meno di un nuovo aggregato o di una nuova partizione, a favore dell'utilizzo della semplice somma tra le cardinalità degli stessi insiemi. Tale andamento risulta non essere particolarmente evidente per esperimenti che presentano input di cardinalità bassa, quanto piuttosto su casi che racchiudono un maggior numero di casi su cui lavorare per recuperare le partizioni.

Andando ad osservare, infine, le differenze tra i due casi d'esempio, si può osservare che il tempo di esecuzione nel *Caso 1* è molto più lungo rispetto a quello del *Caso 2* (soprattutto per l'esperimento finale). Ciò dipende, nuovamente, dall'*aggregate index*. Come si è visto dal grafico al punto precedente, il *Caso 1* presenta valori minori degli indici che implicano la presenza di un maggior numero di sottoinsiemi di N di cardinalità bassa, in grado di costruire un maggior numero di aggregati di cardinalità ≥ 2 . Quando tali aggregati vengono rilevati, viene eseguita la chiamata della funzione *ESPLORA* per l'algoritmo base ed *ESPLORA_PLUS* per l'algoritmo plus, portando alla visita di nuovi nodi, all'esecuzione di nuovi controlli per la ricerca di ulteriori aggregati e, in caso, alla chiamata ricorsiva delle stesse funzioni. Ciò si traduce così in un aumento del tempo complessivo di esecuzione delle funzioni *EC* ed *EC_PLUS* dato un certo input. Questo aumento è molto meno evidente per il *Caso 2*, dove (come conferma il valore più elevato degli *aggregate index*) si ha una collezione N composta da sottoinsiemi di cardinalità più elevata, su cui risulta più difficile costruire aggregati e trovare partizioni, e che quindi porteranno gli algoritmi ad esaurire la computazione in anticipo (numero minore di chiamate di *ESPLORA* ed *ESPLORA_PLUS*, minor numero di confronti).

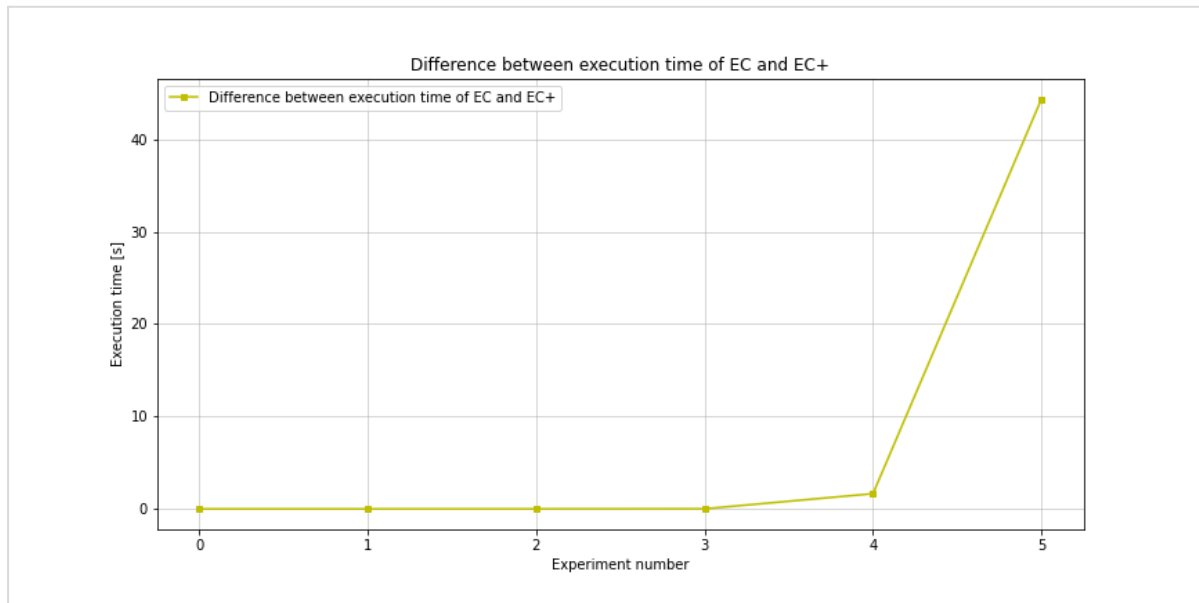
La distribuzione di cardinalità di COV , presentata nelle tabelle di confronto al termine di [Esecuzione degli algoritmi](#) per i due esempi, può confermare quanto detto: il *Caso 2* realizza partizioni di cardinalità minore rispetto a quanto viene realizzato dal *Caso 1*, ad indicare quindi un minor numero di insiemi compatibili nel primo caso causato dalla cardinalità elevata (più vicina a quella di M) dei sottoinsiemi di N .

7. Evoluzione della differenza dei tempi di esecuzione di EC e di EC^+

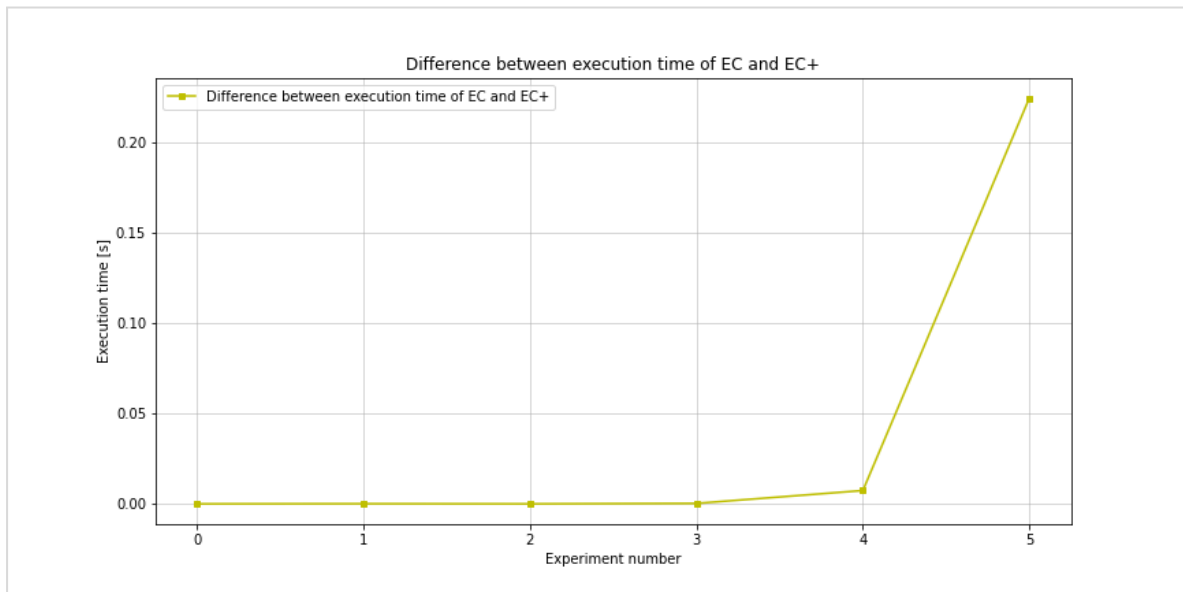
Il grafico definito in questo punto si sofferma invece sulla differenza tra i tempi di esecuzione dei due algoritmi. Ciò permette di capire quanto tempo intercorra tra il termine dell'esecuzione di EC ed il termine dell'esecuzione di EC^+ direttamente relativamente agli stessi esperimenti, e quindi al medesimo valore di input $|M|$. Sulle ascisse sono quindi presenti i valori di cardinalità di M , e sulle ordinate il valore della differenza dei tempi appena esplicitati. Quando il valore dell'ordinata associato ad un certo $|M|$ è positivo significa che l'algoritmo base ha impiegato più tempo a svolgere il suo lavoro; al contrario, ad un valore negativo dell'ordinata è associata la presenza di un'esecuzione più lenta dell'algoritmo plus rispetto a quello base.

Per i due casi in ***esempio***, si ottengono i grafici presentati nella pagina a seguire.

Caso 1



Caso 2



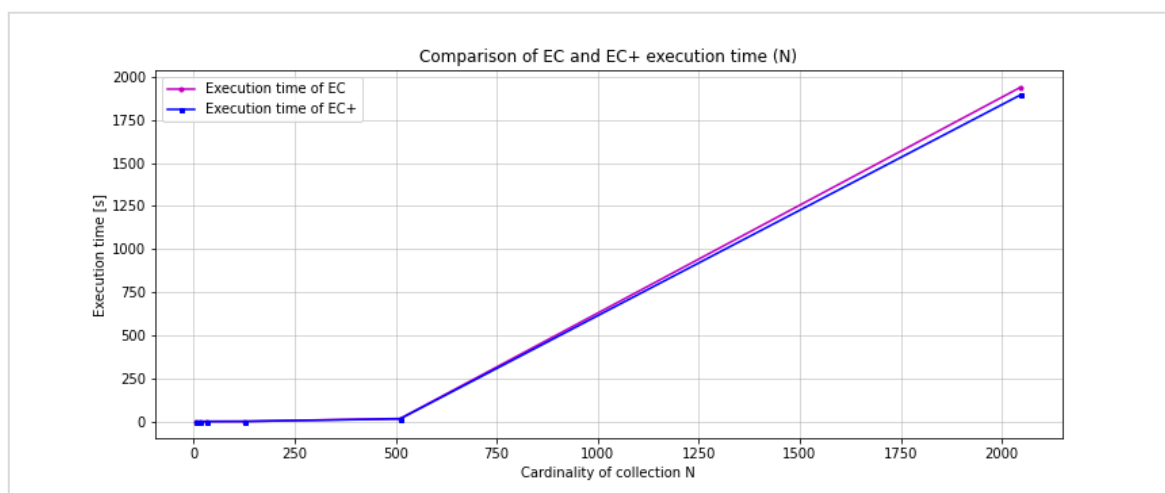
Anche in questo caso valgono le medesime osservazioni fatte al punto precedente. Si rende evidente, però, come la differenza tra il tempo di esecuzione dei due algoritmi sia maggiore in presenza di un maggior numero di casi (sottoinsiemi di N) da analizzare, fattore che spinge alla scelta dell'algoritmo plus su input grandi rispetto all'algoritmo base.

8. Evoluzione del tempo di esecuzione di EC e di EC^+ al variare della cardinalità della collezione N

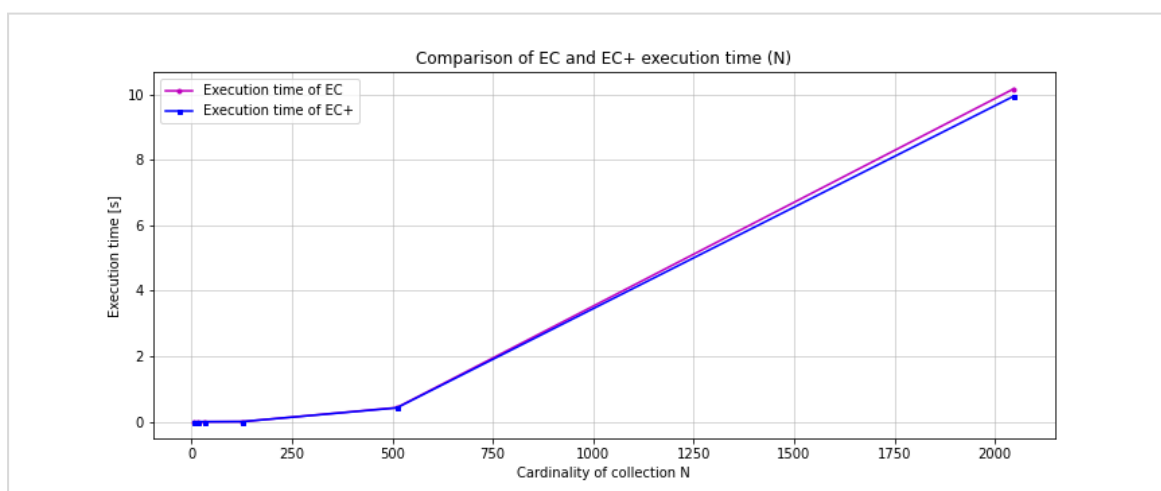
Anche in questo caso si può osservare quanto varia il tempo di esecuzione rispetto alla dimensione della collezione N in input agli algoritmi. Sulle ascisse del grafico sono presenti, quindi, tutte le singole cardinalità di N , mentre sulle ordinate si hanno i tempi di esecuzione (in *secondi*). Il numero di sottoinsiemi di N impatta particolarmente sui risultati, in quanto l'aumento di $|N|$ implica una crescita del numero di iterazioni all'interno di EC per esplorare tutti gli alberi che hanno come radice ciascun sottoinsieme di N e, conseguentemente, maggiore tempo impiegato per eseguire tali visite e le operazioni ad esse associate.

Si presentano ora, qui di seguito, i grafici associati ai due casi di **esempio**.

Caso 1



Caso 2



Anche in questo caso, sono valide tutte le osservazioni fatte ai due punti precedenti. Ci si focalizza maggiormente, però, su ciò che riguarda gli ordini di grandezza: infatti, la crescita di un ordine di grandezza per la cardinalità di N , da 2 a 3, provoca la variazione di un solo ordine di grandezza in più per i tempi di esecuzione nel *Caso 2*, e una variazione molto più grande (tre ordini di grandezza in più) per i tempi del *Caso 1*. È quindi rilevante non solo la cardinalità della collezione N , ma anche la distribuzione di cardinalità degli insiemi che la compongono.

Al termine dell'esecuzione di ciascuna delle sperimentazioni multiple, viene effettuata in modo automatico la compressione della cartella *comparison* relativa ai rispettivi esperimenti, la quale conterrà quindi tutte le cartelle presentate (*comparison_tables*, *images*, *input*, *output*, *output+*), che al loro volta conterranno i relativi file costruiti. Viene in questo modo creato un file "*experiments.zip*" e reso disponibile nuovamente nella sezione File di Colab. In aggiunta, esso viene automaticamente scaricato sul dispositivo in uso.

8. Conclusioni e osservazioni finali

Prima di portare a conclusione questo elaborato, ci si vuole soffermare su tutti ciò che è stato osservato durante l'implementazione dei programmi richiesti e, in particolar modo, le principali problematiche riscontrate e i risultati che si sono ottenuti, al fine di ottenere un resoconto del comportamento complessivo di quanto si è sviluppato.

8.1 Limitazioni e problemi riscontrati

I programmi presentano alcune limitazioni. Per quanto riguarda i parametri di input ai risolutori, per esecuzioni relative a problemi con cardinalità del dominio M maggiore di 15 e, soprattutto, in presenza di un elevato numero di insiemi nella collezione N (nell'ordine delle migliaia e per ordini superiori alle migliaia) si iniziano ad avere tempi di esecuzione più lunghi (superiori ad un'ora di esecuzione per il singolo algoritmo). Inoltre, come si è visto, anche la distribuzione di cardinalità degli insiemi di N risulta avere un impatto sui tempi, provocandone la crescita soprattutto in presenza di un elevato numero di insiemi di cardinalità bassa su cui è più semplice trovare aggregati e, di conseguenza, aumentare il numero di confronti ed operazioni da effettuare.

Il problema di Exact Cover è, in effetti, classificato come *NP-completo*: appartiene ad una classe contenuta in quella dei problemi *NP* (*Non Deterministic Polynomial*) per la quale non è stato trovato alcun algoritmo risolvibile in grado di operare in tempo polinomiale. Se un problema è NP-completo, si ha la (quasi) certezza della sua intrattabilità. Per questo motivo, ci si muove verso lo sviluppo di algoritmi risolvibili che siano in grado di cercare delle soluzioni che siano quasi ottime (algoritmi approssimati), verso l'utilizzo di tecniche euristiche, randomizzate, oppure verso il rimpicciolimento del problema (ad esempio, restringendo lo spazio di ricerca attribuendo valori fissi a certi parametri del problema, così da cercare soluzioni di casi trattabili). Nel caso delle soluzioni

implementate per la ricerca delle coperture esatte, si è optato per più possibilità implementative:

- si è scelto di realizzare principalmente soluzioni con valori di cardinalità di M e di N non troppo elevate, restringendo così lo spazio di ricerca a problemi che possano essere trattati ed analizzati; inoltre, il valore di tali parametri di input è estratto in modo pseudo-casuale, andando ad ampliare le possibilità di analisi e di confronto tra i casi;
- attraverso lo stop automatico è possibile ottenere almeno una parte di tutte le partizioni possibili, evitando che l'esecuzione prosegua per tempi particolarmente lunghi, soprattutto in presenza di input (cardinalità di N) troppo elevati. Si consideri, infatti, che per i problemi NP si può raggiungere un tempo al più esponenziale, e pertanto è preferibile evitare esecuzioni longeve (che superano, quindi, il tempo massimo fissato ad un'ora);
- anche attraverso lo stop manuale è possibile ottenere una soluzione intermedia (non totale e non ottima) del problema.

Si vuole far notare anche un'ulteriore limitazione per quanto riguarda la generazione degli insiemi M ed N (e quindi dell'input): la scelta di adottare come elementi del dominio M solamente delle coppie lettera-cifra araba implica che esiste una cardinalità massima raggiungibile di M del valore di 260, in quanto vengono utilizzate solo lettere comprese tra la a e la z minuscole (26 lettere) e cifre tra 0 e 9 (10 numeri). Si è scelto di adottare comunque tale formato in quanto non vengono effettuati test che superino tale valore, come specificato in precedenza, e per rimanere fedeli a quanto specificato nei dettagli dell'elaborato e nell'introduzione forniti. Per quanto riguarda, invece, la cardinalità di N , essa viene limitata al massimo numero di combinazioni semplici di M di tutte le classi che vanno da 1 a $|M|$ ottenibili. Per evitare computazioni troppo lunghe, si è scelto di estrarne un numero di insiemi di N che sia compreso tra questo massimo ed 1.

Infine, ulteriori limitazioni sono dettate dall'uso di determinate strutture dati. Da questo punto di vista si è cercato di trovare un compromesso tra le

prestazioni temporale e quelle spaziali. Infatti, a scapito della memoria, si è scelto spesso di utilizzare liste, tuple e set, per garantire una maggiore velocità di esecuzione (soprattutto grazie alle funzioni che queste stesse strutture rendono disponibili per lavorare con esse), come accade ad esempio per la matrice A che, per cardinalità di N molto elevate, porta ad occupare molto spazio. Al contrario, si è deciso di dare meno importanza al tempo di esecuzione in altre situazioni, come accade ad esempio per la matrice di compatibilità B : implementandola attraverso la matrice sparsa LIL si evita di occupare inutilmente la memoria relativa alle celle di valore 0 e la occupa solamente per gli elementi non nulli, ma nel contempo viene impiegato maggior tempo sia nell'aggiunta che nella ricerca dei valori al suo interno.

8.2 Risultati ottenuti

I programmi implementati sono in grado di risolvere quanto definito dai requisiti richiesti, funzionali e non funzionali, e di assolvere ai propri compiti. In particolare, l'obiettivo principale raggiunto è stato quello di implementare adeguatamente gli algoritmi base e plus in modo da riuscire, così, a risolvere il problema di Exact Cover, seppur mantenendo le limitazioni di cui si è trattato nel sottocapitolo precedente.

Ciò che si può evincere, soprattutto grazie alla sperimentazione effettuata, è che *il comportamento dell'algoritmo è largamente dipendente dalle operazioni che esso svolge, dalle strutture dati che utilizza, da come queste strutture dati vengono gestite al suo interno*. Le scelte implementative effettuate sono importanti e fanno sì, in particolare, che si riducano i tempi e gli spazi di memoria occupati relativi all'esecuzione dell'algoritmo sia su input diversi che su input uguali. Già confrontando il comportamento dell'algoritmo plus rispetto a quello base si ha evidenza di ciò: sostituendo il lavoro che viene effettuato per elaborare le liste relative all'unione di insiemi per ottenere aggregati di cardinalità ≥ 2 , dispendioso sia per la costruzione della lista stessa che per i confronti che devono essere fatti per realizzarla, con l'assegnazione del risultato

di una somma tra le cardinalità degli stessi insiemi (usando, quindi, solo variabili intere), si ottiene già una velocizzazione nell'individuazione delle partizioni, soprattutto su input di cardinalità più elevata.

Inoltre, sono i parametri in ingresso a definire alcune delle principali caratteristiche del comportamento e dei risultati in uscita degli algoritmi. In particolar modo, la cardinalità di N e la distribuzione di cardinalità degli insiemi che la compongono (riassumibile nell'indice *aggregate index*) influiscono prepotentemente sulle prestazioni dal punto di vista dell'esplorazione degli alberi: il numero di alberi da visitare (anche nei casi in cui si vadano ad esplorare solo la radice e i nodi di primo livello) è $|N|$, e ciò implica una crescita del numero di iterazioni dell'algoritmo per poter lavorare su ciascuno di essi all'aumentare di $|N|$; inoltre, a valori medio-bassi dell'*aggregate index* corrisponde una maggiore probabilità di costruire aggregati tra insiemi di cardinalità medio-bassa, fattore che porta così a visitare gli alberi ancora più in profondità e, di conseguenza, a visitare un maggior numero di nodi. Conseguentemente, anche il tempo di esecuzione dipenderà fortemente da tali aspetti: più alberi presenti, più alberi visitati, più confronti possibili e, quindi, più tempo che intercorre al raggiungimento del termine dell'algoritmo. Analizzare tutti questi aspetti e lavorare sui principali parametri, perciò, rende possibile andare a coprire e, di conseguenza, a comprendere una buona parte dei possibili casi di problema di Exact Cover e il comportamento degli algoritmi implementati per lo stesso.

La ricerca continua di un compromesso tra le prestazioni temporali e quelle spaziali ha portato quindi a risultati che si possono ritenere soddisfacenti. Tuttavia, è sempre aperta la possibilità di sviluppi futuri, di variazioni del codice, di nuove tipologie di strutture dati, di metodi e librerie che possano permettere lo sviluppo di versioni alternative (e, volendo, più efficienti rispetto a quanto prodotto in questo elaborato) dei risolutori del problema di copertura esatta di un insieme.