



UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale
in Ingegneria Informatica

Algoritmi e Strutture Dati
Elaborato a.a. 2023/2024

Calcolo dei Minimal Hitting Set

Referente del corso:

Proff.ssa Zanella Marina

Studentesse:

Adinolfi Ester, matricola 723378
Facchetti Nicole, matricola 731029

Anno Accademico 2024/2025

Indice

1	Introduzione	1
1.1	Il problema dei Minimal Hitting Set	1
1.1.1	Esempio	1
1.2	Specifiche del progetto	2
1.3	Strategia implementativa	2
2	Algoritmi Implementati	4
2.1	Spazio di ricerca e rappresentazione delle ipotesi	4
2.2	Algoritmo seriale (<code>mhs_solver.py</code>)	5
2.2.1	Flusso di esecuzione	5
2.2.2	Strutture dati	8
2.2.3	Monitoraggio memoria	8
2.3	Algoritmo parallelo (<code>mhs_solver_parallel.py</code>)	9
2.3.1	Ordinamento canonico per generazione successori	9
2.3.2	Architettura master-worker	9
	Strutture dati globali	11
2.3.3	Batching delle ipotesi	13
1.	Batch a livello master	13
2.	Batch a livello worker: sub-batching	13
2.3.4	Strategie di deduplicazione adattive	14
1.	Bitset	14
2.	Ordinamento	15
3.	Partizionamento distribuito	15
2.3.5	Monitoraggio memoria e garbage collection	17
2.4	Gestione timeout e interruzioni	18
2.4.1	Modalità di interruzione	18
2.4.2	Salvataggio stato parziale	18
	Implementazione nella versione seriale	19
	Implementazione nella versione parallela	19
3	Interfaccia e parametri	20
3.1	Architettura software	20
3.2	Pre-processamento: rimozione colonne vuote	21
3.3	Esecuzione su singola matrice (<code>run.py</code>)	21
3.3.1	Esempi di utilizzo	22
3.4	Esecuzione batch automatizzata (<code>setup.py</code>)	23
3.4.1	Esempi di utilizzo	24
3.4.2	Gestione errori e robustezza	25

3.5	Script di utilità e manutenzione	25
3.5.1	Pulizia file mal posizionati (<code>cleanup_misplaced.py</code>)	25
	Esempi di utilizzo	26
3.5.2	Riprocessamento matrici mancanti (<code>reprocess_missing.py</code>) . .	26
	Esempi di utilizzo	27
3.6	Menu interattivo (<code>menu.py</code>)	27
3.7	Formato file di input e output	28
3.7.1	File di input (<code>.matrix</code>)	28
3.7.2	File di output (<code>.mhs</code>)	28
4	Sperimentazione e risultati	31
4.1	Matrici di test e categorizzazione	31
4.1.1	Criteri di categorizzazione	31
4.1.2	Distribuzione delle istanze	32
4.2	Ambiente di test e metriche	32
4.2.1	Hardware e software	32
4.2.2	Configurazione dei parametri	32
4.2.3	Metriche raccolte	33
4.3	Obiettivi degli esperimenti	33
4.4	Risultati ottenuti	34
4.4.1	Sommario delle metriche per categoria	34
4.4.2	Modalità automatica	35
4.5	Analisi dell'impatto della densità della matrice	38
4.5.1	Approccio 1: solo matrici completate	40
4.5.2	Approccio 2: tutte le matrici	42
4.5.3	Confronto tra i due approcci	44
4.6	Confronto tra versione seriale e versione automatica	45
4.6.1	Analisi del tasso di completamento	45
4.6.2	Numero di MHS trovati: differenza critica	46
4.6.3	Prestazioni temporali	46
4.6.4	Consumo di memoria	46
4.6.5	Analisi per categoria	47
4.6.6	Grafici di confronto	47
4.6.7	Giustificazione della scelta seriale per matrici piccole	50
5	Conclusioni	52

Elenco delle figure

4.1	Percentuale di completamento per categoria.	35
4.2	Tempo totale per categoria (secondi).	36
4.3	Numero di MHS trovati in funzione del numero di colonne ridotte.	37
4.4	Statistiche di memoria (RSS e picco) per categoria.	38
4.5	Correlazione tra densità della matrice e tempo reale di esecuzione.	40
4.6	Correlazione tra densità della matrice e memoria RSS totale.	41
4.7	Correlazione tra densità della matrice e numero totale di MHS trovati.	42
4.8	Correlazione densità vs tempo (tutte le 43 matrici).	43
4.9	Correlazione densità vs memoria RSS (tutte le 43 matrici).	43
4.10	Correlazione densità vs MHS trovati (tutte le 43 matrici).	44
4.11	Scatter plot del tempo di esecuzione per ogni matrice.	48
4.12	Scatter plot del numero di MHS trovati per ogni matrice.	48
4.13	Scatter plot della memoria RSS utilizzata per ogni matrice.	49
4.14	Scatter plot del picco di memoria per ogni matrice.	50

Elenco delle tabelle

4.1	Metriche principali per categoria nella modalità automatica.	34
4.2	Correlazioni densità-prestazioni (solo 29 matrici completate).	40
4.3	Correlazioni densità-prestazioni (tutte le 43 matrici).	42
4.4	Confronto correlazioni tra approccio 1 (solo completate, 29) e approccio 2 (tutte, 43).	44
4.5	Confronto prestazioni tra versione seriale forzata e versione automatica.	45
4.6	Confronto prestazioni per categoria di matrice.	47
4.7	Confronto prestazioni tra modalità seriale e parallela forzata sulle categorie piccole.	51

Capitolo 1

Introduzione

1.1 Il problema dei Minimal Hitting Set

Il progetto richiede l'implementazione di un algoritmo per il calcolo dei Minimal Hitting Set (MHS) di una matrice rappresentata in modo binario. Data una matrice $Mat \in \{0, 1\}^{N \times M}$, dove le N righe rappresentano una collezione finita di insiemi finiti i cui elementi appartengono al dominio $\{1, \dots, M\}$, e le M colonne rappresentano gli elementi di tale dominio, un *hitting set* è un sottoinsieme di colonne $H \subseteq \{1, \dots, M\}$ tale che ogni riga contenga almeno un 1 nelle colonne selezionate. Formalmente:

$$\forall i \in \{1, \dots, N\} : \exists j \in H \text{ tale che } Mat_{i,j} = 1.$$

Un hitting set si dice *minimale* (MHS) se nessun suo sottoinsieme proprio è ancora un hitting set:

$$\nexists H' \subset H : H' \text{ è un hitting set.}$$

L'obiettivo è enumerare *tutti* i Minimal Hitting Set della matrice di input.

1.1.1 Esempio

Consideriamo la seguente matrice 3×4 :

$$Mat = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Alcuni hitting set validi sono:

- $H = \{3\}$: copre da sola tutte e tre le righe \rightarrow è un MHS;
- $H = \{1, 2, 4\}$: copre tutte le righe e tutte le colonne sono necessarie \rightarrow è un MHS;
- $H = \{1, 2, 3\}$: copre le righe, ma per farlo basterebbe solo la colonna 3 \rightarrow è un hitting set, ma non è minimale.

I Minimal Hitting Set di questa matrice sono: $\{3\}$ e $\{1, 2, 4\}$.

1.2 Specifiche del progetto

Il progetto richiede l'implementazione di un algoritmo che:

1. Legga in input una matrice binaria da un file in formato testuale (file `.matrix`);
2. Calcoli tutti i Minimal Hitting Set della matrice;
3. Producia in output un file `.mhs` contenente:
 - La lista di tutti gli MHS trovati;
 - Statistiche sull'esecuzione (tempo, memoria, numero di ipotesi esplorate).

Il problema è computazionalmente difficile: il numero di MHS può crescere esponenzialmente con la dimensione della matrice e l'esplorazione esaustiva dello spazio di ricerca richiede tecniche di ottimizzazione per gestire istanze di grandi dimensioni. Dal punto di vista della teoria della complessità:

- Il problema di **decisione** (verificare se esiste un hitting set di cardinalità $\leq k$) è NP-completo;
- Il problema di ottimizzazione relativo al calcolo degli HS di cardinalità minima è una riformulazione equivalente del problema Set Covering, che è NP-hard;
- Nel caso peggiore, il numero di MHS può essere **esponenziale** rispetto alla dimensione dell'input e può crescere con N o M ;
- La **difficoltà** intrinseca non risiede solo nell'enumerazione delle soluzioni finali, ma anche nella generazione e nel filtraggio dei candidati intermedi. Lo spazio delle potenziali soluzioni, rappresentato da tutti i possibili sottoinsiemi delle colonne, ha dimensione 2^M . Anche con tecniche di pruning aggressive, il numero di candidati esplorati può rimanere molto elevato.

1.3 Strategia implementativa

Per affrontare il problema abbiamo implementato due versioni dell'algoritmo:

- **Versione seriale:** implementa un'esplorazione BFS (Breadth-First Search) dello spazio delle ipotesi, ottimizzata per matrici di piccole dimensioni. Utilizza:
 - Rappresentazione binaria compatta delle ipotesi con operazioni bitwise efficienti;
 - Ordinamento canonico per valore binario decrescente (regola di precedenza);
 - Generazione univoca delle ipotesi tramite partizionamento succL dei successori immediati;
 - Strutture dati semplici con basso overhead computazionale.
- **Versione parallela:** estende la versione seriale con tecniche di parallelizzazione multi-processo, con particolare attenzione alla deduplicazione (opzionale) e al monitoraggio della memoria (opzionale, attivabile con il flag

--memory-monitoring per evitare overhead in esecuzioni standard), il tutto mantenendo le stesse garanzie di unicità della generazione. È progettata per scalare su matrici di grandi dimensioni sfruttando un'architettura master-worker.

La scelta tra le due versioni avviene automaticamente in base alle dimensioni della matrice di input oppure può essere forzata manualmente dall'utente.

Capitolo 2

Algoritmi Implementati

In questo capitolo vengono descritti i due algoritmi implementati per il calcolo dei Minimal Hitting Set: la versione seriale e la versione parallela. Entrambi seguono un approccio BFS (Breadth-First Search) per esplorare lo spazio delle ipotesi, ma differiscono nelle strutture dati utilizzate e nelle ottimizzazioni adottate per gestire matrici di diverse dimensioni.

2.1 Spazio di ricerca e rappresentazione delle ipotesi

Lo spazio di ricerca è costituito da tutti i possibili sottoinsiemi di colonne della matrice: quindi, date M' colonne non vuote dopo la riduzione, lo spazio ha dimensione $2^{M'}$.

Ogni sottoinsieme viene chiamato *ipotesi* e rappresenta un potenziale hitting set.

Per gestire efficientemente questa esplorazione, ogni ipotesi è rappresentata mediante una struttura compatta che utilizza operazioni bitwize. La classe `Hypothesis` definita in `utility.py` contiene quattro campi:

- `bin` (intero): codifica le colonne presenti nell'ipotesi tramite un bitmask. Il bit in posizione i è 1 se la colonna i -esima appartiene all'ipotesi. Ad esempio, se l'ipotesi contiene le colonne $\{0, 2, 4\}$ in una matrice con 5 colonne, allora `bin` = 10101_2 = 21_{10} . È stato scelto il tipo `int` poiché supporta operazioni bitwize efficienti (come OR e shift) e può rappresentare bitmask di qualsiasi lunghezza grazie alla sua natura arbitrariamente grande, evitando limiti di dimensione. Nell'esempio, la rappresentazione binaria (10101_2) mostra visivamente il bitmask, mentre quella decimale (21_{10}) corrisponde al valore intero utilizzato direttamente nel codice Python;
- `vector` (intero): bitmask che indica quali righe sono coperte dall'ipotesi. Il bit in posizione i è 1 se almeno una colonna dell'ipotesi contiene un 1 nella riga i -esima. Anche in questo caso, è stata scelta la rappresentazione intera per efficienza e flessibilità. Ad esempio, nell'ipotesi $\{0, 2, 4\}$, se le colonne coprono le righe 0, 1 e 3 su 4 righe totali, allora `vector` = 1101_2 = 13_{10} ;
- `card` (intero): cardinalità dell'ipotesi, cioè il numero di colonne presenti. Viene precomputato tramite `bin.bit_count()` per evitare calcoli ripetuti. Anche in

questo caso, è stata scelta la rappresentazione intera per efficienza e flessibilità. Ad esempio, per l'ipotesi $\{0, 2, 4\}$, `card` = 3;

- `num_cols` (intero): numero totale di colonne nella matrice ridotta, necessario per alcune operazioni di manipolazione e, insieme al numero di righe N , come upper bound nell'esplorazione dei livelli di ipotesi (`max_level` = $\max(N, \text{num_cols})$). Anche in questo caso, è stata scelta la rappresentazione intera per efficienza e flessibilità. Ad esempio, nella matrice con 5 colonne, `num_cols` = 5.

Questa rappresentazione permette operazioni molto efficienti:

- **Verifica di copertura:** controllare in tempo costante se un'ipotesi copre tutte le righe significa verificare se `vector` = $2^N - 1$, con N numero di righe. Questa condizione corrisponde ad avere tutti i bit di `vector` a 1;
- **Estensione di un'ipotesi:** secondo il teorema succL, un'ipotesi viene estesa aumentando la cardinalità di 1, cioè il successore aggiunge una nuova colonna i all'insieme dell'ipotesi genitore. La colonna i è scelta solo se corrisponde a un bit che passa da 0 a 1 e questo bit deve essere a sinistra del bit più significativo dell'ipotesi corrente (operazione di shift). Per fare ciò, si aggiorna `bin` con `bin' = bin | (1 << i)` (shift) e `vector` con `vector' = vector | col_vectors[i]` (estensione dell'ipotesi).
- **Confronto e ordinamento:** immediato, confrontando i campi interi.

2.2 Algoritmo seriale (`mhs_solver.py`)

L'algoritmo seriale è progettato per matrici di piccole e medie dimensioni. Segue un approccio BFS classico, esplorando le ipotesi livello per livello in ordine di cardinalità crescente.

2.2.1 Flusso di esecuzione

L'algoritmo si articola nelle seguenti fasi:

1. Parsing e pre-processamento:

- Lettura del file `.matrix`;
- Rimozione delle colonne completamente vuote. Ciò è opzionale: anche se sconsigliato, si può evitare con l'opzione `--no-reduction`. Questa opzione è disponibile anche nel menu interattivo nelle impostazioni avanzate (Capitolo 3);
- Costruzione del vettore `col_map` che mappa gli indici ridotti agli originali;
- Conversione di ogni colonna in un bitmask intero tramite il metodo `get_column_vectors()`;
- Determinazione del **livello massimo di esplorazione**: viene calcolato come $\max(N, M')$, dove N è il numero di righe e M' è il numero di colonne non-vuote. Questo limite è indipendente dall'opzione `--no-reduction`:

anche se l'utente sceglie di mantenere le colonne vuote nella matrice elaborata, il limite di esplorazione si basa solo sulle colonne che effettivamente contribuiscono agli MHS, cioè quelle non-vuote, poiché le colonne vuote non possono mai far parte di una soluzione.

2. Inizializzazione:

- Creazione della variabile globale `last_saved_state`. Questa salva periodicamente lo stato corrente dell'esplorazione (MHS trovati, livello raggiunto, statistiche per livello e distribuzione MHS per cardinalità) per consentire il recupero in caso di interruzioni impreviste (timeout, interruzione utente o esaurimento memoria). È una tupla aggiornata a ogni livello BFS completato, garantendo la possibilità di scrivere un file di output parziale consistente anche dopo un'interruzione;
- Creazione dell'ipotesi vuota (livello 0) con `bin=0`, `vector=0`, `card=0`;
- Inizializzazione delle strutture dati:
 - `found_mhs`: lista degli MHS trovati. Una lista semplice permette di accumulare gli MHS in ordine di scoperta, facilitando l'output finale e l'accesso sequenziale. È efficiente per aggiunte frequenti e iterazioni, senza bisogno di ricerche rapide (che non sono richieste qui);
 - `found_mhs_sets`: set di `frozenset` per verifiche rapide di minimalità. Un set di `frozenset` è ideale per verifiche rapide di contenimento ($O(1)$ medio), necessarie per controllare la minimalità (ad esempio, se un nuovo MHS è già contenuto in uno esistente). I `frozenset`, rappresentando insiemi immutabili non ordinati di oggetti unici, evitano modifiche accidentali e supportano operazioni di insieme efficienti. Si sono preferiti, quindi, a strutture dati come le `tuple`, che pur essendo hashabili e immutabili rappresentano sequenze ordinate, e alle `liste`, che non sono hashabili, non possono essere usate direttamente in un `set` e permettono duplicati. I `frozenset` sono più adatti per rappresentare insiemi di colonne in un MHS, dove l'ordine non conta (essendo ogni colonna rappresentata da un numero intero univoco) e non sono ammessi duplicati;
 - `stats_per_level`: dizionario per statistiche di esplorazione. Un dizionario chiave-valore permette aggiornamenti rapidi e accessi per livello. È adatto a raccogliere metriche, come il numero di ipotesi esplorate. È flessibile e non richiede un ordine specifico;
 - `mhs_per_level`: dizionario che traccia il numero di MHS trovati per ogni cardinalità. Quando un'ipotesi viene identificata come MHS, la sua cardinalità effettiva (campo `card`) viene utilizzata come chiave per incrementare il contatore corrispondente. Questo permette di calcolare la cardinalità minima e massima degli MHS nel file di output.
- Calcolo di `all_rows_mask` = $2^N - 1$ per verifiche di copertura. Questo bitmask pre-calcolato rappresenta tutte le righe coperte (tutti i bit a 1), abilitando verifiche di copertura istantanee con confronti bitwise (con complessità temporale pari a $O(1)$) tra `vector` e `all_rows_mask`,

invece di iterare sulle righe (complessità temporale pari a $O(N)$).

Ad esempio, se $N = 4$, `all_rows_mask` = 15_{10} = 1111_2 . Se `vector` = 13_{10} = 1101_2 , le righe $\{0, 2, 3\}$ sono coperte (bit 0, 2 e 3 sono impostati a 1), ma la riga 1 non lo è (bit 1 è 0), quindi l'ipotesi non è completa. Se invece `vector` = 15_{10} = 1111_2 , tutte le righe sono coperte e l'ipotesi è una soluzione valida.

3. Esplorazione BFS:

Per ogni livello k (da 0 fino al livello massimo calcolato, cioè fino a $\max(N, M')$ dove M' è il numero di colonne non-vuote):

- Le ipotesi del livello corrente (provenienti dal livello $k - 1$, già ordinate canonicamente) vengono processate in ordine;
- Per ogni ipotesi H del livello corrente (in ordine):
 - (a) **Verifica copertura:** se `vector` = `all_rows_mask`, allora H copre tutte le righe ed è un MHS;
 - (b) **Aggiunta MHS:** se H è una soluzione (copre tutte le righe), viene aggiunta direttamente a `found_mhs` e `found_mhs_sets` dopo una conversione degli indici da ridotti a originali tramite `col_map`. Infine, l'MHS viene registrato nel dizionario `mhs_per_level` usando la sua cardinalità effettiva (`card`) come chiave.

La **minimalità è garantita per costruzione** dall'algoritmo BFS combinato con la potatura, senza necessità di controlli esplicativi:

- **Livello 1 (singeletti):** ogni MHS è automaticamente minimale (cardinalità minima possibile);
- **Livelli $k \geq 2$:** la minimalità è garantita da:
 - * BFS esplora per cardinalità crescente: ogni MHS di cardinalità k viene trovato prima di qualsiasi MHS di cardinalità $k' > k$;
 - * Gli MHS non generano figli (condizione “`if not is_mhs`” nella generazione successori);
 - * La potatura durante la generazione figli scarta automaticamente ipotesi contenenti MHS come sottoinsiemi;
 - * Il teorema `succL` garantisce che ogni ipotesi sia generata esattamente una volta dal suo predecessore più a destra.

Quindi, se un'ipotesi H del livello $k \geq 2$ è soluzione, è **certamente minimale**: non può contenere MHS più piccoli (con cardinalità $< k$) perché la potatura avrebbe impedito la sua creazione.

- (c) **Generazione figli:** genera i successori immediati $\text{succL}(H)$ solo se H non è MHS (come da specifica, per potare lo spazio di ricerca). La generazione implementa il teorema `succL`: per un'ipotesi H , i suoi successori H' appartenenti a $\text{succL}(H)$ vengono generati complementando (impostando a 1) i soli bit 0 che si trovano a sinistra del bit 1 più significativo di H .

Questo approccio, combinato con l'ordinamento canonico del livello, garantisce che ogni ipotesi figlia venga generata esattamente una volta dal suo unico predecessore "più a destra" (quello con `bin(h)` più piccolo).

- **Ordinamento canonico del livello successivo:** le ipotesi generate vengono ordinate per `bin` decrescente ($\text{bin}(h_1) > \text{bin}(h_2)$) per garantire la regola di precedenza `succL`, poi si passa al livello successivo;
- L'esplorazione **termina** quando non ci sono più ipotesi da esplorare.

4. Finalizzazione:

- Ordinamento degli MHS trovati per cardinalità crescente per una presentazione ordinata dei risultati;
- Calcolo delle prestazioni di esecuzione (tempo CPU e wall-clock) tramite la funzione `measure_performance()`;
- Scrittura del file `.mhs` con i risultati ottenuti, statistiche per livello, distribuzione MHS per cardinalità e metadati (origine, densità, categoria) tramite la funzione `write_mhs_output()`;
- Stampe di riepilogo a console: percorso del file di output, numero totale di MHS trovati, tempo di esecuzione totale e livello BFS raggiunto;
- Se presenti MHS, visualizzazione della distribuzione per cardinalità (numero di MHS per ogni livello di cardinalità);
- Indicazione dello stato di completamento dell'algoritmo (completato o interrotto).

2.2.2 Strutture dati

L'algoritmo seriale si basa su strutture dati efficienti e scalabili, progettate per gestire l'esplorazione BFS senza eccessivo overhead di memoria o tempo. Oltre alle strutture principali già introdotte nella fase di inizializzazione (`found_mhs`, `found_mhs_sets` e `stats_per_level`), l'implementazione utilizza delle **liste** per ogni livello: `current_level_hypotheses` contiene le ipotesi del livello attualmente in elaborazione, mentre `next_level_hypotheses` accumula i successori generati per il livello successivo. Il riferimento a `next_level_hypotheses` viene assegnato a `current_level_hypotheses` alla fine di ogni iterazione (*passaggio per riferimento*), permettendo un passaggio efficiente tra livelli senza copie aggiuntive. Questa scelta sfrutta la mutabilità delle liste per un'implementazione semplice e performante, evitando strutture più complesse come code dedicate che introduirebbero overhead inutile.

2.2.3 Monitoraggio memoria

A differenza della versione parallela, dove il monitoraggio del picco di memoria con `tracemalloc` è opzionale per motivi di prestazioni, nella versione seriale `tracemalloc` è **sempre attivo** per semplicità implementativa. Questo garantisce che il picco di memoria sia sempre disponibile nei file di output (`.mhs` e JSON), fornendo metriche complete senza richiedere configurazioni aggiuntive dall'utente.

L'opzione `--memory-monitoring`, quindi, non è supportata in modalità seriale. L'overhead introdotto da `tracemalloc` è accettabile poiché la versione seriale è progettata per istanze di dimensioni più piccole, dove le prestazioni non sono critiche come nella versione parallela. Proprio per questo motivo, la versione seriale non monitora continuamente l'andamento della memoria per interruzioni basate su superamento di soglie. L'algoritmo continua l'elaborazione fino a quando il sistema operativo non solleva un'eccezione `MemoryError`, che viene catturata per salvare i risultati parziali accumulati. Questo approccio **reattivo** è possibile grazie alla semplicità del singolo processo/thread, che permette di interrompere e salvare in modo affidabile senza rischi di deadlock o stati inconsistenti.

2.3 Algoritmo parallelo (`mhs_solver_parallel.py`)

L'algoritmo parallelo mantiene la stessa logica BFS della versione seriale, ma introduce ottimizzazioni e tecniche di parallelizzazione per scalare su matrici grandi e sfruttare architetture multi-core.

2.3.1 Ordinamento canonico per generazione successori

L'algoritmo parallelo, come quello seriale, richiede un **ordinamento canonico** delle ipotesi per garantire la correttezza del teorema di generazione `succL`. Questo ordinamento assicura che ogni ipotesi figlia venga generata esattamente una volta dal suo predecessore più a destra.

L'ordinamento viene effettuato una sola volta per livello, alla fine (se richiesta, immediatamente dopo la deduplicazione globale dei risultati ottenuti dai worker). Questo serve a garantire l'ordine nel caso in cui la sincronizzazione dei worker non avvenga in modo perfetto. L'ordinamento segue il criterio `bin(h1) > bin(h2)` in ordine decrescente (`current_level.sort(key=lambda x: x[0], reverse=True)`), garantendo che l'ordine canonico sia rispettato globalmente anche in ambiente parallelo e che ogni worker possa applicare correttamente il teorema `succL` senza violare le condizioni di unicità.

2.3.2 Architettura master-worker

In un algoritmo parallelo non si può semplicemente far girare tutto in un singolo processo, perché Python ha il **GIL (Global Interpreter Lock)** che limita la vera parallelizzazione. Quindi, è necessario usare il modulo `multiprocessing` per creare processi separati, ognuno con memoria propria, invece di `threading` (limitato dal GIL per operazioni CPU-bound come le nostre) o `asyncio` (ottimizzato per I/O-bound, ma non per operazioni CPU-intensive).

L'architettura master-worker divide i compiti:

- **Processo master:** coordina l'esplorazione, mantiene lo stato globale (MHS trovati, strutture per pruning), e distribuisce il lavoro ai worker.

Il master:

- **Inizializza tutto:** crea il pool di worker, imposta strutture dati globali (come `found_mhs`, `found_mhs_sets`, `mhs_per_level`), e prepara l’ipotesi iniziale;
- **Coordina l’esplorazione livello per livello:** per ogni livello k BFS:
 - * Prende le ipotesi del livello corrente (`current_level`);
 - * **Identificazione MHS:** itera sulle ipotesi di `current_level` per identificare quali sono MHS (verificando se `vector == (1 << num_rows) - 1`). Gli MHS trovati vengono aggiunti a `found_mhs` e rimossi da `current_level`. Questa fase implementa la specifica `succL`: le ipotesi MHS non generano figli poiché qualsiasi loro estensione non sarebbe minimale. Rimuovendole da `current_level`, solo le ipotesi non-MHS vengono inviate ai worker per la generazione dei successori;
 - * Suddivide le ipotesi in batch adattivi (la dimensione minima, configurabile, è 1000) bilanciati tra i processi disponibili;
 - * Distribuisce i batch ai worker tramite code thread-safe del modulo `multiprocessing`;
 - * Raccoglie i risultati dai worker e aggiorna lo stato globale (nuove ipotesi e statistiche);
 - * Se richiesta, applica deduplicazione per rimuovere eventuali duplicati generati dai worker paralleli (Sezione 2.3.4);
 - * Ordina le ipotesi canonicamente (per `bin` decrescente) per rispettare il teorema `succL` al livello successivo, garantendo che ogni ipotesi figlia venga generata esattamente una volta anche in ambiente parallelo.
- **Gestisce timeout e interruzioni:** monitora il tempo rimanente e risponde a richieste di interruzione (Sezione 2.4);
- **Finalizza:** come nella versione seriale, ordina gli MHS per cardinalità, scrive il file `.mhs` di output, stampa il riepilogo a console, mostra la distribuzione per cardinalità degli MHS trovati e indica lo stato di completamento dell’algoritmo.
- **Pool di worker:** un pool di processi (tipicamente pari al numero di core CPU – 1) che elaborano in modo indipendente e in parallelo i batch di ipotesi. Ogni worker:
 - **Riceve un batch** dal master (lista di valori `bin`) tramite code thread-safe;
 - **Elabora il batch** suddividendolo in sub-batch da 250 ipotesi per responsività alle interruzioni. Per ciascuna ipotesi nei sub-batch:
 1. **Ricostruzione campi:** ricalcola `vector` e `card` dal valore `bin` utilizzando i vettori colonna pre-caricati (`GLOBAL_COL_VECTORS`);
 2. **Generazione successori:** genera i figli delle ipotesi non-MHS tramite `generate_succ_left()` (definita in `utility.py`) applicando il teorema `succL`. L’algoritmo `succL` garantisce che ogni ipotesi sia

generata esattamente una volta dal suo predecessore più a destra, quindi ognuna è unica per costruzione e viene raccolta direttamente nella lista `children_tuples` senza necessità di controlli di duplicati;

3. **Controlli frequenti:** verifica timeout e interruzioni tra i sub-batch per terminazioni rapide con risultati parziali.

- **Restituisce al master i risultati:** invia al master una tupla contenente:

- * `children_tuples`: lista di nuove ipotesi figlie (tuple `(bin, vector, card)`) che verranno elaborate nel livello BFS successivo;
- * `timeout_reached`: flag booleano che indica se il worker si è fermato per timeout.

I worker sono inizializzati una volta sola con `worker_initializer`, che copia variabili globali (come `GLOBAL_COL_VECTORS`, `GLOBAL_COL_MAP`, Sottosezione 2.3.2) per evitare serializzazioni ripetute di dati grandi. Questo rende tutto più veloce.

Ogni worker lavora in isolamento, senza condividere memoria con gli altri o il master (tranne tramite code).

La comunicazione avviene tramite code thread-safe di `multiprocessing`:

- **Coda di input:** il processo master inserisce i batch di ipotesi nella coda di input; i worker prelevano i batch per l'elaborazione parallela;
- **Coda di output:** i worker depositano nella coda di output i risultati elaborati (ad es. `children_tuples`, flag di timeout, statistiche); il master legge e aggrega tali risultati;
- **Sincronizzazione dei livelli:** il master attende la terminazione di tutti i worker relativi al livello corrente prima di procedere al livello BFS successivo, garantendo la consistenza dello stato globale e l'applicazione corretta delle operazioni di deduplicazione e aggiornamento degli MHS.

Grazie a questo modello non c'è concorrenza sui dati globali.

Strutture dati globali

Precedentemente abbiamo citato delle strutture dati globali condivise tra master e worker. Queste sono:

- `GLOBAL_COL_VECTORS`: lista dei vettori colonna della matrice ridotta, dove ogni vettore rappresenta le righe coperte da una colonna specifica. Questa struttura è essenziale per il calcolo del vettore di copertura `vector` delle ipotesi nei worker, poiché permette di ricostruire rapidamente il vettore bitwise tramite operazioni OR per ogni colonna presente nell'ipotesi.

Si è optato per una lista Python per la sua semplicità, facilità di serializzazione con pickle (conversione in byte effettuata automaticamente dalle operazioni del modulo `multiprocessing` per avere comunicazione di dati dal master ai worker o viceversa), per l'accesso in tempo costante $O(1)$ e per la sua flessibilità. Strutture alternative come array NumPy introdurrebbero overhead per operazioni non vettoriali, mentre `array.array` non offre benefici significativi per dati interi;

- **GLOBAL_COL_MAP**: dizionario che mappa gli indici originali delle colonne (della matrice completa) agli indici nella matrice ridotta. Questa struttura è necessaria per la corretta interpretazione dei risultati finali, poiché permette di risalire alle colonne originali quando si scrivono gli MHS nel file di output. Si è scelto un dizionario Python per il lookup efficiente in tempo costante $O(1)$. Le liste, infatti, richiederebbero scansioni lineari $O(n)$, mentre le tuple non supportano lookup diretto;
- **GLOBAL_NUM_ROWS**: intero che rappresenta il numero di righe della matrice. È utilizzato per verificare la copertura completa e per calcoli di upper bound nell'esplorazione BFS;
- **GLOBAL_NUM_COLS**: intero che rappresenta il numero di colonne nella matrice ridotta. È utilizzato per operazioni di manipolazione bitwise e come limite superiore nell'esplorazione (`max_level=max(num_rows, num_cols)`);
- **GLOBAL_STOP_EVENT**: oggetto `multiprocessing.Event` condiviso tra master e worker per segnalare interruzioni coordinate. Permette al master di notificare tutti i worker di terminare immediatamente in caso di timeout, interruzione utente o superamento limite memoria. Si è scelto `multiprocessing.Event` per la sua semplicità e sicurezza nelle segnalazioni thread-safe tra processi, senza rischio di race conditions. Variabili condivise richiederebbero sincronizzazione manuale, aumentando la complessità.

Queste strutture dati sono inizializzate una volta sola nel master e copiate in memoria locale per ciascun processo worker tramite la funzione `worker_initializer`, chiamata automaticamente dal `multiprocessing.Pool` all'avvio di ogni processo. Poiché fare riferimento alle variabili globali, che sono immutabili e read-only, richiederebbe una continua serializzazione e deserializzazione tramite `pickle`, con conseguente spreco di tempo, si è deciso di evitarlo facendone una copia alla creazione del singolo worker. In questo modo si sacrifica memoria aggiuntiva per guadagni significativi in velocità e semplicità del processo, eliminando problemi di concorrenza senza trasmissione ripetuta.

Nel master, invece, sono presenti strutture dati aggiuntive per il coordinamento globale:

- **found_mhs**: lista dei Minimal Hitting Set trovati, accumulati in ordine di scoperta;
- **found_mhs_sets**: set di frozenset per verifica rapida di dominanza (pruning) rispetto agli MHS già trovati;
- **stats_per_level**: dizionario con statistiche per livello BFS (ipotesi esplorate, generate, deduplicate);
- **mhs_per_level**: dizionario che traccia il numero di MHS trovati per cardinalità.

Queste strutture sono mantenute esclusivamente nel master, poiché richiedono aggiornamenti coordinati e non possono essere distribuite ai worker senza complicare la sincronizzazione.

2.3.3 Batching delle ipotesi

Per ridurre l'overhead di comunicazione tra processi, l'algoritmo parallelo organizza le ipotesi in una gerarchia di batch su due livelli: batch principali distribuiti ai worker e sub-batch interni ai worker per maggiore responsività.

1. Batch a livello master

Il master suddivide le ipotesi del livello corrente in batch per la distribuzione ai worker. La dimensione minima è configurabile dall'utente (default: 1000 ipotesi, parametro `--batch-size`), ma quella effettiva viene calcolata adattativamente secondo la formula:

$$\text{batch_size_local} = \max \left(\text{batch_size}, \frac{\text{num_hyp}}{\text{num_processes} \cdot \text{adaptive_factor}} \right)$$

dove `batch_size` è la soglia minima configurabile, e

$$\text{adaptive_factor} = \min \left(3, 1 + \frac{\text{level}}{3} \right)$$

cresce gradualmente con il livello BFS per gestire l'esplosione combinatoria. Questa formula bilancia comunicazione e parallelizzazione: divide le ipotesi per processi e fattore adattivo per distribuire il carico, mentre farne il massimo garantisce una dimensione minima. Tuttavia, per insiemi di ipotesi molto piccoli (quando `num_hyp` < `batch_size_local`), viene creato un singolo batch contenente tutte le ipotesi per evitare overhead inutile di comunicazione. Il limite di 3 su `adaptive_factor` è stato scelto arbitrariamente come valore iniziale, senza una dimostrazione formale. Valori *inferiori* potrebbero causare batch troppo piccoli, aumentando l'overhead di comunicazione, mentre valori *superiori* potrebbero generare batch troppo grandi, portando a squilibri di carico. Un possibile sviluppo futuro potrebbe essere esplorare valori ottimali attraverso test su istanze diverse, per affinare il bilanciamento tra efficienza e scalabilità.

Infine, il numero di batch è calcolato come un *multiplo* del numero di processi per ottimizzare il bilanciamento del carico, tranne nei casi di insiemi molto piccoli dove viene creato un singolo batch. La distribuzione avviene tramite il meccanismo di **load balancing dinamico** del **multiprocessing.Pool**: non appena un processo worker si libera, preleva il batch successivo disponibile da una coda. Avere un numero di batch multiplo a quello dei processi permette al Pool di bilanciare il carico in modo efficace, assicurando che i worker più veloci elaborino più batch e minimizzando i tempi di inattività.

Ogni batch è una lista di valori `bin` (interi): questa scelta riduce l'uso di memoria e l'overhead di comunicazione tra processi. Invece di trasferire oggetti `Hypothesis` completi (con campi aggiuntivi come `vector`, `card` e `num_cols`), si inviano solo i valori `bin`, che sono leggeri e facili da serializzare. I worker calcolano `vector` e `card` usando i vettori colonna pre-caricati, con operazioni veloci. Questo ottimizza le prestazioni per istanze grandi, bilanciando risparmio di memoria con un piccolo calcolo aggiuntivo nei worker.

2. Batch a livello worker: sub-batching

Ogni worker riceve un batch dal master (di dimensione adattiva) e lo elabora suddividendolo internamente in sub-batch di 250 ipotesi. Si è scelto 250 in quanto rappre-

senta un quarto del batch minimo. Questa ulteriore granularità serve esclusivamente per **responsività**: tra un sub-batch e l’altro, il worker verifica frequentemente timeout e interruzioni, permettendo terminazioni rapide con salvataggio dei risultati parziali senza dover completare l’intero batch ricevuto dal master.

Il sub-batching non influisce sulla logica dell’algoritmo, ma migliora significativamente la gestione delle interruzioni in scenari con batch grandi.

2.3.4 Strategie di deduplicazione adattive

Una delle sfide principali nella versione parallela è la deduplicazione efficiente delle ipotesi, cioè la rimozione dei valori duplicati.

A livello teorico, ogni worker non dovrebbe poter generare la stessa ipotesi figlia. Infatti, ognuna è generata univocamente solo dal suo predecessore più a destra. Tuttavia, per aumentare la robustezza del sistema, l’implementazione prevede una deduplicazione globale opzionale dopo l’aggregazione dei risultati dai worker.

La deduplicazione globale è attiva di default, ma può essere disabilitata tramite l’opzione `--skip-global-dedup`.

L’implementazione adotta **tre strategie** che si adattano automaticamente alle caratteristiche dell’istanza, cioè al numero di colonne ridotte. Di seguito sono descritte tutte e tre.

1. Bitset

Quando il numero di colonne ridotte è molto piccolo (≤ 24), viene utilizzata una strategia di deduplicazione basata su un **bitset** (array di bit).

Il bitset (`seen_bitset`) è un array di bit inizializzato a 0. La deduplicazione avviene scorrendo tutte le ipotesi generate dall’unione dei risultati dei worker (lista aggregata di tuple `(bin_val, vec, card)`) e per ciascuna si prende il valore `bin_val` (l’intero che rappresenta il bitmask dell’ipotesi) e lo si usa direttamente come indice nell’array `seen_bitset`. Se il bit all’indice `bin_val` è 0, l’ipotesi non è stata vista e viene marcata impostando il bit a 1, aggiungendola alla lista delle uniche; se è 1, è un duplicato e viene scartata. Questo permette operazioni di lookup e inserimento in tempo $O(1)$ esatte, senza falsi positivi o negativi, garantendo massima efficienza per spazi binari piccoli. I campi `vec` (vettore di copertura) e `card` (cardinalità) non servono per la deduplicazione, vengono semplicemente trasferiti alla lista risultante senza ulteriori elaborazioni.

La **motivazione** che sta dietro al **limite 24** riguarda la complessità spaziale. Il bitset richiede esattamente $2^{\text{num_cols}}$ bit di memoria, poiché ogni possibile valore `bin` (da 0 a $2^{\text{num_cols}} - 1$) corrisponde a una posizione nell’array e `bin_val` viene usato direttamente come indice intero. Per `num_cols = 24`, questo corrisponde a $2^{24} = 16.777.216$ bit, equivalenti a circa 2 MB. Questo è gestibile sulla maggior parte dei sistemi. Da questo valore in poi, la memoria necessaria cresce esponenzialmente: per `num_cols = 25` la memoria raddoppierebbe a 4 MB, per `num_cols = 30` raggiungerebbe 1 GB. Ciò renderebbe l’allocazione impraticabile o inefficiente.

La complessità temporale totale di questa deduplicazione è, quindi, $O(n)$, dove n è il numero di ipotesi aggregate.

2. Ordinamento

Per istanze di dimensioni medie ($24 < \text{num_cols} \leq 128$), viene adottata una strategia basata sull'**ordinamento**:

- Le ipotesi generate dai worker vengono prima **aggregate** in una lista unica dal master;
- La lista viene **ordinata** per valore **bin** usando il metodo stabile **sort()** di Python, che usa l'algoritmo Timsort. Questo confronta gli elementi solo tramite relazione $<$. La complessità temporale di questa fase è $O(n \log n)$, dove n è il numero totale di ipotesi generate;
- Dopo l'ordinamento, i duplicati diventano **consecutivi** e possono essere rimossi in un singolo passaggio lineare: l'algoritmo confronta ciascun **bin_val** con **last_bin** (che tiene traccia del valore precedente). Se sono uguali, è un duplicato consecutivo e viene contato in **duplicates_found** e scartato con **continue**. Altrimenti, **last_bin** viene aggiornato e l'elemento viene aggiunto alla lista **unique**. Questa fase ha complessità temporale $O(n)$.

La complessità totale di questa strategia è determinata dall'ordinamento e dalla rimozione dei duplicati, risultando in $O(n \log n)$.

Anche la memoria usata risulta controllata: richiede solo lo spazio per la lista ordinata, senza strutture ausiliarie.

Tuttavia, per istanze molto grandi ($> 10^6$ ipotesi), l'ordinamento può diventare costoso in tempo e memoria, motivando l'uso della strategia distribuita (Sottosezione 2.3.4).

3. Partizionamento distribuito

Quando il numero di colonne ridotte supera 128, lo spazio di ricerca diventa enorme e l'ordinamento impraticabile. Adottiamo una strategia di **partizionamento distribuito**, dividendo lo spazio binario in bucket più piccoli per deduplicazione scalabile.

La logica di funzionamento è la seguente:

- Lo spazio degli interi **bin** è partizionato in P bucket, dove P è il numero di processi paralleli (**num_processes**, tipicamente pari al numero di core CPU-1);
- Ogni ipotesi è assegnata a un bucket in base a $\text{bin} \bmod P$, garantendo una distribuzione uniforme se i valori **bin** sono casuali. Ad esempio, con **num_processes** = 4, un **bin** = 10 andrà al bucket $10 \bmod 4 = 2$;
- Ogni bucket utilizza un set (**seen**) per tracciare i valori **bin** già incontrati: le ipotesi uniche sono aggiunte a **unique**, mentre i duplicati sono scartati;
- Dopo aver processato un bucket, la memoria è liberata con **gc.collect()** per ridurre l'uso di RAM.

Usiamo **liste di liste** per i bucket per semplicità: le liste permettono un accumulo rapido (complessità temporale pari a $O(1)$ per ogni **append**, quindi $O(n)$ totale)

senza overhead di hash. Strutture come set di set introdurrebbero complessità aggiuntiva ($O(1)$ medio per ogni inserimento, ma $O(n)$ nel caso peggiore per collisioni), rallentando il processo su volumi grandi. Inoltre, separare partizionamento (liste) e deduplicazione (set locali) rende il codice più modulare e debuggabile, permettendo controlli di memoria e timeout senza interrompere la logica.

Un’alternativa sarebbe deduplicare direttamente durante il partizionamento, usando set ausiliari per evitare duplicati. Ad esempio:

```

1  bucket_seen = [set() for _ in range(num_processes)]
2  buckets = [[] for _ in range(num_processes)]
3
4  for child in all_children:
5      bin_val = child[0]
6      idx = bin_val % num_processes
7      if bin_val not in bucket_seen[idx]:
8          bucket_seen[idx].add(bin_val)
9          buckets[idx].append(child)

```

Ciò ridurrebbe la memoria accumulata, ma introdurrebbe overhead iniziale per i set. La complessità temporale resterebbe $O(n)$, con un possibile risparmio spaziale in caso di alta duplicazione (scenario raro nel nostro progetto). Abbiamo evitato questa modifica poiché la deduplicazione è solo una misura di sicurezza.

Per quanto riguarda la nostra implementazione, dal punto di vista della complessità *temporale* questa strategia è lineare: $O(n)$, dove n è il numero totale di ipotesi. Infatti, il partizionamento richiede un solo passaggio su tutti gli elementi, mentre ogni bucket viene deduplicato indipendentemente in tempo $O(m)$, con $m \ll n$ pari al numero di ipotesi nel singolo bucket. Rispetto all’ordinamento (Sottosezione 2.3.4), che ha complessità $O(n \log n)$, questo approccio è più veloce quando n diventa molto grande.

Per quanto riguarda lo *spazio*, è importante notare che entrambe le strategie (ordinamento e partizionamento) richiedono $O(n)$ memoria. La differenza sta nel modo in cui questa memoria viene gestita: mentre l’ordinamento lavora su un’unica grande lista, il partizionamento divide i dati in P strutture più piccole (i `set` dei bucket) che vengono processate una alla volta e poi liberate con `gc.collect()`. Questo permette di lavorare con blocchi più piccoli di memoria, evitando problemi con sistemi che hanno poca RAM disponibile o memoria frammentata. D’altra parte, per matrici di dimensioni medie questa suddivisione introduce un overhead (strutture multiple e chiamate ripetute al garbage collector) che può risultare meno efficiente dell’ordinamento. Si ha, quindi, un vantaggio pratico di gestione della memoria: anche se lo spazio totale è $O(n)$, processando i bucket sequenzialmente e deallocandoli si riduce il picco di memoria necessario in un singolo istante, rendendo l’algoritmo più robusto su macchine con memoria frammentata.

Per questo motivo abbiamo riservato questa strategia alle istanze molto grandi, dove sia il vantaggio in velocità che la gestione granulare della memoria risultano necessari.

2.3.5 Monitoraggio memoria e garbage collection

L’implementazione parallela monitora continuamente l’uso di memoria tramite `psutil` e adotta politiche proattive, cioè non aspetta che la memoria sia esaurita per intervenire:

- **Monitoraggio RSS (Resident Set Size):** la RSS è la quantità di memoria fisica (RAM) effettivamente usata dal processo in un dato momento. Questa è controllata periodicamente (ogni 0.2 secondi) tramite `psutil.Process().memory_info().rss`. La RSS rappresenta la memoria finale effettivamente occupata al termine dell’esecuzione ed è la metrica principale riportata nei risultati;
- **Monitoraggio picco con tracemalloc (opzionale):** come accennato per la versione seriale, per tracciare il picco di memoria durante l’esecuzione (massima RAM raggiunta), l’implementazione può utilizzare il modulo `tracemalloc` di Python. Questo, tuttavia, introduce un overhead significativo a causa del tracciamento granulare di ogni allocazione. Per questo motivo, il modulo è stato reso **opzionale** e viene attivato solo se l’utente specifica esplicitamente il flag `--memory-monitoring`: senza questo flag, viene misurata solo la RSS finale, evitando l’overhead e migliorando le prestazioni. Nei file di output (`.mhs` e `JSON`), quando il monitoraggio non è attivo, il campo del picco di memoria riporta “non rilevato”, indicando chiaramente che la metrica non è disponibile. Questa ottimizzazione è stata introdotta dopo aver identificato che il tracciamento continuo penalizzava inutilmente le esecuzioni standard;
- **Soglie di intervento:** quando la memoria RSS supera il 95% (configurabile tramite `--memory-threshold`, con valore possibile compreso tra il 50% e il 99%. Il valore di default nel caso in cui l’utente scelga di monitorare la memoria (`--memory-monitoring`), ma non scelga una soglia, è 95%) della RAM disponibile:
 1. *Garbage collection* forzata con `gc.collect()`, per liberare memoria inutilizzata;
 2. Se ancora critico, *terminazione controllata* con salvataggio risultati parziali.
- **Gestione proattiva dei batch:** controlli frequenti durante elaborazione batch per evitare sovraccarichi improvvisi;
- **Salvataggio di emergenza:** utilizza una struttura dati globale `_emergency_data` (Sezione 2.4.2) per mantenere lo stato corrente (MHS trovati, statistiche, livello). Questa struttura viene aggiornata a ogni iterazione per salvataggi rapidi.

Questo approccio **preventivo** contrasta con quello *reattivo* della versione seriale, dove non vengono impostate soglie e l’interruzione avviene solo su eccezione di sistema sollevata dal sistema operativo. L’intervento proattivo è necessario per gestire la complessità del multiprocessing, dove un’esaurimento improvviso della memoria potrebbe lasciare i worker in stati inconsistenti o causare deadlock durante il tentativo di salvataggio.

2.4 Gestione timeout e interruzioni

Entrambe le implementazioni (seriale e parallela) supportano meccanismi di interruzione controllata e salvataggio dello stato per gestire esecuzioni prolungate o richieste di terminazione anticipata.

2.4.1 Modalità di interruzione

L'esecuzione può essere interrotta in tre modi:

1. **Timeout configurabile**: l'utente può specificare un tempo massimo di esecuzione tramite il parametro `--timeout`. L'algoritmo monitora continuamente il tempo trascorso e si interrompe quando il limite viene raggiunto;
2. **Interruzione da tastiera**: l'utente può terminare l'esecuzione in qualsiasi momento premendo:
 - **Ctrl+C** (segnale `SIGINT`): è un segnale di sistema immediato, catturato dal sistema operativo di Windows e gestito dal programma per terminazioni rapide;
 - Il tasto **q** (quit) o **ESC**: sono input da tastiera controllati attivamente (ogni 20ms) dal programma e in background, cioè il programma verifica periodicamente se l'utente ha premuto questi tasti. Sono utili quando il programma è in esecuzione, ma non bloccato. Essendo un controllo manuale e periodico, potrebbe essere meno immediato. Allo stesso tempo, risulta essere più flessibile proprio perché non dipendente dal sistema operativo.
3. **Limite memoria**: nella versione parallela, se l'uso di memoria supera la soglia configurata (se non configurata, default 100%; altrimenti è configurabile tramite `--memory-threshold`), l'esecuzione viene interrotta per evitare crash del sistema.

2.4.2 Salvataggio stato parziale

In caso di interruzione, entrambe le versioni salvano automaticamente lo stato corrente dell'esplorazione:

- **MHS trovati**: tutti i Minimal Hitting Set identificati fino al momento dell'interruzione;
- **Statistiche per livello**: numero di ipotesi esplorate, generate e nel caso parallelo eventualmente deduplicate per ogni livello BFS;
- **Metadati**: tempo di esecuzione, livello BFS raggiunto, eventuale motivo dell'interruzione (*timeout, utente, memoria*). Il livello riportato è quello dell'*ultima iterazione completata*: se il timeout scatta all'inizio del livello k (prima di processare le sue ipotesi), viene salvato il livello $k - 1$ come ultimo completato.

Il file di output generato include un campo `status` che indica se l'esplorazione è stata completata o interrotta, permettendo di distinguere i risultati completi da quelli parziali.

Implementazione nella versione seriale

La versione seriale utilizza un approccio molto semplice, ma comunque efficace:

- **Salvataggio periodico:** al termine di ogni livello BFS, lo stato corrente viene salvato in una variabile locale `last_saved_state`;
- **Handler del segnale di interruzione:** cattura le interruzioni da tastiera e salva immediatamente lo stato utilizzando `last_saved_state`;
- **Controlli timeout:** verifiche periodiche del tempo trascorso con logica analoga alla versione parallela, ma senza la complessità della coordinazione multiprocesso.

Implementazione nella versione parallela

L'implementazione parallela adotta meccanismi più sofisticati per garantire salvaggi rapidi e consistenti:

- **Struttura dati di emergenza (`_emergency_data`):** un dizionario globale condiviso che mantiene lo stato aggiornato dell'algoritmo, utile per generare un file di output parziale completo e consistente in caso di interruzione. Contiene:
 - **Campi mutabili:** MHS trovati, statistiche per livello, livello corrente. Vengono aggiornati a ogni iterazione tramite la funzione `update_emergency_data()`;
 - **Campi immutabili:** dati di input (`num_cols_original`, `num_rows`, `col_vectors`, `removed_cols`), inizializzati una sola volta tramite la funzione `initialize_emergency_input_data()`.
- **Handler del segnale di interruzione:** cattura le interruzioni da tastiera e invoca immediatamente il salvataggio utilizzando i dati presenti nella struttura dati `_emergency_data`;
- **Controlli timeout frequenti:** verifiche periodiche (ogni 0.05 secondi nel master) con margini di sicurezza adattivi per massimizzare l'utilizzo del tempo disponibile. I margini variano in base al contesto:
 - **Worker:** margine iniziale di 0.05s per permettere l'elaborazione anche con timeout molto brevi (es. 1-2 secondi), controllando successivamente ogni sub-batch;
 - **Master:** margini tra 0.5s e 2.0s per operazioni critiche (deduplicazione) che richiedono tempo aggiuntivo per completamento e cleanup.

I controlli sono stati calibrati per:

- Ridurre gli sprechi di tempo, utilizzando quasi tutto il timeout invece di interrompere con diversi secondi di anticipo;
- Garantire tempo sufficiente per la pulizia (terminazione worker, garbage collection) e il salvataggio prima della scadenza effettiva;
- Permettere l'elaborazione efficace anche con timeout molto brevi (1-2 secondi), evitando che i worker restituiscano risultati vuoti.

Capitolo 3

Interfaccia e parametri

Per facilitare l'utilizzo del programma, è stata sviluppata un'interfaccia testuale interattiva. In questo capitolo vengono descritti i file creati, il loro flusso di utilizzo e tutti i parametri configurabili tramite l'interfaccia.

3.1 Architettura software

Il progetto è organizzato nei seguenti moduli principali:

- `mhs_solver.py`: implementazione dell'algoritmo seriale;
- `mhs_solver_parallel.py`: implementazione dell'algoritmo parallelo;
- `utility.py`: funzioni di supporto (tra cui funzioni di parsing e di gestione I/O);
- `run.py`: wrapper per l'esecuzione del programma su una singola matrice;
- `matrices_selection.py`: selezione automatica delle matrici di test;
- `setup.py`: script per la selezione e l'esecuzione automatizzata su tutte le matrici selezionate;
- `reprocess_missing.py`: script per rieseguire l'elaborazione su file mancanti o incompleti;
- `cleanup_misplaced.py`: script per la gestione e il riposizionamento dei file generati in posizioni errate;
- `collector_performance.py`: raccolta e analisi delle statistiche di performance;
- `menu.py`: interfaccia testuale interattiva per l'utente.

È inoltre disponibile `show_selection.py`, uno script di utilità per la visualizzazione formattata delle matrici selezionate.

3.2 Pre-processamento: rimozione colonne vuote

Prima dell'esecuzione dell'algoritmo, il sistema applica automaticamente un passo di pre-processamento che rimuove tutte le colonne completamente vuote (contenenti solo zeri) dalla matrice di input.

Questo perché una colonna vuota non copre alcuna riga, quindi non può mai far parte di un Minimal Hitting Set. La sua inclusione aumenterebbe inutilmente lo spazio di ricerca da $2^{M'}$ a 2^M (con M' numero di colonne ridotte, M numero di colonne originali, $M' < M$) senza modificare l'insieme delle soluzioni.

Il sistema mantiene un vettore `col_map` che mappa gli indici delle colonne ridotte agli indici originali. Questo permette di:

- Eseguire l'algoritmo sulla matrice ridotta ($N \times M'$);
- Ricostruire gli MHS con gli indici originali nel file di output.

L'opzione `--no-reduction` disabilita questa ottimizzazione, mantenendo tutte le colonne nella matrice elaborata. Questo è utile solo per:

- Debugging dell'implementazione;
- Confronti teorici tra configurazioni diverse;
- Validazione della correttezza dell'ottimizzazione.

La riduzione è sempre consigliata e attiva per default in tutti gli script.

3.3 Esecuzione su singola matrice (run.py)

Lo script `run.py` permette di eseguire il solver su una singola matrice.

La sintassi generale è la seguente:

```
python run.py [file_matrice] [opzioni]
```

`file_matrice` rappresenta il percorso del file `.matrix` da elaborare (default: `esempio.matrix`).

Le opzioni disponibili sono:

- **Timeout** (`--timeout=N`): imposta il tempo massimo di esecuzione in secondi. Se omesso, non c'è limite temporale. In caso di timeout, l'esecuzione viene interrotta e viene salvato un file `.mhs` contenente tutti gli MHS trovati fino a quel momento;
- **Modalità di esecuzione** (`--serial` o `--parallel`): forza l'utilizzo del solver seriale o parallelo. Se omesso, la modalità viene scelta automaticamente in base alle dimensioni della matrice utilizzando i criteri di categorizzazione definiti per la selezione delle matrici di test (Sezione 4.1): usa la modalità seriale con le categorie *trivial*, *tiny* e *small*, mentre per le altre (*medium*, *large*, *xlarge*) si utilizza la modalità parallela.

La scelta automatica è implementata nella funzione `is_small_matrix()` in `utility.py`;

- **Directory di output** (`--outdir=DIR`): specifica la cartella dove salvare il file `.mhs` generato. Se omessa, il file viene salvato nella stessa directory del file di input;
- **Monitoraggio memoria** (`--memory-monitoring` e `--memory-threshold=N`): abilita il monitoraggio continuo della RAM (solo in modalità parallela). Quando si usa il menu interattivo, questa opzione è abilitata per default; quando si invocano gli script direttamente da riga di comando, nella versione parallela richiede il flag esplicito `--memory-monitoring`.

Il parametro `--memory-threshold=N` configura la soglia percentuale. I valori validi sono compresi tra 50% e 99% (default: 95%). Valori più bassi offrono maggiore margine di sicurezza, ma potrebbero interrompere prematuramente l'esecuzione, per questo motivo non vengono permessi. Quando il monitoraggio è disabilitato, la soglia non è rilevante.

Questi parametri sono utili per matrici molto grandi o sistemi con RAM limitata proprio per prevenire crash da esaurimento memoria;

- **Parametri di parallelizzazione** (`--processes=N` e `--batch-size=SIZE`): ottimizzano l'esecuzione parallela per specifiche configurazioni hardware (attivi solo in modalità parallela):
 - `--processes=N`: numero di processi worker (default: CPU count - 1). I valori consigliati sono: 2-3 per CPU a 4 core, 4-7 per 8 core, 8-15 per 16+ core. Troppi processi causano overhead eccessivo;
 - `--batch-size=SIZE`: soglia minima per la dimensione dei batch (default: 1000). La dimensione effettiva viene calcolata adattativamente come descritto nella Sottosezione 2.3.3.

Se omessi, utilizzano valori automatici ottimizzati;

- **Riduzione colonne vuote** (`--no-reduction`): disabilita la rimozione automatica delle colonne completamente vuote. Per default, le colonne vuote vengono rimosse per migliorare le prestazioni senza alterare i risultati (vedi Sezione 3.2), riducendo lo spazio di ricerca da 2^M a $2^{M'}$. Mantenere tutte le colonne è utile solo per debugging o confronti teorici;
- **Deduplicazione globale** (`--skip-global-dedup`): disabilita la deduplicazione globale delle ipotesi nel solver parallelo. Per default, la deduplicazione è attiva per garantire robustezza, ma può essere disabilitata per confronti teorici o debugging. È attivo solo in modalità parallela.

3.3.1 Esempi di utilizzo

Esecuzione con timeout di 60 secondi, modalità seriale forzata:

```
python run.py matrice.matrix --timeout=60 --serial
```

Esecuzione in modalità parallela forzata con monitoraggio memoria (soglia 90%):

```
python run.py matrice_grande.matrix --parallel --memory-
    ↪ monitoring --memory-threshold=90
```

Esecuzione parallela con parametri personalizzati (8 processi, batch da minimo 2000):

```
python run.py matrice_xlarge.matrix --parallel --processes
    ↪ =8 --batch-size=2000
```

3.4 Esecuzione batch automatizzata (setup.py)

Lo script `setup.py` gestisce l'intero workflow di elaborazione automatica su un insieme di matrici di test. È progettato per esperimenti sistematici e raccolta di statistiche di performance.

L'esecuzione di `setup.py` esegue le seguenti fasi in sequenza:

1. **Verifica e creazione cartelle:** crea le cartelle specificate per le matrici selezionate e la cartella dei risultati (`risultati_auto/`, `risultati_parallel/` o `risultati_serial/` a seconda della modalità scelta) se non esistono;
2. **Selezione matrici:** se la cartella delle matrici selezionate è vuota, esegue `matrices_selection.py` che:
 - Scansiona le cartelle `benchmarks/` (o altre specificate);
 - Analizza ogni matrice (dimensioni, densità);
 - Categorizza le matrici (trivial, tiny, small, medium, large, xlarge);
 - Seleziona un sottoinsieme rappresentativo;
 - Copia le matrici selezionate nella cartella specificata;
 - Genera `selection.json` con metadati.
3. **Esecuzione solver:** per ogni matrice nella cartella selezionata:
 - Determina automaticamente la modalità (seriale/parallela);
 - Esegue il solver tramite `run.py`;
 - Verifica il codice di uscita del processo per rilevare errori o timeout;
 - Controlla che il file `.mhs` sia stato effettivamente generato;
 - Salva il file `.mhs` nella cartella dei risultati appropriata;
 - In caso di errore, visualizza diagnostica e continua con le matrici successive;
 - Al termine, mostra un riepilogo con statistiche di successo/fallimento.
4. **Verifica e pulizia post-elaborazione:**
 - Esegue `cleanup_misplaced_mhs()` per spostare eventuali file `.mhs` trovati in posizioni errate;

- Esegue `reprocess_missing_matrices()` per riprocessare automaticamente matrici che non hanno generato file `.mhs`;
- Entrambe le operazioni utilizzano gli stessi parametri specificati per il batch principale (timeout, modalità, monitoraggio memoria, directory, ecc.);
- Operano in modalità non interattiva senza richiedere conferme;
- Eventuali errori vengono loggati, ma non bloccano il workflow.

5. **Raccolta performance:** esegue `collector_performance.py` che:

- Scansiona tutti i file `.mhs` generati;
- Estrae metriche (tempo, memoria, MHS trovati, completamento);
- Genera `results.json` nella cartella dei risultati.

6. **Analisi statistiche:** genera `statistiche_prestazioni.txt` con report dettagliato delle performance per categoria nella cartella dei risultati.

La sintassi generale per eseguire lo script è la seguente:

```
python setup.py [opzioni]
```

Le opzioni disponibili sono le stesse di `run.py` (Sezione 3.3) e vengono applicate uniformemente a tutte le matrici elaborate nel batch. Inoltre, sono disponibili le seguenti opzioni specifiche:

- `--selected-dir=DIR`: specifica la directory delle matrici selezionate (default: `selezionate/`);
- `--results-dir=DIR`: prende il posto di `--outdir=DIR` presente in `run.py` e specifica la directory dei risultati (default: basato sulla modalità scelta).

3.4.1 Esempi di utilizzo

Esecuzione standard con impostazioni automatiche:

```
python setup.py
```

Esecuzione con timeout di 120 secondi e modalità parallela forzata:

```
python setup.py --timeout=120 --parallel
```

Esecuzione con parametri di parallelizzazione personalizzati:

```
python setup.py --parallel --processes=12 --batch-size
    ↪ =1500
```

Esecuzione con directory personalizzate:

```
python setup.py --selected-dir altre_matrici --results-dir
    ↪ altri_risultati
```

3.4.2 Gestione errori e robustezza

Lo script `setup.py` implementa meccanismi robusti di gestione errori per garantire che l'elaborazione batch continui anche in caso di problemi su singole matrici:

- **Verifica exit code:** dopo ogni esecuzione, controlla il codice di uscita del processo. Un codice diverso da zero indica timeout, crash o errore interno;
- **Continuazione dopo errore:** anche in caso di errore critico su una matrice, l'elaborazione procede con le matrici rimanenti. Questo evita che un singolo caso problematico blocchi l'intero batch;
- **Gestione interruzioni utente:** gli script gestiscono anche le interruzioni da tastiera (`KeyboardInterrupt`). In questo caso, l'intero programma si arresta, dopo aver salvato i risultati parziali ottenuti fino a quel momento;
- **Verifica file output:** controlla che il file `.mhs` sia stato effettivamente creato. La mancanza del file indica un'interruzione anomala (es. timeout prima del salvataggio);
- **Verifica e pulizia post-elaborazione:** dopo l'esecuzione principale, sposta automaticamente i file `.mhs` mal posizionati e riprocesso le matrici mancanti utilizzando gli stessi parametri del batch, garantendo completezza dei risultati senza intervento manuale;
- **Riepilogo statistiche:** al termine, mostra un report con numero totale di matrici processate, successi, fallimenti e percentuale di successo.

Questa gestione è particolarmente importante per batch di grandi dimensioni o matrici challenging che potrebbero causare timeout o esaurimento memoria. Il riepilogo finale permette di identificare rapidamente quali matrici hanno causato problemi per analisi successive.

3.5 Script di utilità e manutenzione

Per facilitare la gestione e il debugging del sistema, sono stati implementati due script helper che automatizzano le operazioni di pulizia dei file mal posizionati e il riprocessamento di matrici mancanti. `setup.py` integra automaticamente le funzionalità di questi script come parte del workflow batch, eliminando la necessità di intervento manuale nella maggior parte dei casi.

3.5.1 Pulizia file mal posizionati (`cleanup_misplaced.py`)

Durante l'esecuzione batch, può accadere che alcuni file `.mhs` vengano erroneamente salvati nella cartella delle matrici sorgente (`selezionate/`) invece che nella cartella dei risultati appropriata (`risultati_auto/`, `risultati_parallel/` o `risultati_serial/`). Questo script automatizza il processo di identificazione e spostamento. La sintassi è la seguente:

```
python cleanup_misplaced.py [opzioni]
```

Le opzioni disponibili sono:

- `--selected-dir=DIR`: specifica la directory delle matrici selezionate (default: `selezionate/`);
- `--results-dir=DIR`: specifica la directory dei risultati (default: `risultati_auto/`).

Il funzionamento prevede:

- Scansiona ricorsivamente `--selected-dir/benchmarks1/` e `--selected-dir/benchmarks2/`;
- Identifica tutti i file con estensione `.mhs`;
- Sposta automaticamente tutti i file `.mhs` trovati nelle posizioni sbagliate alla directory corretta, sovrascrivendo eventuali conflitti senza richiedere conferma;
- Mostra un riepilogo finale con numero di file spostati ed eventuali errori.

Questo script è particolarmente utile dopo esecuzioni batch interrotte o se si sospetta che alcuni file siano stati salvati nella posizione errata. Previene inconsistenze nei risultati e facilita la pulizia del workspace.

Esempi di utilizzo

Pulizia standard (directory di default):

```
python cleanup_misplaced.py
```

Pulizia su directory personalizzate, per esempio partendo dalla cartella `esempio_matrix` per arrivare alla cartella `esempio_mhs`:

```
python cleanup_misplaced.py --selected-dir esempio_matrix
                           ↪ --results-dir esempio_mhs
```

3.5.2 Riprocessamento matrici mancanti (reprocess_missing.py)

Durante elaborazioni batch molto lunghe, alcune matrici potrebbero non essere state processate a causa di timeout globali, crash o interruzioni manuali. Invece di rieseguire l'intero batch (potenzialmente molto costoso), questo script identifica e riprocesso solo le matrici mancanti.

Funzionalità:

- Confronta le matrici nella cartella con le matrici selezionate con i file `.mhs` nella cartella dei risultati appropriata (`risultati_auto/`, `risultati_parallel/` o `risultati_serial/`);
- Identifica quali matrici non hanno un file di output corrispondente;
- Mostra un elenco dettagliato delle matrici mancanti organizzato per sottocartella e procede direttamente al riprocessamento senza richiedere conferma;
- Esegue il solver su ciascuna matrice mancante utilizzando i parametri specificati;
- Verifica che il file `.mhs` sia stato effettivamente creato;

- Mostra un riepilogo con successi e fallimenti;
- In caso di errore, logga l'eccezione ma prosegue comunque con la collezione delle prestazioni.

La sintassi è la seguente:

```
python reprocess_missing.py [opzioni]
```

Le opzioni disponibili sono le stesse di `setup.py` (Sezione 3.4). Quando invocato nel `setup.py`, queste sono passate in automatico allo script per garantire coerenza nei parametri di esecuzione.

Esempi di utilizzo

Riprocessamento con impostazioni automatiche:

```
python reprocess_missing.py
```

Riprocessamento in modalità parallela forzata con monitoraggio memoria e timer di 60 secondi:

```
python reprocess_missing.py --parallel --memory-monitoring
                           ↪ --memory-threshold=90 --timeout=60
```

Riprocessamento con deduplicazione globale disabilitata:

```
python reprocess_missing.py --skip-global-dedup
```

Riprocessamento con directory personalizzate, per esempio partendo dalla cartella `esempio_matrix` per arrivare alla cartella `esempio_mhs`:

```
python reprocess_missing.py --selected-dir esempio_matrix
                           ↪ --results-dir esempio_mhs
```

3.6 Menu interattivo (`menu.py`)

Il menu interattivo rappresenta il punto di ingresso principale per l'utente. Offre quattro opzioni:

1. **Eseguire una singola matrice:** permette di selezionare una matrice e configurare tutti i parametri di esecuzione in modo guidato, per poi avviare `run.py` con i parametri scelti;
2. **Eseguire il programma automatico:** avvia `setup.py` per selezionare le matrici dai benchmarks forniti e per l'elaborazione di quelle incluse nel catalogo;
3. **Informazioni e aiuto:** mostra una guida dettagliata con spiegazione di tutti i parametri e modalità di utilizzo;
4. **Esci:** termina il programma.

Il menu gestisce autonomamente:

- Elenco delle matrici disponibili nella cartella corrente;

- Validazione dell'input utente;
- Costruzione del comando con i parametri specificati, con traduzione degli input in flag appropriati passati agli script `run.py` o `setup.py`. Si occupa di gestire anche i parametri avanzati (timeout, modalità, monitoraggio memoria, parallelizzazione, riduzione colonne vuote, deduplicazione globale);
- Esecuzione del comando e visualizzazione dell'output;
- Gestione di interruzioni (solo con `Ctrl+C`) ed errori.

Per avviare il menu interattivo è necessario eseguire il seguente comando da riga di comando:

```
python menu.py
```

3.7 Formato file di input e output

3.7.1 File di input (.matrix)

Il file di input è un file testuale con estensione `.matrix` che contiene una matrice binaria. Il formato è:

- Ogni riga della matrice è su una riga del file;
- Gli elementi sono separati da spazi o virgole;
- I valori ammessi sono 0 e 1;
- Linee vuote e commenti (iniziano con `;;;`) vengono ignorati.

La funzione `parse_matrix_file(path)` in `utility.py` gestisce il parsing.

3.7.2 File di output (.mhs)

Il file di output è un file testuale con estensione `.mhs` che contiene:

1. **Intestazione commentata** (righe che iniziano con `;;;`):
 - Nome del solver utilizzato (`mhs_solver.py` o `mhs_solver_parallel.py`);
 - Informazioni sulla matrice originale (N, M , origine, densità, categoria);
 - Numero di colonne rimosse (se pre-processamento attivo) e indici delle colonne non vuote;
 - Livello massimo di esplorazione;
 - Numero totale di MHS trovati, con relativa cardinalità minima e massima;
 - Indicazione se l'esplorazione è stata completata o interrotta;
 - Indicazione sul livello in cui si è interrotta l'esplorazione;
 - Numero totale di ipotesi generate;
 - Numero di MHS trovati per livello;

- Prestazioni: tempo di esecuzione (reale e CPU) e memoria utilizzata (RSS e picco).

2. Lista degli MHS trovati:

- Ogni MHS è rappresentato da una riga binaria di lunghezza M (colonne originali);
- Il bit in posizione i è 1 se la colonna i appartiene all'MHS;
- Gli MHS sono ordinati per cardinalità crescente.

Esempio di file `.mhs` (modalità seriale):

```
;;; MHS generati dal solver mhs_solver.py
;;;
;;; Matrice di input:
;;; |N| (righe) = 3
;;; |M| (colonne) = 5
;;; Matrice ridotta |M'| (colonne non vuote) = 5
;;; Livello massimo di esplorazione = 5
;;;
;;; Numero di MHS trovati = 3
;;; Cardinalita' minima = 1, Cardinalita' massima = 2
;;; Completato? True
;;; Timeout imposto = 10 secondi
;;; Tempo trascorso = 0.0021 secondi
;;;
;;; Numero ipotesi generate per livello:
;;;     Livello 0: 5
;;;     Livello 1: 6
;;;     Livello 2: 1
;;;     Livello 3: 0
;;;
;;; MHS trovati per livello:
;;;     Livello 1: 1 MHS
;;;     Livello 2: 2 MHS
;;;     Totale: 3 MHS
;;;
;;; Prestazioni:
;;;     Tempo reale = 0.0031 s
;;;     CPU time totale = 0.0000 s
;;;     CPU time singoli worker: non presente (esecuzione
     ↪ seriale)
;;;     Memoria RSS = 20380.0 KB
;;;     Picco memoria = 18.0 KB
;;;
0 0 0 1 0
1 0 1 0 0
0 1 1 0 0
```

In modalità parallela, la sezione prestazioni include dettagli aggiuntivi sui tempi CPU dei worker:

```
;;; Prestazioni:
;;;     Tempo reale = 1.2345 s
```

```
;;;      CPU time totale = 4.5678 s
;;;      CPU time master = 0.1234 s
;;;      CPU time worker (totale) = 4.4444 s
;;;      CPU time worker (max) = 0.9876 s
;;;      CPU time worker (media) = 0.7407 s
;;;      Numero worker = 6
;;;      CPU time singoli worker per livello:
;;;          Livello 1: [0.1234, 0.2345, 0.3456, 0.4567,
;;;          ↪ 0.5678, 0.6789] s
;;;      Memoria RSS = 51234.0 KB
;;;      Picco memoria = 67890.0 KB
```

La funzione `write_mhs_output()` in `utility.py` gestisce la generazione del file.

Capitolo 4

Sperimentazione e risultati

In questo capitolo descriviamo la sperimentazione condotta: le matrici utilizzate, come sono state selezionate e categorizzate, gli obiettivi degli esperimenti e i risultati ottenuti con la relativa interpretazione.

4.1 Matrici di test e categorizzazione

Le sperimentazioni sono state condotte su un insieme di istanze benchmark organizzate nelle cartelle `benchmarks1` e `benchmarks2`. Queste matrici provengono dal repository fornитoci alla consegna e coprono un ampio spettro di caratteristiche.

4.1.1 Criteri di categorizzazione

Le matrici sono state classificate automaticamente in sei categorie in base a:

- Numero di righe (N);
- Numero di colonne ridotte (M' , dopo rimozione colonne vuote).

La categorizzazione tiene conto della complessità esponenziale dell'algoritmo MHS. Le sei categorie definite sono:

1. ***Trivial***: $N \leq 1$ o $M' \leq 1$. Soluzioni banali;
2. ***Tiny***: $N \leq 3$ e $M' \leq 15$. Matrici minuscole;
3. ***Small***: $N \leq 5$ e $M' \leq 30$. Matrici piccole;
4. ***Medium***: $N \leq 6$ e $M' \leq 50$. Matrici di medie dimensioni;
5. ***Large***: $N \leq 8$ e $M' \leq 90$. Matrici grandi;
6. ***Xlarge***: tutte le istanze oltre le soglie di *large*. Qui si ha alta probabilità di timeout o di esaurimento memoria.

Questi criteri sono implementati nella funzione `categorize()` in `matrices_selection.py` e vengono utilizzati coerentemente anche per la scelta automatica del solver con la funzione `is_small_matrix()` in `utility.py`, la quale riconosce le matrici piccole e fa in modo che vengano eseguite in modalità seriale.

4.1.2 Distribuzione delle istanze

Il dataset è stato selezionato utilizzando lo script `matrices_selection.py`, che analizza automaticamente tutte le matrici disponibili nei benchmark e ne seleziona un sottoinsieme rappresentativo per categoria. Lo script garantisce che vengano selezionate matrici uniche sia per nome che per contenuto (ignorando duplicati tra le cartelle `benchmarks1` e `benchmarks2`), e bilancia la selezione tra le due cartelle sorgente per una rappresentazione equilibrata. La distribuzione effettivamente selezionata è:

- **Trivial**: 3 istanze (`c880.005.matrix`, `74182.006.matrix` e `74182.001.matrix`);
- **Tiny**: 10 istanze (`74182.025.matrix`, `74182.022.matrix`, `74182.021.matrix`, `74182.013.matrix`, `74182.010.matrix`, `74182.004.matrix`, `74L85.009.matrix`, `74181.007.matrix`, `74181.003.matrix` e `c1908.000.matrix`);
- **Small**: 10 istanze (`74182.044.matrix`, `74283.000.matrix`, `c5315.002.matrix`, `74181.006.matrix`, `c880.009.matrix`, `74182.042.matrix`, `c880.002.matrix`, `c432.003.matrix`, `74182.047.matrix` e `c499.003.matrix`);
- **Medium**: 10 istanze (`c880.028.matrix`, `c432.007.matrix`, `74181.000.matrix`, `c1908.018.matrix`, `c5315.009.matrix`, `c499.002.matrix`, `c880.044.matrix`, `74L85.025.matrix`, `c2670.032.matrix` e `c880.049.matrix`);
- **Large**: 5 istanze (`c1355.001.matrix`, `74283.018.matrix`, `c499.020.matrix`, `74L85.020.matrix` e `c499.012.matrix`);
- **Xlarge**: 5 istanze (`c5315.238.matrix`, `c499.145.matrix`, `c1355.054.matrix`, `c7552.279.matrix` e `c880.224.matrix`).

La distribuzione bilanciata (in totale sono 43 matrici) favorisce istanze *tiny*, *small* e *medium* che forniscono un buon equilibrio tra copertura dei casi e tempo di esecuzione ragionevole. Le istanze *large* e *xlarge* servono principalmente per testare il comportamento dell'algoritmo su casi limite e la gestione dei timeout.

4.2 Ambiente di test e metriche

4.2.1 Hardware e software

Gli esperimenti sono stati eseguiti su:

- **Processore**: AMD Ryzen 5 5600H with Radeon Graphics (3.30 GHz);
- **RAM**: 16 GB;
- **Processori**: 1 processore fisico, 6 core fisici e 12 thread logici;
- **Sistema Operativo**: Windows 11 Home;
- **Python**: versione 3.12.4.

4.2.2 Configurazione dei parametri

Sono stati raccolti i risultati di due serie di esperimenti: una forzando l'esecuzione in modalità seriale e l'altra lasciando che il programma scegliesse automaticamente la modalità (seriale o parallela) in base alla dimensione della matrice.

In entrambi i casi, il timeout di ogni istanza è stato fissato a massimo 300 secondi e la soglia di memoria, il cui monitoraggio è stato abilitato nel caso parallelo, è stata fissata al 98% della RAM disponibile. Infine, si è scelto di disabilitare la deduplicazione globale nel caso parallelo (che è attiva di default) utilizzando il flag `--skip-global-dedup`, per evitare overhead teoricamente inutili e misurare le prestazioni pure dell'algoritmo succL.

4.2.3 Metriche raccolte

Per ciascuna istanza sono state raccolte:

- **Tempo reale (wall-clock)**: tempo totale trascorso dall'inizio alla fine dell'esecuzione del programma. Include tutto il tempo impiegato: calcolo effettivo della CPU, attesa di operazioni di input/output e qualsiasi altro overhead di sistema;
- **Tempo CPU**: tempo effettivo durante il quale la CPU è stata utilizzata per eseguire le istruzioni del programma. Per il solver seriale corrisponde al tempo CPU del singolo processo; per il solver parallelo è calcolato come somma del tempo CPU del processo master (che coordina l'esecuzione) e dei tempi CPU di tutti i processi worker (che eseguono i calcoli in parallelo). Dei worker, inoltre, sono stati raccolti i singoli tempi CPU suddivisi per livelli. Non include i tempi di attesa per input/output o l'esecuzione di altri processi del sistema;
- **Memoria RSS**: RAM occupata al termine;
- **Picco di memoria**: massima RAM raggiunta durante l'esecuzione;
- **Numero di MHS trovati**;
- **Numero di ipotesi per livello**;
- **Stato di completamento**: successo, timeout, o esaurimento memoria.

I dati sono aggregati in:

- `[cartella_risultati]/results.json`: dati strutturati completi estratti da tutti i file `.mhs`, contenenti metriche dettagliate per ogni matrice;
- `[cartella_risultati]/statistiche_prestazioni.txt`: report analitico dettagliato con statistiche aggregate per gruppi di benchmark, per categoria, correlazioni densità-prestazioni, statistiche di densità per categoria e, infine, statistiche sintetiche per cartella di origine.

dove `[cartella_risultati]` è `risultati_serial` o `risultati_auto` a seconda della modalità di esecuzione.

4.3 Obiettivi degli esperimenti

Gli esperimenti sono stati progettati per registrare e valutare criticamente le prestazioni spaziali e temporali delle prove condotte, nel rispetto dei requisiti funzionali e non funzionali dell'applicazione. In particolare, ci si è focalizzati su:

1. **Valutazione prestazioni:** misurazione dei tempi di esecuzione (wall-clock e CPU) e dell'utilizzo di memoria (RSS e picco) per ciascuna istanza benchmark, con analisi della distribuzione per categoria di matrice;
2. **Verifica robustezza:** test della capacità dell'applicazione di gestire istanze di diversa complessità, valutazione del tasso di completamento per categoria e analisi dei meccanismi di gestione timeout e interruzioni;
3. **Confronto implementazioni:** valutazione delle differenze prestazionali tra modalità seriale forzata e selezione automatica del solver, analisi dello speedup ottenuto con la parallelizzazione su matrici grandi.

4.4 Risultati ottenuti

I grafici sono stati generati automaticamente dallo script `generate_plots.py` utilizzando Matplotlib/Seaborn.

4.4.1 Sommario delle metriche per categoria

Categoria	MHS	Tempo reale medio (s)	RSS (MB)	Picco RAM (MB)	Completate
Trivial	6	0.00	19.85	0.03	3/3 (100%)
Tiny	152	0.01	19.79	0.04	10/10 (100%)
Small	1'107	15.95	25.91	11.29	10/10 (100%)
Medium	12'150	186.52	296.12	1'205.24	4/10 (40%)
Large	34'488	194.30	254.74	926.43	2/5 (40%)
Xlarge	0	317.19	2'703.85	5'162.56	0/5 (0%)
Totale	47'903	106.56	424.45	991.34	29/43 (67.4%)

Tabella 4.1: Metriche principali per categoria nella modalità automatica.

4.4.2 Modalità automatica

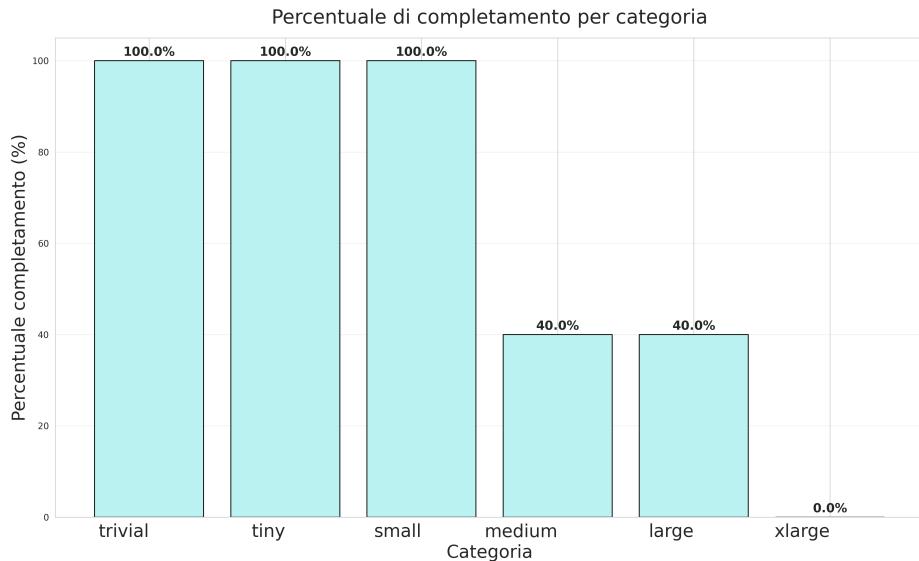


Figura 4.1: Percentuale di completamento per categoria.

In Figura 4.1 si nota che:

- Le categorie *trivial*, *tiny* e *small* mostrano un tasso di completamento del 100% (3/3, 10/10, 10/10 rispettivamente), confermando che l'algoritmo gestisce efficacemente istanze piccole e medie;
- La percentuale scende drasticamente per *medium* (40%, 4/10) e *large* (40%, 2/5), evidenziando il salto di complessità quando le istanze crescono. Il timeout di 300s diventa, quindi, insufficiente per il 60% delle *medium*, delle *large* e per tutte le matrici *xlarge*;
- Il tasso di completamento *globale* è di 29/43 (67.4%).

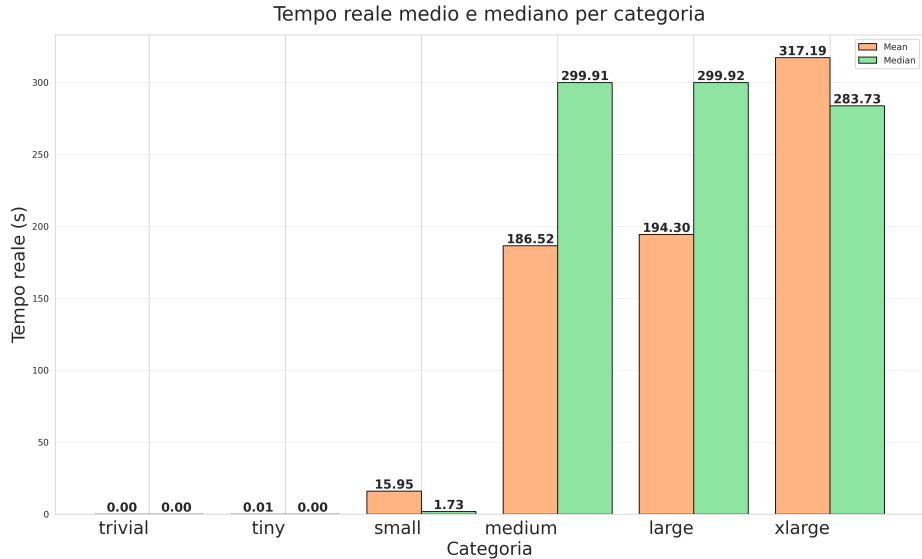


Figura 4.2: Tempo totale per categoria (secondi).

In Figura 4.2 si nota che:

- Il tempo di esecuzione totale cresce rapidamente con la categoria;
- Per le categorie più piccole (*trivial*, *tiny*, *small*), i tempi sono molto contenuti e tutte le istanze completano con successo. Si nota un primo salto significativo passando alla categoria *small*, dove il tempo medio sale a circa 16 secondi;
- Il punto critico si raggiunge con le categorie *medium* e *large*, dove i tempi medi si attestano intorno ai 190 secondi, con molte istanze che raggiungono il timeout di 300s. Questo evidenzia chiaramente il limite pratico dell'algoritmo con le risorse disponibili;
- La categoria *xlarge* mostra tempi estremamente elevati (media oltre 317s), oltre il timeout imposto probabilmente perché vengono considerati anche i tempi spesi in operazioni di I/O, gestione della memoria e chiusura dei processi in maniera anticipata. Questo valore conferma che tali matrici sono al momento intrattabili con i parametri attuali.

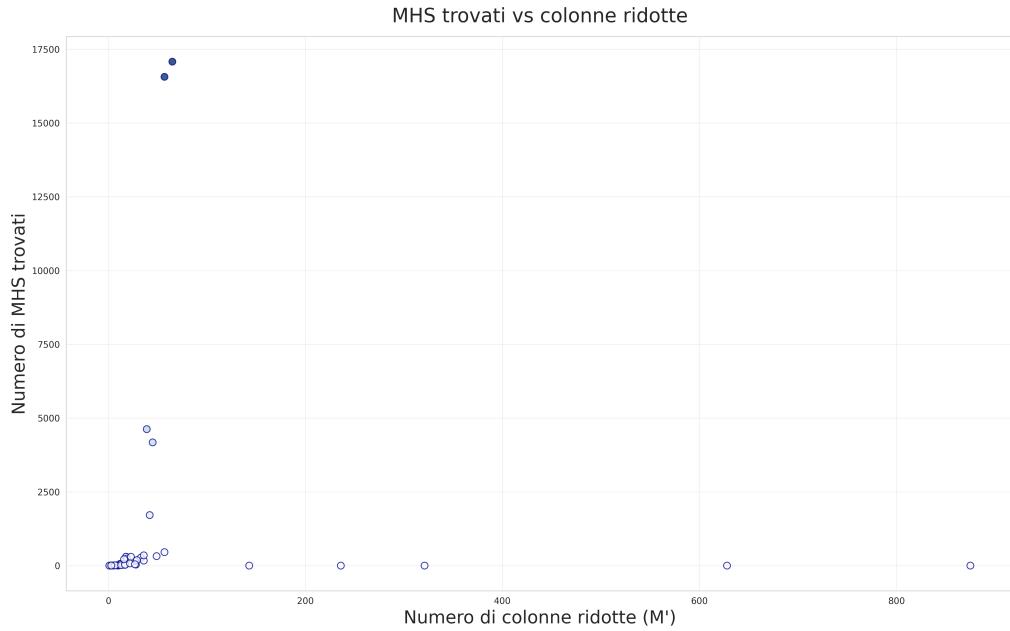


Figura 4.3: Numero di MHS trovati in funzione del numero di colonne ridotte.

In Figura 4.3 si nota che:

- Il numero di MHS cresce esponenzialmente con il numero di colonne ridotte, come atteso dalla teoria. La distribuzione mostra:
 - Matrici con $M' \leq 10$: tipicamente < 100 MHS;
 - Matrici con $10 < M' \leq 30$: range 10–1'000 MHS;
 - Matrici con $M' > 40$: migliaia o decine di migliaia di MHS.
 - Totale MHS trovati (di cui 29 istanze sono state completate): 47'903 MHS. La distribuzione è fortemente sbilanciata:
 - *Trivial*: 6 MHS (0.01%);
 - *Tiny*: 152 MHS (0.32%);
 - *Small*: 1'107 MHS (2.31%);
 - *Medium*: 12'150 MHS (25.36%);
 - *Large*: 34'488 MHS (71.99%);
 - *Xlarge*: 0 MHS (0%, nessuna completata).

Come si nota, le matrici *large* generano la maggior parte degli MHS;

- Le matrici *xlarge* non generano alcun MHS: una spiegazione possibile è che i loro eventuali MHS abbiano cardinalità maggiore di quella elaborata dall'algoritmo. Probabilmente con risorse maggiori (tempo/memoria) si potrebbero ottenere risultati.

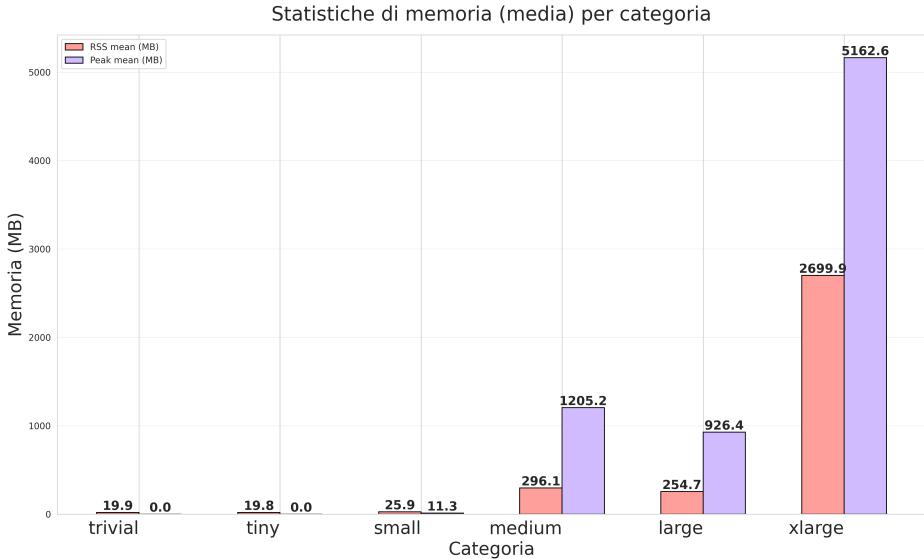


Figura 4.4: Statistiche di memoria (RSS e picco) per categoria.

In Figura 4.4 si nota che:

- Il consumo di memoria (RSS) mostra un andamento crescente con la categoria. In particolare, si nota che in media la categoria *medium* è ≈ 11 volte maggiore rispetto a *small*, mentre *Xlarge* occupa ≈ 2.7 GB;
- I picchi di memoria sono significativamente più alti della RSS media. In particolare, si osserva che nella categoria *medium* il picco è ≈ 4 volte maggiore della RSS media, mentre in *xlarge* il picco raggiunge ≈ 5.2 GB, indicando che durante l'esecuzione si verificano momenti di intenso utilizzo di memoria, la maggior parte dei quali porta a un esaurimento della memoria stessa e alla conseguente chiusura forzata dell'elaborazione;
- L'analisi mostra tre regimi di consumo memoria:
 - *Regime leggero* (*trivial*, *tiny*, *small*): consumo costante e contenuto (< 30 MB);
 - *Regime intermedio* (*medium*, *large*): consumo significativo, ma gestibile (200-300 MB RSS, picchi 1-1.2 GB);
 - *Regime pesante* (*xlarge*): consumo molto elevato (2.7 GB RSS media, picchi 5.2 GB) che può diventare critico su sistemi con memoria limitata.

4.5 Analisi dell'impatto della densità della matrice

Oltre alle dimensioni della matrice (numero di righe e colonne), un fattore critico che influenza le prestazioni dell'algoritmo MHS è la **densità della matrice**, definita come la percentuale di elementi pari a 1 nella matrice. Una matrice **densa** (alta percentuale di 1) presenta più sovrapposizioni tra le righe, il che può influenzare significativamente la complessità dell'esplorazione dello spazio delle soluzioni.

È stata condotta un'analisi statistica per valutare la relazione tra la densità della matrice e le principali metriche prestazionali: tempo di esecuzione, consumo di memoria e numero di MHS trovati. L'analisi utilizza il coefficiente di **correlazione di Pearson** per quantificare la forza e la direzione della relazione lineare.

La formula della correlazione di Pearson è:

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \cdot \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

Dove:

- x_i e y_i sono i valori delle due variabili per la i -esima matrice;
- \bar{x} e \bar{y} sono le medie delle rispettive variabili;
- n è il numero di matrici (43 nel nostro caso);

Il risultato r è un valore compreso tra -1 e 1.

Nota: Questa formula rappresenta la correlazione per popolazioni complete. Nel codice Python (`statistics.correlation()`), viene utilizzata una variante con correzione campionaria che divide per $n - 1$ invece di n nel calcolo delle deviazioni standard. Per dataset di dimensioni moderate come il nostro ($n = 43$), la differenza è trascurabile (ordine di 10^{-4}).

I tipi di correlazione sono:

- $r > 0$: correlazione *positiva* o *diretta*, cioè all'aumentare di x , aumenta anche y ;
- $r < 0$: correlazione *negativa* o *inversa*, cioè all'aumentare di x , diminuisce y ;
- $r = 0$: nessuna correlazione lineare tra le due variabili, che si dicono *incollegate*.

Per le correlazioni dirette e inverse, sono possibili i seguenti casi:

- $0 < |r| < 0.3$: correlazione *debole*;
- $0.3 \leq |r| < 0.7$: correlazione *moderata*;
- $0.7 \leq |r| \leq 1$: correlazione *forte*.

La direzione della relazione lineare può essere utile a dimostrare empiricamente se la densità è un fattore rilevante per le prestazioni, aiutando a interpretare il perché alcune matrici sono più difficili da risolvere. Non predice con certezza, ma identifica tendenze statistiche nei dati sperimentali.

Per esempio, un valore di r vicino a -1 indica una forte correlazione negativa, suggerendo che all'aumentare della densità, la metrica considerata (per esempio il tempo reale di esecuzione) tende a diminuire. Al contrario, un valore vicino a 1 indicherebbe che all'aumentare della densità, il valore della metrica tende ad aumentare. Un valore vicino a 0 indicherebbe nessuna correlazione lineare significativa tra le due variabili.

Per garantire completezza metodologica, l'analisi è stata condotta considerando i risultati ottenuti dalla modalità automatica e secondo due approcci distinti:

1. **Solo matrici completate:** considera esclusivamente le 29 istanze completate con successo, escludendo quelle terminate per timeout o esaurimento memoria;
2. **Tutte le matrici:** include tutte le 43 istanze elaborate, indipendentemente dallo stato di completamento.

Il confronto tra i due approcci permette di valutare l'impatto dei valori censurati (timeout) sulla validità statistica delle correlazioni e di comprendere meglio il comportamento dell'algoritmo su matrici di diversa complessità.

4.5.1 Approccio 1: solo matrici completate

Nella Tabella 4.2 sono riportati i risultati dell'analisi di correlazione calcolata esclusivamente sulle 29 istanze completate con successo.

Metrica	Correlazione di Pearson	Interpretazione
Tempo di esecuzione reale	0.281	Debole correlazione positiva
Memoria RSS	0.500	Moderata correlazione positiva
Picco memoria	0.401	Moderata correlazione positiva
MHS trovati	0.370	Moderata correlazione positiva

Tabella 4.2: Correlazioni densità-prestazioni (solo 29 matrici completate).

Nota: i valori di correlazione sono calcolati utilizzando i valori individuali di ciascuna delle 29 istanze completate, escludendo quelle terminate per timeout o esaurimento memoria per evitare distorsioni da valori censurati.

Di seguito sono riportati i grafici di dispersione per ciascuna metrica.

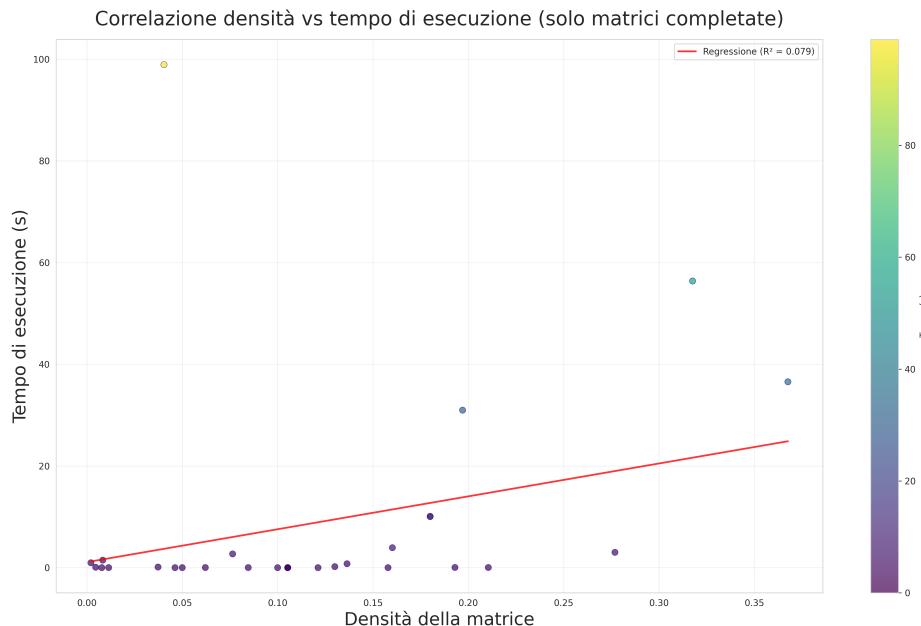


Figura 4.5: Correlazione tra densità della matrice e tempo reale di esecuzione.

Nella Figura 4.5 della correlazione densità-tempo si osserva che la correlazione è debolmente positiva (0.281). Ciò indica che matrici più dense tendono sì a richiedere

più tempo di esecuzione, ma che questa correlazione non è particolarmente forte e quindi non si verifica in tutti i casi.

La *linea di regressione* (retta che minimizza la somma dei quadrati delle distanze tra i punti dati e la linea stessa), calcolata tramite il metodo dei minimi quadrati, sembra seguire l'andamento suggerito dalla correlazione, anche se non perfettamente: c'è una grande dispersione attorno alla retta. Ciò significa che ci sono anche altri fattori (come le dimensioni e la struttura specifica della matrice) che influenzano le prestazioni.

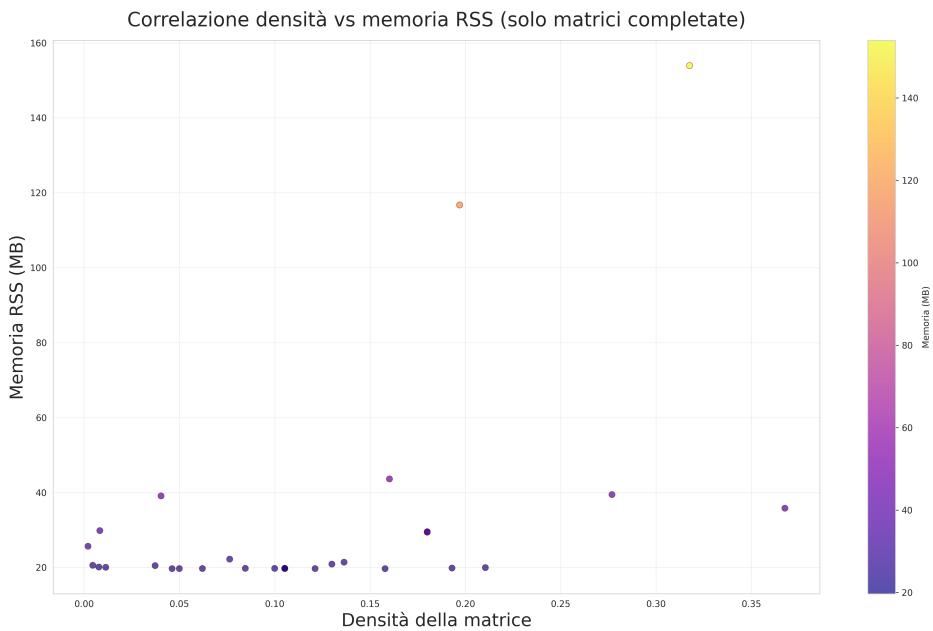


Figura 4.6: Correlazione tra densità della matrice e memoria RSS totale.

In Figura 4.6 si osserva una correlazione positiva e moderata (0.500). Questa suggerisce che matrici più dense tendono a utilizzare più memoria RSS, con una relazione più consistente rispetto al tempo di esecuzione. Questo può essere dovuto al fatto che matrici dense richiedono l'esplorazione di porzioni più ampie dello spazio delle soluzioni, accumulando più strutture dati intermedie.

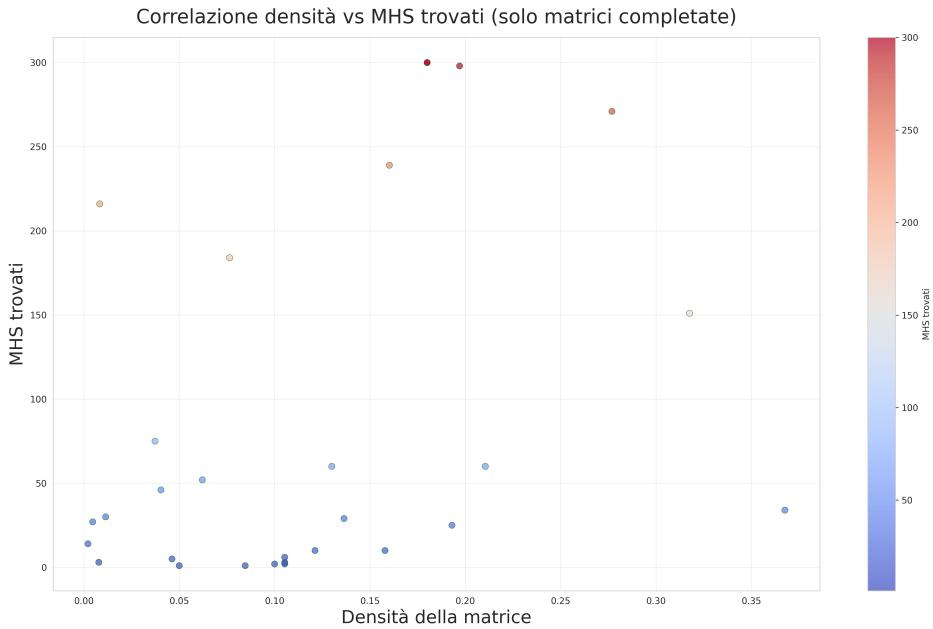


Figura 4.7: Correlazione tra densità della matrice e numero totale di MHS trovati.

In Figura 4.7 si osserva una correlazione positiva e moderata (0.370). Questo indica che matrici più dense tendono a produrre un numero maggiore di MHS, anche se la relazione non è particolarmente forte. Questo è ragionevole poiché matrici dense possono avere strutture che ammettono più insiemi massimalmente indipendenti. Ciò suggerisce che, sebbene la densità influenzi il numero di MHS, altri fattori strutturali della matrice giocano un ruolo più significativo.

4.5.2 Approccio 2: tutte le matrici

Questo approccio permette di valutare se l'inclusione delle matrici non completate (con valori di tempo censurati al timeout di 300s o per esaurimento della memoria) altera significativamente le correlazioni osservate.

Metrica	Correlazione di Pearson	Interpretazione
Tempo di esecuzione reale	-0.356	Moderata correlazione negativa
Memoria RSS	-0.192	Debole correlazione negativa
Picco memoria	-0.319	Moderata correlazione negativa
MHS trovati	-0.123	Correlazione molto debole

Tabella 4.3: Correlazioni densità-prestazioni (tutte le 43 matrici).

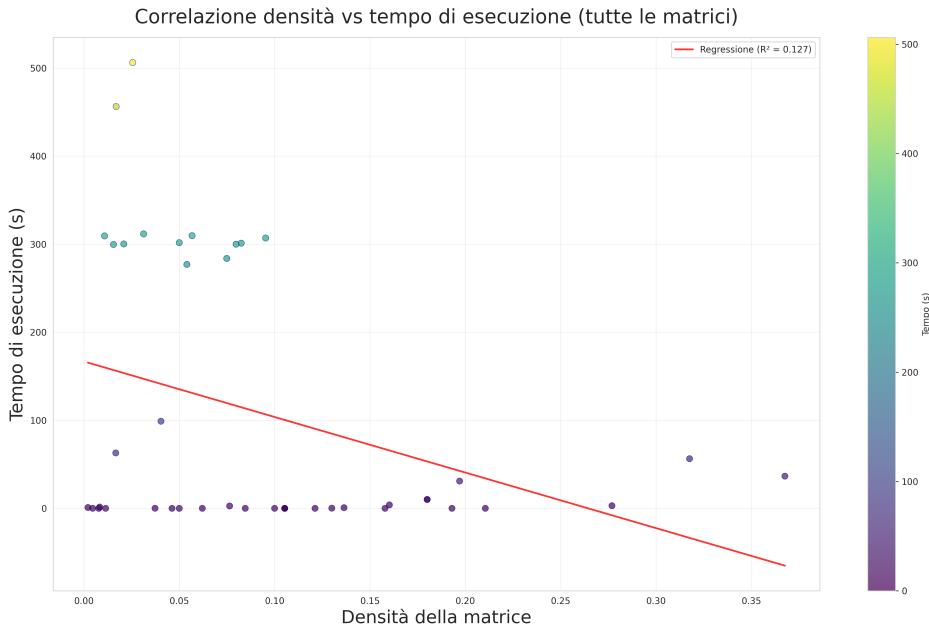


Figura 4.8: Correlazione densità vs tempo (tutte le 43 matrici).

In Figura 4.8 della correlazione densità-tempo si osserva una correlazione moderatamente negativa (-0.356). Questa inversione rispetto all'approccio 1 è dovuta all'inclusione delle matrici non completate, che hanno densità generalmente più bassa, ma tempi fissati al timeout di 300 secondi. Ciò crea una distorsione statistica che suggerisce erroneamente che matrici meno dense richiedano più tempo, quando in realtà il timeout maschera le vere differenze prestazionali.

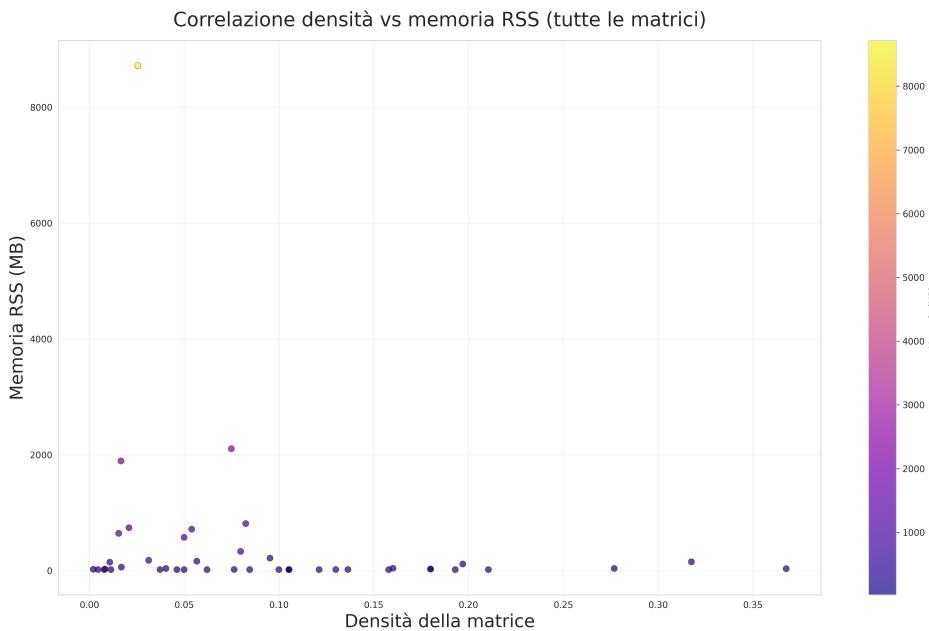


Figura 4.9: Correlazione densità vs memoria RSS (tutte le 43 matrici).

In Figura 4.9 si osserva una correlazione molto debole e negativa (-0.192) tra densità e memoria RSS. Anche in questo caso, l'inclusione delle matrici non completate (che spesso esauriscono la memoria) contribuisce a questa distorsione, suggerendo che

matrici meno dense consumino più memoria, quando invece il pattern opposto è più rappresentativo del comportamento normale dell'algoritmo.

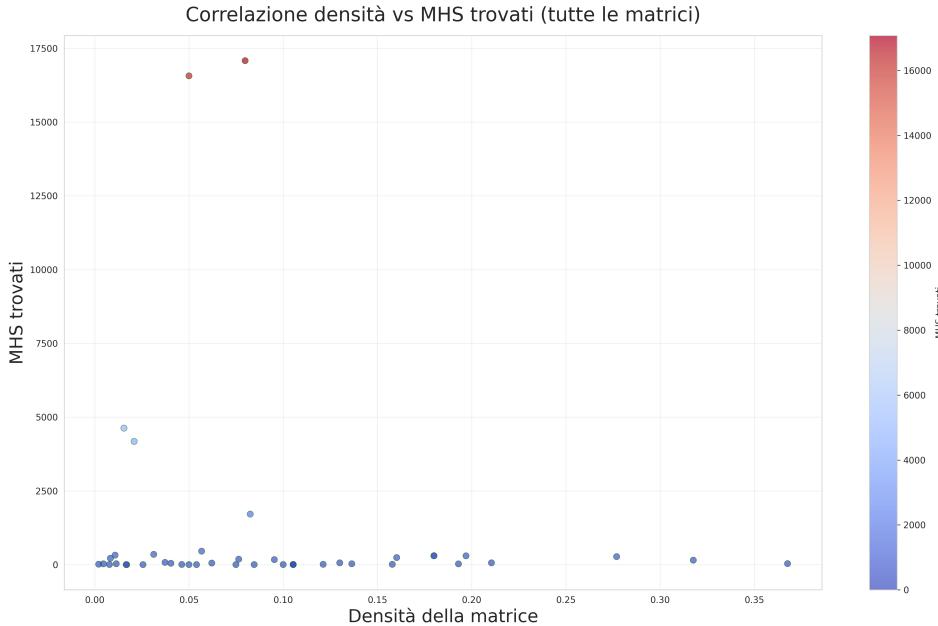


Figura 4.10: Correlazione densità vs MHS trovati (tutte le 43 matrici).

In Figura 4.10 si osserva una correlazione molto debole e negativa (-0.123) tra densità e numero di MHS trovati. Questa relazione quasi nulla riflette il fatto che molte matrici non completate (con densità variabile) producono zero MHS, diluendo qualsiasi tendenza positiva che potrebbe emergere dalle sole matrici completate.

4.5.3 Confronto tra i due approcci

La Tabella 4.4 confronta i risultati ottenuti con i due approcci.

Metrica	A1: solo completate	A2: tutte	A1-A2: Differenza
Tempo esecuzione reale	0.281	-0.356	0.637
Memoria RSS	0.500	-0.192	0.692
Picco memoria	0.401	-0.319	0.720
MHS trovati	0.370	-0.123	0.494

Tabella 4.4: Confronto correlazioni tra approccio 1 (solo completate, 29) e approccio 2 (tutte, 43).

In sintesi:

- **Approccio 1** (solo completate): mostra correlazioni positive, suggerendo che tra le matrici risolvibili, quelle più dense richiedono più risorse, ma producono più soluzioni;
- **Approccio 2** (tutte): mostra correlazioni negative a causa dell'effetto distortivo delle matrici non completate, che sono sparse e vanno in timeout;

- La differenza evidenzia che le correlazioni globali sono influenzate dai casi limite piuttosto che dal comportamento tipico.

Quindi si può affermare che:

1. L'approccio 1 è metodologicamente superiore per l'analisi delle relazioni densità-prestazioni;
2. La selezione del campione è cruciale per evitare bias statistici;
3. Le correlazioni globali possono essere fuorvianti quando includono dati censurati;
4. L'analisi dovrebbe privilegiare i casi "normali" piuttosto che gli estremi.

In generale, possiamo notare come statisticamente la densità sembri essere un fattore rilevante per le prestazioni dell'algoritmo MHS, con effetti positivi su tempo, memoria e numero di soluzioni trovate.

4.6 Confronto tra versione seriale e versione automatica

Come anticipato all'inizio del capitolo, per valutare l'efficacia della strategia di selezione automatica del solver (seriale per matrici piccole, parallelo per matrici grandi), sono state condotte due esecuzioni complete sullo stesso dataset di 43 matrici:

1. **Versione seriale forzata:** tutte le matrici elaborate con il solver seriale (`--serial`);
2. **Versione automatica:** selezione automatica del solver in base alla categoria della matrice (`--auto`), con soglia memoria al 98%.

Di seguito sono riportati i risultati comparativi delle due esecuzioni.

La Tabella 4.5 riassume i risultati principali delle due esecuzioni.

Metrica	Seriale forzata	Automatica
Matrici completate	29/43 (67.4%)	29/43 (67.4%)
Totale MHS trovati	21.443	47.903
Tempo reale medio (s)	116.98	106.56
Memoria RSS media (MB)	44.73	424.45
Picco memoria medio (MB)	803.12	990.95

Tabella 4.5: Confronto prestazioni tra versione seriale forzata e versione automatica.

4.6.1 Analisi del tasso di completamento

Entrambe le versioni hanno completato esattamente lo stesso numero di matrici (29 su 43, pari al 67.4%), con le stesse 14 matrici che hanno raggiunto il timeout. Questo è un risultato significativo che conferma:

- La soglia di timeout di 300 secondi è il fattore limitante principale, non la modalità di esecuzione;

- Le matrici *medium*, *large* e *xlarge* che vanno in timeout sono intrinsecamente troppo complesse per essere completate nel tempo disponibile, indipendentemente dalla strategia di parallelizzazione.

4.6.2 Numero di MHS trovati: differenza critica

La versione automatica ha trovato 47'903 MHS totali, più del doppio rispetto ai 21'443 MHS della versione seriale. Questa differenza sostanziale è dovuta principalmente alle prestazioni superiori sulle matrici *large*:

- **Large (seriale)**: 9'555 MHS totali;
- **Large (automatica)**: 34'488 MHS totali.

Questo dimostra che per matrici di dimensioni rilevanti, la versione parallela riesce a esplorare una porzione significativamente maggiore dello spazio di ricerca prima del timeout, trovando molte più soluzioni valide.

4.6.3 Prestazioni temporali

- **Tempo medio**: la versione automatica è più veloce di 10.42 secondi (116.98s vs 106.56s), dimostrando un vantaggio complessivo nelle prestazioni;
- **Per categoria**:
 - *Trivial/Tiny/Small*: tempi identici. Questo non ci è informativo, in quanto per scelta implementativa entrambe le versioni usano il solver seriale per queste categorie;
 - *Medium*: tempo medio 196.65s (seriale) vs 186.52s (auto), vantaggio marginale per la versione automatica;
 - *Large*: tempo medio 233.67s (seriale) vs 194.30s (auto), vantaggio significativo per la versione automatica nonostante entrambe raggiungano spesso il timeout;
 - *Xlarge*: tempo medio 347.33s (seriale) vs 317.19s (auto), differenza dovuta principalmente alla diversa gestione del timeout e dell'esaurimento delle risorse tra i due approcci.

4.6.4 Consumo di memoria

L'utilizzo di memoria presenta un trade-off interessante:

- **Memoria RSS media**: la versione automatica consuma significativamente più memoria (424.45 MB vs 44.73 MB), principalmente a causa del parallelismo, che mantiene in memoria le strutture dati di più worker contemporaneamente, e che esplora più livelli della ricerca, dovendo gestire molte più ipotesi rispetto alla versione seriale;
- **Picco memoria medio**: la versione automatica ha picchi più elevati (990.95 MB vs 803.12 MB). Questo conferma le ipotesi precedenti. Tuttavia, il picco non cresce in modo proporzionale alla RSS media, suggerendo che la maggior parte del consumo aggiuntivo è distribuito nel tempo piuttosto che concentrato in momenti critici.

4.6.5 Analisi per categoria

La Tabella 4.6 dettaglia le differenze per ciascuna categoria di matrice.

Categoria	Completate	MHS		Tempo (s)	
		Seriale	Auto	Seriale	Auto
Trivial	3/3 (100%)	6	6	0.00	0.00
Tiny	10/10 (100%)	152	152	0.02	0.01
Small	10/10 (100%)	1'107	1'107	15.83	15.95
Medium	4/10 (40%)	10'623	12'150	196.65	186.52
Large	2/5 (40%)	9'555	34'488	233.67	194.30
Xlarge	0/5 (0%)	0	0	347.33	317.19
Totale	29/43 (67.4%)	21'443	47'903	116.98	106.56

Tabella 4.6: Confronto prestazioni per categoria di matrice.

Osservazioni chiave:

- *Trivial/Tiny/Small*: risultati praticamente identici come previsto, poiché entrambe le versioni utilizzano il solver seriale per queste categorie;
- *Medium*: la versione automatica trova più MHS con un tempo medio leggermente inferiore, dimostrando un leggero vantaggio del parallelismo;
- *Large*: qui emerge la differenza più importante: la versione automatica trova molti più MHS con un tempo medio inferiore, confermando che il parallelismo è essenziale per matrici grandi;
- *Xlarge*: nessuna delle due versioni completa alcuna matrice, ma la versione automatica impiega meno tempo per gestire l'interruzione dei processi.

4.6.6 Grafici di confronto

Per un'analisi più dettagliata delle prestazioni a livello di singola matrice, presentiamo dei grafici scatter in scala logaritmica che confrontano direttamente le due versioni. In questi grafici, ogni punto rappresenta una singola matrice del benchmark, con le prestazioni della versione *seriale* sull'asse *x* e quelle della versione *automatica* sull'asse *y*.

La linea diagonale $y = x$ rappresenta prestazioni identiche: punti sotto la diagonale ($y < x$) indicano che la versione automatica performa meglio per tempo e memoria, che devono avere valori bassi; i punti sopra ($y > x$) indicano una prestazione migliore della modalità automatica solo per il numero di MHS trovati, che deve essere il più alto possibile, mentre per tempo e memoria indicano che la versione seriale è superiore (cioè più veloce o che consuma meno).

Ogni categoria di matrice è rappresentata con un marker e un colore distintivi per facilitare l'identificazione della complessità delle istanze: cerchi per *trivial*, quadrati per *tiny*, triangoli per *small*, diamanti per *medium*, stelle per *large* ed esagoni per *xlarge*.

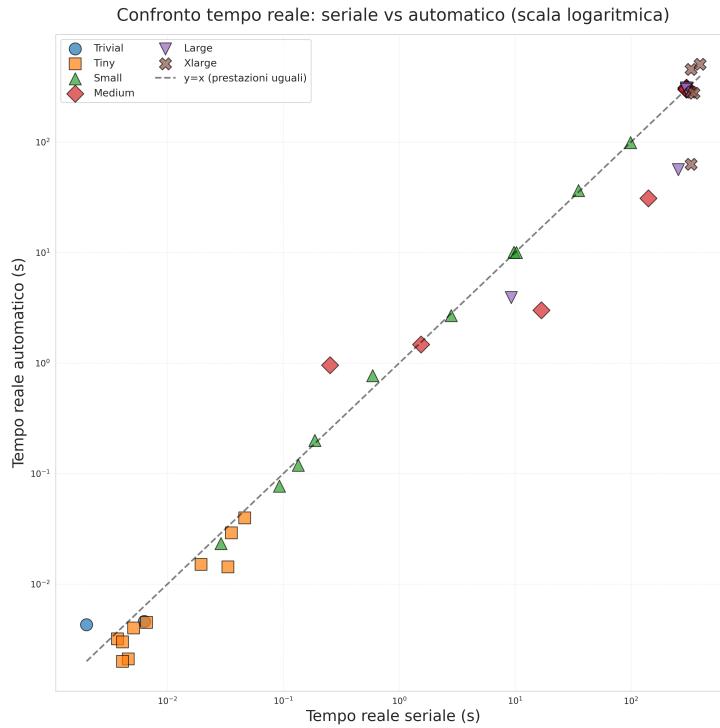


Figura 4.11: Scatter plot del tempo di esecuzione per ogni matrice.

La Figura 4.11 mostra il confronto tra i tempi di esecuzione per ogni matrice. Le categorie più piccole (*trivial, tiny, small*) sono vicine alla diagonale, indicando prestazioni simili tra le due versioni (entrambe in modalità seriale). Le restanti, le più grandi, tendono a essere sotto la diagonale, evidenziando un vantaggio temporale della versione automatica (più veloce) quando attiva il parallelismo.

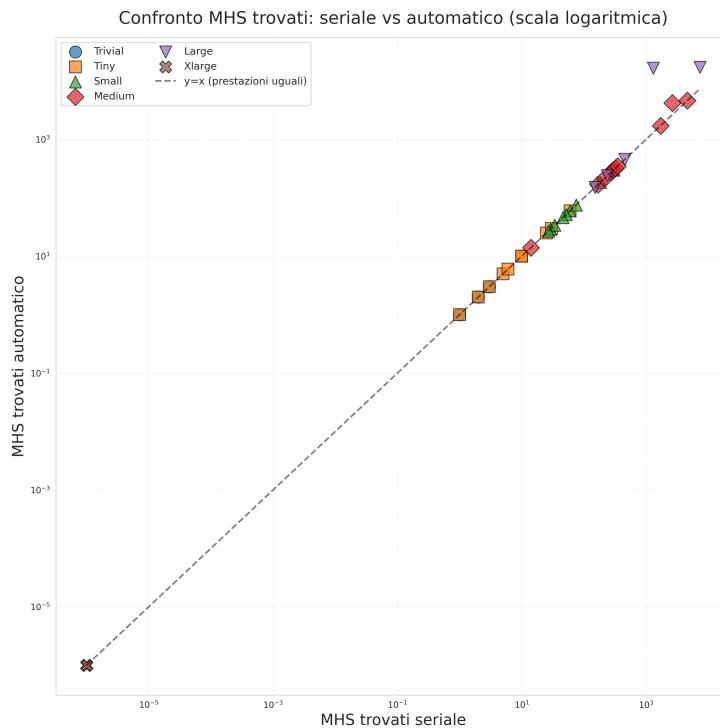


Figura 4.12: Scatter plot del numero di MHS trovati per ogni matrice.

La Figura 4.12 evidenzia il vantaggio della versione automatica in termini di numero di soluzioni trovate. Il grafico mostra come le categorie più piccole abbiano punti vicini alla diagonale (stesse prestazioni), mentre le categorie *medium* e *large* presentano dei punti sopra la diagonale, confermando che la versione automatica, essendo più veloce, riesce a esplorare una porzione maggiore dello spazio di ricerca prima del timeout, trovando quindi un numero superiore di soluzioni.

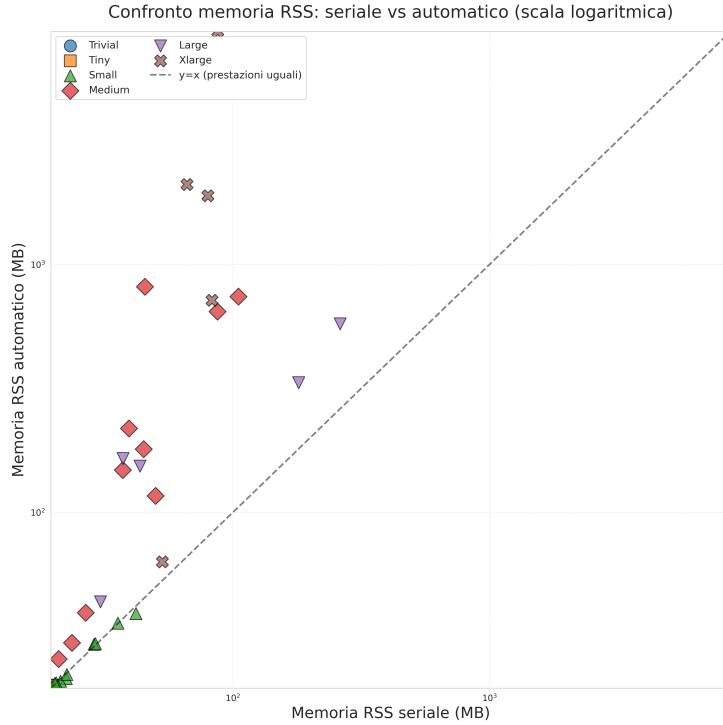


Figura 4.13: Scatter plot della memoria RSS utilizzata per ogni matrice.

La Figura 4.13 evidenzia il consumo di memoria RSS (Resident Set Size) delle due versioni. Si osserva una chiara bipartizione del comportamento: le categorie più piccole (*trivial*, *tiny*, *small*) hanno punti che giacciono esattamente sulla diagonale, avendo un consumo di memoria identico tra le due versioni. Le categorie più grandi (*medium*, *large*, *xlarge*), invece, mostrano punti sopra la diagonale, evidenziando che la versione automatica consuma più memoria RSS rispetto alla seriale. Questo incremento è atteso ed è dovuto al parallelismo, che richiede la gestione simultanea di strutture dati duplicate per ciascun processo worker, code di comunicazione inter-processo, overhead di coordinamento del processo master, stati intermedi di esplorazione mantenuti per più livelli simultanei e buffer temporanei per la sincronizzazione tra processi.

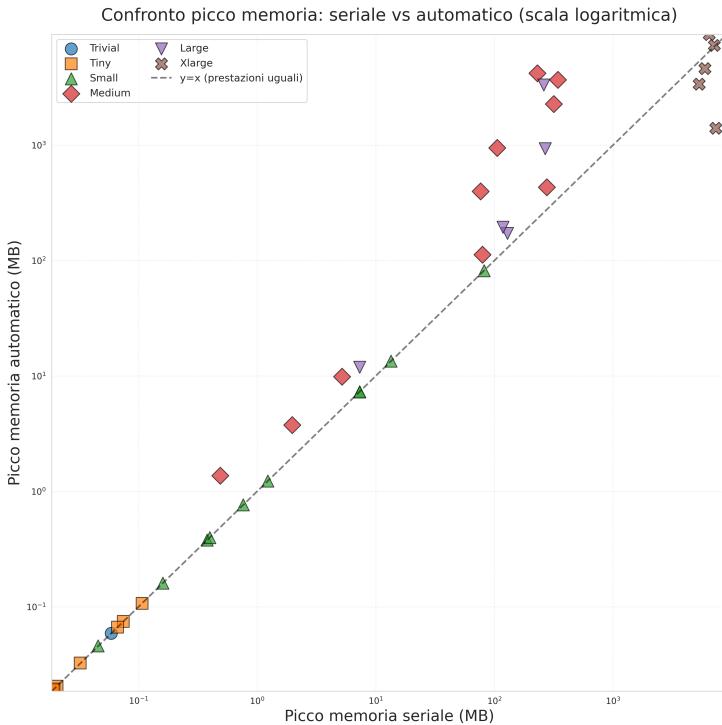


Figura 4.14: Scatter plot del picco di memoria per ogni matrice.

La Figura 4.14 analizza i picchi di memoria raggiunti durante l'esecuzione. Il grafico conferma lo stesso pattern osservato per la memoria RSS: le categorie più piccole hanno punti sulla diagonale, mentre le categorie più grandi mostrano punti sopra la diagonale, indicando che i picchi di memoria della versione automatica sono superiori rispetto alla seriale. Questo è dovuto ai momenti di massima attività parallela, quando tutti i worker sono attivi simultaneamente e il sistema mantiene in memoria diverse strutture dati per ciascun processo.

È importante notare che, sebbene i picchi siano più elevati nella versione automatica, questi rappresentano momenti transitori di massima attività computazionale. Il sistema gestisce efficacemente queste situazioni grazie al monitoraggio attivo della memoria (soglia al 98%), che previene l'esaurimento completo delle risorse terminando precocemente le elaborazioni che diventano troppo onerose. L'incremento di memoria rimane quindi un trade-off accettabile, considerando i significativi guadagni in termini di prestazioni temporali e numero di soluzioni trovate.

4.6.7 Giustificazione della scelta seriale per matrici piccole

Come evidenziato nelle sezioni precedenti, la strategia di selezione automatica del solver utilizza l'implementazione seriale per le categorie *trivial*, *tiny* e *small*, riservando il parallelismo solo per matrici di dimensioni maggiori (*medium*, *large*, *xlarge*). Questa scelta progettuale non è arbitraria, ma è motivata da test empirici che dimostrano come il parallelismo su istanze di piccole dimensioni introduca un overhead che peggiora le prestazioni complessive invece di migliorarle.

Per validare questa strategia, è stata condotta una terza serie di esperimenti utilizzando una **versione parallela forzata** che applica il parallelismo a *tutte* le matrici,

indipendentemente dalla loro dimensione. La Tabella 4.7 confronta le prestazioni di due modalità (seriale forzata e parallela forzata) sulle categorie piccole.

Categoria	Modalità	Tempo reale (s)	Media RSS (MB)	Media picco (MB)
<i>Trivial</i>	Seriale	0.0036	19.90	0.03
	Parallela	0.2929	23.54	0.61
<i>Tiny</i>	Seriale	0.0163	19.71	0.04
	Parallela	0.5513	23.42	0.65
<i>Small</i>	Seriale	15.8311	26.22	11.28
	Parallela	5.7072	40.07	16.53

Tabella 4.7: Confronto prestazioni tra modalità seriale e parallela forzata sulle categorie piccole.

I risultati mostrano chiaramente che:

- **Categorie *trivial* e *tiny*:** il parallelismo forzato degrada drasticamente le prestazioni temporali, con tempi di esecuzione medi rispettivamente 81 volte e 47 volte superiori alla modalità seriale. Questo overhead è dovuto ai costi fissi di inizializzazione dei processi worker, sincronizzazione e comunicazione inter-processo, che dominano completamente il tempo di calcolo effettivo per istanze così piccole. Inoltre, si osserva un incremento significativo della memoria (RSS e picco) dovuto alle strutture dati parallele, completamente sprecato data la semplicità delle istanze;
- **Categoria *small*:** anche per matrici leggermente più grandi, il parallelismo forzato mostra un comportamento ambiguo. Sebbene la modalità parallela sia circa 3 volte più rapida (5.7s vs 15.8s), la strategia *automatica* implementata utilizza comunque la modalità seriale. Questa scelta progettuale è stata fatta per privilegiare la minimizzazione assoluta dell'occupazione di memoria (RSS: 26.22 MB vs 40.07 MB) su questa categoria, accettando un lieve peggioramento delle prestazioni temporali. Si è preferito riservare l'overhead del parallelismo solo alle categorie *medium* e superiori, dove il guadagno temporale diventa essenziale per completare l'esecuzione.

In conclusione, i test con la versione parallela forzata dimostrano empiricamente che l'overhead del parallelismo è controproducente per matrici piccole, giustificando, come già detto, la scelta implementativa adottata.

Capitolo 5

Conclusioni

Il progetto ha sviluppato due implementazioni funzionanti dell'algoritmo per il calcolo dei Minimal Hitting Set: una versione seriale per istanze medio-piccole e una versione parallela per le istanze medio-grandi.

Il sistema sviluppato include inoltre:

- Interfaccia interattiva (`menu.py`) per utilizzo guidato;
- Script di automazione (`setup.py`) per elaborazione collettiva di istanze multiple;
- Raccolta e analisi automatica delle performance (`collector_performance.py`);
- Gestione robusta di timeout e interruzioni con controlli ottimizzati.

Gli esperimenti condotti hanno mostrato che:

- L'algoritmo è molto efficace su istanze fino a *small* (completamento 100%);
- Le istanze *medium* e *large* mostrano già difficoltà significative (completamento 40%);
- Le istanze *xlarge* sono oltre la fattibilità (nessun completamento), con memoria e tempo come principali colli di bottiglia;
- Le ottimizzazioni implementate (rimozione colonne vuote; se attiva, una deduplicazione adattiva; garbage collection) sono fondamentali per gestire istanze di dimensioni realistiche;
- I controlli timeout ottimizzati massimizzano l'utilizzo del tempo disponibile, riducendo gli sprechi e migliorando l'efficienza sui timeout brevi.

In generale, il progetto ha raggiunto gli obiettivi prefissati, fornendo un sistema completo e robusto per il calcolo dei Minimal Hitting Set, con buone performance su istanze di dimensioni moderate e una solida base per futuri miglioramenti e ottimizzazioni.