Progetto Algoritmi e Strutture Dati

Bernocchi Alessandro, Ceresara Gabriele, Tedeschi Jacopo

A.A. 2023/2024

Indice

0 Introduzione

0.1 Definizione del Problema

Questo progetto ha l'obiettivo di sviluppare un algoritmo per il calcolo degli *Hitting Set Minimali* (MHS) di una collezione di insiemi finiti. Un *Hitting Set* (HS) è un insieme che interseca tutti gli insiemi della collezione, mentre un *Minimal Hitting Set* (MHS) è un HS tale che nessun suo sottoinsieme è ancora un HS.

0.2 Applicazioni degli MHS

Il problema del calcolo degli MHS è di grande rilevanza in diversi ambiti applicativi. Essi sono utilizzati, tra le altre cose, per:

- Convertire espressioni booleane tra la forma normale disgiuntiva (DNF) e quella congiuntiva (CNF);
- Individuare insiemi di landmark nel planning AI;
- Isolare difetti nel codice di programmi software;
- Determinare diagnosi nei sistemi diagnostici AI basati su modelli (dove le diagnosi corrispondono agli MHS dei conflitti).

0.3 Difficoltà Computazionale

Il problema di determinare se esiste un HS di cardinalità al massimo k è noto per essere NP-completo, rendendo il suo calcolo particolarmente oneroso dal punto di vista computazionale in quanto non esistono algoritmi in grado di risolverlo in tempo polinomiale.

0.4 Obiettivi della Relazione

In questa relazione verranno analizzate le strutture dati adottate per migliorare l'efficienza dell'algoritmo, le tecniche utilizzate per ridurre la complessità computazionale e i limiti incontrati durante l'implementazione.

1 Scelte Implementative

1.1 Linguaggio di programmazione

Gli algoritmi e il codice utilizzato per la sperimentazione sono scritti in python pertanto esso rappresenta un requisito necessario per l'esecuzione del programma. La scelta del linguaggio di programmazione è ricaduta su di esso in quanto è un linguaggio che presenta una sintassi chiara e vicina al linguaggio naturale che ne facilita la lettura e la conversione dello pseudocodice fornito. Esso è un linguaggio interpretato che funziona su diversi sistemi operativi senza la modifica di parti di codice, inoltre, grazie all'uso di librerie esterne ha prestazioni che non si allontanano da linguaggi compilati come C e RUST.

1.2 Rappresentazione della Matrice e Preprocessing

Il problema del calcolo degli MHS è rappresentato mediante una **matrice binaria**, in cui le righe corrispondono agli insiemi e le colonne agli elementi candidati per far parte della soluzione.

Per migliorare l'efficienza, il solver esegue un **preprocessing della matrice** con le seguenti operazioni:

- **Rimozione di colonne vuote**: Se una colonna non copre alcuna riga (presenta soli zeri), viene eliminata perché non contribuisce a nessuna soluzione.
- **Permutazione delle righe e delle colonne**: Viene effettuata una randomizzazione per evitare pattern sfavorevoli nella ricerca delle soluzioni.

Questi passaggi riducono lo spazio di ricerca e migliorano la probabilità di trovare una soluzione più rapidamente.

1.3 Struttura dei Nodi e Generazione delle Soluzioni

Il solver utilizza una **struttura ad albero** per esplorare lo spazio delle soluzioni, basata su nodi (istanze della classe `Node`). Ogni nodo rappresenta una **ipotetica soluzione parziale** e ha le seguenti caratteristiche:

- Un **vettore binario** che indica quali elementi sono inclusi.
- Un **livello nell'albero** che rappresenta la profondità della ricerca.
- **Metodi di propagazione** per aggiornare i candidati in base alle scelte precedenti.

I nodi vengono generati attraverso la funzione `generate_children()`, che costruisce nuovi candidati **aggiungendo elementi progressivamente** fino a trovare un MHS valido.

1.4 Gestione della Complessità Computazionale

Per affrontare la **complessità computazionale elevata** del problema (NP-completo), il solver utilizza varie strategie:

- **Eliminazione anticipata di ipotesi non promettenti**:

  - Se un nodo non può generare un MHS, viene rimosso immediatamente.
  - Se una soluzione valida viene trovata, viene salvata e non vengono esplorati ulteriori percorsi irrilevanti.

- **Uso di strutture dati ottimizzate**:
  - L'uso di `numpy` per la manipolazione della matrice permette calcoli vettorializzati e riduce il costo computazionale delle operazioni di base.
  - Viene mantenuto un insieme di **soluzioni già trovate**, evitando di esplorare nuovamente le stesse combinazioni.

- **Tracciamento dell'uso di memoria** con `tracemalloc`:
  - Permette di monitorare il consumo di risorse.

1.5 Algoritmo di Ricerca e Costruzione delle Soluzioni

L'algoritmo segue un **approccio iterativo a livelli**:

1. **Si parte da un nodo vuoto** (nessun elemento selezionato).
2. **Si esplorano le ipotesi a livello 0**, generando i primi candidati.
3. **Per ogni nodo esplorato**, si verificano due condizioni:
   - Se è già una soluzione valida, viene salvata.
   - Se è ancora incompleto, vengono generati i suoi figli sinistri; vengono generati solo questi successori poiché ogni ipotesi è successore sinistro di un singolo predecessore. Grazie a questa proprietà ogni ipotesi viene generata una sola volta.
4. **Si continua fino a raggiungere la massima profondità** o finché non ci sono più ipotesi da esplorare.

L'approccio utilizza tecniche di **potatura** per ridurre il numero di combinazioni da esplorare e migliorare l'efficienza complessiva.

2 Compito 1

2.1 Descrizione del problema

L'obiettivo del progetto è lo sviluppo di un'applicazione software in grado di calcolare gli MHS (Minimal Hitting Set) utilizzando gli algoritmi proposti.

Nel materiale fornito, lo pseudocodice distingue esplicitamente tra il concetto di **ipotesi** e la sua **rappresentazione binaria**. Nell'implementazione software si è deciso di semplificare questa distinzione: ogni ipotesi viene identificata univocamente dalla sua rappresentazione binaria, rendendo le due nozioni equivalenti. Ciò consente una maggiore efficienza nella gestione delle ipotesi e una rappresentazione più diretta nel codice.

2.2 Implementazione

Per facilitare la manipolazione di matrici e array, si è fatto uso della libreria esterna `numpy`, che fornisce strutture dati ottimizzate e operazioni vettoriali ad alte prestazioni. Oltre a migliorare l'efficienza computazionale, `numpy` offre una sintassi chiara e compatta per accedere a righe e colonne, filtrare i dati e applicare trasformazioni su larga scala, semplificando notevolmente il lavoro con matrici booleane. L'utilizzo di `numpy` ci consente di non dover valutare metodi alternativi per la lettura delle matrici per colonne in quanto l'accesso è semplificato dalla sintassi intrinseca della libreria.

2.2.1 Parsing dell'input

Una delle prime funzionalità sviluppate è stata quella relativa all'analisi e alla lettura dei file di input. In particolare, è stata implementata una funzione dedicata al parsing dei dati, con lo scopo di convertire il contenuto testuale del file in una matrice booleana utilizzabile nei successivi passaggi.

```
import numpy as np

COMMENT = ';;;'
def read_file(folder, filename):
    file = f"{folder}/{filename}"
    with open(file, 'r') as f:
        lines = f.readlines()

        matrix = []
        for line in lines:
            if line.startswith(COMMENT):
                continue

            line = line.replace('-', '')
            line = line.strip()
            matrix.append([bool(int(x)) for x in line.split()])

        matrix = np.array(matrix, dtype=bool)
        return matrix
```

La funzione riportata si occupa di aprire il file specificato e leggerne il contenuto riga per riga, ignorando le righe di commento (identificate dal prefisso `;;;`). Ogni riga informativa viene pulita da eventuali caratteri superflui (come il trattino `-`), suddivisa in singoli elementi, e trasformata in una lista di valori booleani.

Infine, l'insieme delle righe viene convertito in una matrice `numpy` di tipo booleano. Questa matrice, di dimensione $M \times N$, rappresenta la struttura dati su cui verranno eseguite le elaborazioni principali, come la generazione e valutazione degli insiemi di hitting.

2.2.2 Scrittura dell'output

Successivamente è stata implementata una funzione dedicata alla scrittura dei risultati su file, secondo le specifiche richieste dal progetto. Il file di output viene generato nella cartella `results` e contiene sia informazioni dettagliate sull'esecuzione, sia l'elenco completo degli MHS trovati. Di seguito vengono riepilogati i principali contenuti scritti nel file:

- la lista di tutti i MHS individuati per l'istanza di problema data, rappresentati in forma binaria;
- le dimensioni della matrice originaria e, se applicata, la riduzione tramite rimozione di colonne irrilevanti;
- il numero totale di MHS trovati, assieme alla loro cardinalità minima e massima;
- il tempo totale di esecuzione e la memoria utilizzata durante il calcolo;
- l'eventuale interruzione manuale da parte dell'utente;
- il livello massimo raggiunto dall'algoritmo e il numero di ipotesi generate a ciascun livello.

La funzione `write_solutions` riceve in ingresso il nome del file di input, la lista delle soluzioni trovate (MHS), un oggetto contenente informazioni sulle risorse computazionali (tempo e memoria), e un flag che indica se l'esecuzione è stata interrotta manualmente.

Il contenuto viene scritto in formato testuale, con intestazioni commentate (prefisso `;;;`) per separare chiaramente il riepilogo dell'esecuzione dai risultati. Ogni soluzione è rappresentata da una stringa di 0 e 1 che riflette la corrispondente ipotesi in formato binario, seguita da un trattino.

Inoltre, se durante il pre-processing sono state rimosse delle colonne irrilevanti, queste vengono elencate e viene indicata la reale dimensione della matrice su cui è stata condotta la computazione.

Il seguente estratto illustra un'esempio di output generato dalla funzione:

```
;;; Input matrix 2 X 28
;;; Number of MHS found: 7
;;; Minimum cardinality: 1
;;; Maximum cardinality: 2
;;; Elapsed time: 0.016098976135253906s
;;; Memory used: 11.1298828125 KB
;;; Computation done removing the columns: [1, 3, 5, 6, 7, 8, 11,
    12, 13, 14, 15, 16, 18, 19, 20, 21, 23, 25, 26, 27, 28, 29, 30,
    31, 33]
;;; -> The dimensions of the matrix really used are 2 X 3
;;; Computation stopped at level 3
;;;
;;; Hypotesis generated for each level:
;;; Level 0: 1 hypotesis
```

```
;;; Level 1: 3 hypoteses
;;; Level 2: 1 hypotesis
;;;
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 -
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 -
0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -
0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -
0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 -
```

La funzione di scrittura dei risultati ottenuti rappresenta un passaggio cruciale per la tracciabilità e la valutazione delle prestazioni dell'algoritmo implementato, fornendo all'utente un riepilogo completo e leggibile delle diverse esecuzioni sui vari file di output

2.2.3 Terminazione anticipata

Per garantire una gestione corretta dell'interruzione manuale dell'esecuzione (ad esempio tramite la combinazione di tasti `CTRL + C`), è stato implementato un meccanismo di terminazione controllata utilizzando la libreria `signal` di Python.

Grazie all'uso di un handler personalizzato associato al segnale `SIGINT`, è possibile intercettare l'interruzione da tastiera, salvare i risultati parziali fin lì ottenuti, e terminare il programma in modo sicuro. In questo modo si evita la perdita dei dati già calcolati e si rende l'applicazione più robusta durante esecuzioni lunghe o sperimentazioni batch.

La funzione definita per la gestione del segnale esegue le seguenti operazioni:

- stampa un messaggio a schermo per informare l'utente dell'interruzione;
- imposta il flag `solver.stopped` per notificare al resto del programma che l'elaborazione è stata fermata volontariamente;
- calcola il tempo trascorso dall'inizio dell'esecuzione;
- invoca la funzione `write_solutions` per scrivere su file le soluzioni parziali ottenute, insieme alle informazioni sulle risorse computazionali e allo stato di interruzione;
- infine, chiude il programma in modo pulito tramite `sys.exit(0)`.

Il codice seguente mostra l'implementazione di tale meccanismo:

```python
import sys
import time
import signal

def signal_handler(sig, frame):
    print("\nCtrl+C detected! Cleaning up...")
    solver.stopped = True
    elapsed = time.time() - solver.start_time
    print("Writing solutions to file...")
    write_solutions(solver.instance_filename, solver.solutions, [
   elapsed, solver.get_used_memory()], interruped=True)

    sys.exit(0)

signal.signal(signal.SIGINT, signal_handler)
```

Questo approccio rende l'esecuzione del programma più resiliente e user-friendly, consentendo all'utente di interrompere in qualsiasi momento senza perdere completamente il lavoro svolto fino a quel punto.

2.2.4 Classe `Solver`: gestione della computazione

Per gestire l'intero processo di calcolo degli MHS è stata definita la classe `Solver`, responsabile dell'organizzazione dei dati, della verifica delle soluzioni, del pre-processing della matrice e della gestione delle risorse computazionali.

Questa classe incapsula diverse funzionalità chiave e consente una gestione ordinata e modulare del flusso di esecuzione. Di seguito vengono descritti i metodi principali:

- `__init__`: inizializza l'istanza del solver, salvando la matrice di input, il nome dell'istanza, il livello corrente di esplorazione (`current_level`), e alcune variabili di controllo per il debug e la terminazione.

- `check_solution(solution)`: consente di verificare se una soluzione binaria passata come input rappresenta effettivamente un MHS. Lo fa combinando logicamente le colonne selezionate e confrontando il risultato con la copertura dell'intera matrice. Stampa un messaggio di conferma o errore in base all'esito.

  Questa funzione è cruciale poiché, nel contesto di un problema NP-completo come il *Minimum Hitting Set*, permette di verificare in tempo polinomiale la correttezza di una soluzione candidata, anche se la sua individuazione può richiedere tempo esponenziale. Questo riflette una caratteristica fondamentale dei problemi appartenenti alla classe NP: mentre trovare una soluzione può essere computazionalmente difficile, controllarne la validità è computazionalmente efficiente. Inoltre, la funzione è essenziale per validare le soluzioni generate da algoritmi euristici, approssimati o esaustivi, garantendo che esse soddisfino effettivamente i vincoli del problema.

- `parse_matrix()`: esegue una fase di pre-processing eliminando tutte le colonne della matrice che non contengono alcun elemento attivo (tutti zeri). Tali colonne vengono considerate irrilevanti per la computazione degli MHS e i loro indici vengono salvati in `deleted_columns_index` per poterli eventualmente reinserire a posteriori. Se il flag `debug` è attivo, viene stampata la matrice risultante e il numero di colonne rimosse.

- `add_deleted_columns_to_solution(solution)`: una volta completata la computazione degli MHS sulla matrice ridotta, questo metodo reinserisce le colonne precedentemente eliminate, impostandole a `False` (cioè non selezionate). In questo modo le soluzioni tornano ad avere la stessa dimensionalità della matrice originale.

- `get_used_memory()`: restituisce la quantità di memoria utilizzata durante l'esecuzione, sfruttando il modulo `tracemalloc` di Python. Viene usato anche per riportare statistiche dettagliate nei file di output.

- `permute_rows()` e `permute_columns()`: permettono di randomizzare rispettivamente l'ordine delle righe e delle colonne della matrice. Queste permutazioni possono essere utili per evitare bias nell'elaborazione e valutare la stabilità dell'algoritmo rispetto all'ordine dei dati in input.

Questa struttura a oggetti favorisce la riusabilità e la chiarezza del codice, semplificando l'estensione a funzionalità future, come l'ottimizzazione degli algoritmi o l'aggiunta di tecniche euristiche.

Un esempio di utilizzo del metodo `parse_matrix()` è mostrato di seguito. Il metodo scorre la matrice dalla colonna finale verso l'inizio e rimuove ogni colonna composta unicamente da valori falsi (cioè zeri nella rappresentazione binaria). Le colonne eliminate vengono salvate per un eventuale reinserimento successivo nelle soluzioni finali:

```
def parse_matrix(self):
    self.deleted_columns_index = []
    M = self.matrix.shape[1]
    for i in range(self.matrix.shape[1] - 1, 0, -1):
        column = self.matrix[:, i].reshape(-1)
        if np.sum(column) == 0:
            self.matrix = np.delete(self.matrix, i, 1)
            self.deleted_columns_index.append(i)

    if self.debug:
        print(f'Start columns: {M} - End columns: {self.matrix.shape[1]}')
        matrix_to_print = [[0 if x == False else 1 for x in row] for row in self.matrix]
        for row in matrix_to_print:
            print(row)
```

2.2.5 Classe `Node`: rappresentazione delle ipotesi

La classe `Node` è progettata per rappresentare una singola ipotesi all'interno dell'algoritmo di calcolo degli MHS. Ogni oggetto di questa classe corrisponde a una possibile combinazione di colonne (in formato booleano) della matrice di input e costituisce un nodo nello spazio di ricerca.

L'idea fondamentale alla base è che ogni ipotesi può essere descritta come un vettore binario in cui ogni bit indica se una colonna è selezionata (`True`) o meno (`False`). Questo approccio consente di trattare direttamente lo spazio delle soluzioni in modo compatto e facilmente comparabile.

Di seguito si riportano le funzionalità principali implementate nella classe:

- `__init__`: inizializza un nodo ipotesi con un valore booleano, calcolandone anche il livello (cioè la cardinalità dell'ipotesi, ovvero il numero di colonne selezionate). Il campo `vector` rappresenta la combinazione logica delle colonne scelte, ed è calcolato successivamente.
- `update_level()`: aggiorna il livello dell'ipotesi calcolando il numero di colonne attive nel vettore `value`.
- `_set_fields(matrix)`: inizializza il campo `vector` combinando logicamente le colonne della matrice corrispondenti ai bit attivi dell'ipotesi. Viene eseguita solo se il nodo si trova al primo livello (cioè con una sola colonna attiva); in caso contrario, viene inizializzato con uno zero-vettore.
- `lm()`: restituisce l'indice del primo elemento `True` (cioè la colonna più a sinistra selezionata).

- `propagate(h)`: propaga la copertura dell'ipotesi `h` sull'ipotesi corrente. In pratica, aggiorna il campo `vector` tramite un'operazione di OR logico.
- `initial(h, matrix)` e `final(h, matrix)`: generano rispettivamente una nuova ipotesi iniziale o finale partendo da un nodo `h`. Entrambe rimuovono una colonna attiva secondo un criterio prestabilito e ricreano un nuovo nodo aggiornato.
- `global_initial(matrix)`: genera un'ipotesi iniziale "globale" secondo le regole imposte dalla teoria.
- `distance(h)`: calcola la distanza Hamming tra l'ipotesi corrente e un'altra ipotesi `h`, cioè il numero di bit diversi tra le due.
- `is_solution()`: determina se l'ipotesi corrente rappresenta una soluzione valida, verificando se la copertura del vettore `vector` è completa (cioè copre tutte le righe della matrice).
- `__str__()`: restituisce una rappresentazione leggibile del vettore booleano `value`.
- `__eq__`, `__lt__`, `__gt__` e operatori comparativi: permettono il confronto tra ipotesi sulla base della loro rappresentazione binaria. Questo è utile per garantire ordinamenti consistenti all'interno delle strutture dati (ad esempio, set o heap).

Grazie a questa struttura, ogni nodo `Node` incapsula in maniera compatta sia l'informazione sull'ipotesi che la sua copertura logica rispetto alla matrice. Questo modello rende più semplice l'implementazione di strategie di esplorazione e pruning nello spazio delle soluzioni.

2.3 Algoritmo di calcolo degli MHS: scelte progettuali e ottimizzazioni

Il cuore del sistema è rappresentato dall'algoritmo implementato nel metodo `calculate_solutions`, che si occupa di esplorare lo spazio delle ipotesi e determinare tutti i Minimi Hitting Set (MHS) di una matrice binaria.

Approccio generale

L'algoritmo si basa su una strategia di generazione iterativa delle ipotesi, rappresentate tramite oggetti della classe `Node`. Ogni nodo corrisponde a una possibile combinazione di colonne selezionate della matrice. A partire da un'ipotesi iniziale vuota, vengono generati i figli attraverso il metodo `generate_children`, esplorando lo spazio delle soluzioni livello per livello (in ordine crescente di cardinalità).

La struttura segue una logica a livelli (similarmente a un BFS modificato) che garantisce l'individuazione delle soluzioni minime prima di quelle più grandi, e consente di tenere traccia delle ipotesi generate a ciascun livello attraverso il dizionario `hypoteses_per_level`.

Scelte strutturali per scalabilità e performance

1. Rappresentazione binaria compatta Le ipotesi vengono rappresentate come array booleani di dimensione M (numero di colonne della matrice), memorizzati tramite oggetti NumPy. Questa scelta permette di:

- Ottimizzare l'utilizzo della memoria rispetto a strutture come liste di interi.
- Sfruttare operazioni vettoriali e logiche ad alte prestazioni (`np.logical_or`, `np.sum`, ecc.).

2. Eliminazione preventiva di colonne inutili All'avvio viene effettuata un'analisi della matrice per rimuovere colonne che non partecipano ad alcuna copertura (ovvero colonne con tutti zeri). Questa riduzione dimensionale, effettuata in `parse_matrix()`, ha impatto diretto sulla scalabilità del sistema: una matrice con molte colonne "vuote" viene semplificata prima dell'esecuzione, riducendo lo spazio di ricerca.

3. Evitare la generazione di nodi ridondanti Nel metodo `generate_children` si evita la generazione di figli che non soddisfano i criteri minimi richiesti o che non introducono nuova informazione. Si verifica, ad esempio, che i nuovi figli siano compatibili con la struttura corrente dello spazio di ricerca (mediante le ipotesi `initial` e `final`) e si applicano filtri sulle distanze Hamming tra ipotesi per decidere se un'ipotesi è effettivamente "valida".

4. Gestione iterativa senza ricorsione La scelta di evitare chiamate ricorsive per la generazione dei figli e l'esplorazione dello spazio di ricerca consente di evitare stack overflow (max recursion limit) e permette una maggiore visibilità e controllo sull'esecuzione. Questo approccio è più robusto e si adatta meglio a computazioni di lunga durata su dataset ampi.

5. Controllo del consumo di memoria e tempo Per monitorare costantemente le risorse utilizzate, si impiegano le librerie `tracemalloc` e `time`, che consentono:

- Il tracciamento della memoria allocata in tempo reale.
- La misura del tempo di esecuzione globale del processo.
- La scrittura di statistiche dettagliate all'interno del file di output.

Queste informazioni vengono utilizzate anche per finalità diagnostiche e per comprendere il comportamento del sistema su diverse istanze.

6. Interruzione controllata e salvataggio stato parziale L'algoritmo supporta la terminazione anticipata (via `Ctrl+C`), catturando il segnale `SIGINT` e salvando lo stato corrente in un file di output parziale. Ciò permette di non perdere le soluzioni trovate fino a quel momento e offre maggiore flessibilità nelle esecuzioni batch.

Descrizione del ciclo principale

Nel metodo `calculate_solutions`, il ciclo principale si articola come segue:

1. Viene inizializzato il nodo vuoto `h0`, da cui parte l'esplorazione.
2. Per ciascun livello di cardinalità (memorizzato in `self.current_level`) vengono processate tutte le ipotesi attive.
3. Se un'ipotesi è valida (`is_solution()`), viene salvata.
4. Altrimenti, vengono generati i figli con `generate_children()`.
5. Le nuove ipotesi vengono ordinate in modo da garantire una visita coerente dello spazio binario.
6. Il ciclo termina quando non vi sono più ipotesi da esplorare o viene raggiunto il livello massimo.

Strutture dati e ottimizzazioni per la scalabilità Uno degli obiettivi principali dell'implementazione è stato massimizzare la scalabilità, cioè la capacità di elaborare matrici binarie anche di grandi dimensioni (sia in termini di righe che di colonne). A tal fine, è stata posta particolare attenzione alla scelta delle strutture dati, con lo scopo di ridurre l'overhead di memoria e accelerare le operazioni fondamentali nel ciclo di generazione delle soluzioni.

- **Array booleani NumPy:** ogni ipotesi viene rappresentata come un array booleano `np.array(dtype=bool)`. Rispetto a una lista di booleani nativa di Python, questa struttura:
  - occupa **1 byte per valore** a livello di rappresentazione (contro i 24 byte di un oggetto booleano Python), consentendo di mantenere migliaia di ipotesi in memoria simultaneamente;
  - permette operazioni vettoriali (ad esempio `np.logical_or`, `np.sum`, `np.logical_xor`) che sono implementate in C e ottimizzate internamente per uso su array di grandi dimensioni;
  - consente di evitare cicli espliciti in Python, che sono notoriamente costosi in termini di prestazioni.
- **Evita strutture complesse o ridondanti:** l'algoritmo non fa uso di alberi, grafi o strutture dati annidate, ma lavora con:
  - liste piatte (`list[Node]`) per mantenere lo stato attuale e il livello successivo;
  - dizionari leggeri (`dict[int, int]`) per tracciare il numero di ipotesi generate per livello;
  - array booleani per rappresentare le ipotesi stesse.

  Questa scelta è motivata dal fatto che strutture più sofisticate (es. alberi n-ari o heap) avrebbero introdotto costi extra di gestione e memoria per ogni nodo, non necessari nel contesto di una generazione livello-per-livello.

- **Rimozione colonne nulle a tempo zero:** durante il pre-processing, le colonne della matrice che non coprono alcun vincolo vengono rimosse (tramite `np.delete`), e i relativi indici vengono memorizzati in una lista. In questo modo, si riduce da subito la dimensione delle ipotesi da generare e lo spazio di ricerca viene ristretto in modo deterministico prima di iniziare. Questo accorgimento ha impatto diretto sulla complessità sia spaziale che temporale dell'algoritmo.
- **Memorizzazione compatta delle matrici:** la matrice di input viene mantenuta in formato NumPy nativo `dtype=bool`, che consente:
  - una rapida selezione di colonne e righe;
  - un uso minimo della memoria rispetto a strutture nested (liste di liste o DataFrame);
  - accesso diretto e slicing efficiente.
- **Vector caching:** in ogni nodo, il campo `vector` viene utilizzato come "accumulatore" per rappresentare la copertura dei vincoli garantita da quell'ipotesi. Questo evita di dover ricalcolare la copertura ogni volta da zero e consente una propagazione logica incrementale tra nodi genitori e figli.

Queste scelte rendono l'implementazione idonea a gestire matrici di dimensioni modeste, minimizzando l'allocazione di memoria ridondante e riducendo il numero di operazioni ad alta complessità. La vera criticità del problema non sta nell'utilizzo della memoria bensì risiede nella complessità dell'algoritmo, che cresce in modo esponenziale all'aumentare delle dimensioni dell'istanza.

3 Compito 2

3.1 Descrizione del problema

Il secondo compito prevede l'utilizzo dell'algoritmo sviluppato nel compito 1 per eseguire delle sperimentazioni, effettuando delle prove di funzionamento andando ad acquisire in ingresso delle matrici specificate in alcuni file di benchmark forniti dai docenti. Il fine principale delle sperimentazioni è quello di registrare e valutare le prestazioni spaziali e temporali delle prove condotte.

3.2 Implementazione

Alla classe `Node` (che ricordiamo essere un nodo dello spazio di ricerca ad albero rappresentante un ipotesi) sono state aggiunte un attributo (`start_time`) e un metodo (`get_used_memory`) utili a salvare l'occupazione di memoria e il tempo di esecuzione del processo che si occupa dell'esecuzione dell'algoritmo. Per migliorare l'utilizzo delle risorse sono stati implementati alcuni accorgimenti: il dominio è spesso un superinsieme dell'unione degli insiemi della collezione dei vincoli, quindi possiamo restringere il dominio da considerare nell'esplorazione andando a rimuovere tutte le variabili che non appaiono nemmeno una volta nei vincoli. Il metodo usato per rimuoverli è il seguente:

```
def parse_matrix(self):

    self.deleted_columns_index = []
    M = self.matrix.shape[1]
    for i in range(self.matrix.shape[1] - 1, 0, -1):
        column = self.matrix[:, i].reshape(-1)
        if np.sum(column) == 0:
            self.matrix = np.delete(self.matrix, i, 1)
            self.deleted_columns_index.append(i)
```

Con il codice specificato andiamo a rimuovere tutte le colonne della matrice che contengono solo 'zeri', salvando anche in una lista tutti gli indici rimossi; questo perché, nella generazione delle soluzioni, è necessario includere anche le colonne precedentemente troncate.

Un altro accorgimento implementato è stato sfruttare il fatto che la cardinalità dei MHS non può essere essere mai maggiore del valore $max\{|N|, |M'|\}$, dove N indica il numero di vincoli del problema e M' rappresenta la cardinalità del dominio (dopo essere stato ridotto con le modalità specificate precedentemente).

3.3 Benchmark e Metodologia di Sperimentazione

Per valutare le prestazioni dell'applicazione, sono stati utilizzati i benchmark forniti dai docenti, i quali contengono matrici di input di diversa dimensione e complessità. Ogni matrice è stata processata più volte per assicurare una misurazione affidabile dei tempi di esecuzione e dell'uso di memoria. Per ogni esperimento sono stati raccolti i seguenti dati:

- Tempo di esecuzione complessivo dell'algoritmo
- Memoria utilizzata al picco dell'esecuzione
- Numero di soluzioni trovate (MHS)

- Eventuali fallimenti o tempi limite

I dati raccolti sono stati poi confrontati tra loro per identificare eventuali pattern o anomalie legate alla struttura della matrice in ingresso.

3.4 Analisi dei Risultati

Le sperimentazioni condotte hanno messo in evidenza i seguenti aspetti significativi:

- Il tempo di esecuzione dell'algoritmo cresce in modo non lineare all'aumentare della dimensione della matrice, in particolare quando aumenta il numero di vincoli attivi. Questo conferma la natura computazionalmente onerosa del problema affrontato.
- L'ottimizzazione basata sulla rimozione delle colonne nulle ha prodotto una riduzione sostanziale dei tempi di esecuzione. Senza l'impiego di questa strategia di preprocessing, l'algoritmo non riesce a risolvere la maggior parte delle istanze entro tempi ragionevoli.
- L'utilizzo di memoria è rimasto contenuto nei casi più semplici, ma ha registrato picchi significativi per matrici più dense o complesse. Questo comportamento è attribuibile al numero elevato di nodi esplorati nello spazio di ricerca, che cresce rapidamente con la complessità dell'istanza.

La figura 1 mostra l'andamento del tempo di esecuzione dell'algoritmo al variare del numero di colonne (ovvero della dimensione del dominio effettivamente utilizzato, dopo la fase di preprocessing).

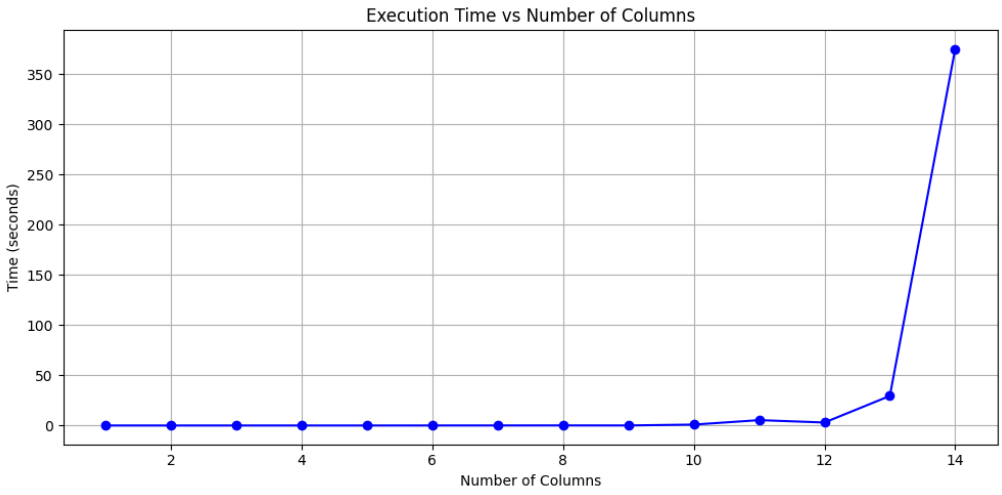

Figura 1: Tempo di esecuzione dell'algoritmo in funzione del numero di colonne considerate.

Dal grafico si evidenziano chiaramente i seguenti aspetti:

- Per domini con meno di 10 colonne, l'algoritmo restituisce una soluzione in pochi secondi, mostrando un tempo di esecuzione pressoché costante.
- A partire da 11-12 colonne, il tempo di calcolo inizia a crescere sensibilmente, suggerendo l'insorgere di una complessità computazionale di tipo esponenziale.

- Con 14 colonne, il tempo di esecuzione supera i 360 secondi, indicando una soglia critica oltre la quale l'approccio adottato risulta poco efficiente senza ulteriori ottimizzazioni.

4 Compito 3

4.1 Descrizione del problema

Il terzo obiettivo del progetto consiste nello sviluppo di un modulo software che applichi permutazioni alla matrice di input e analizzi l'effetto di tali trasformazioni sulle prestazioni dell'algoritmo implementato nel primo compito. In particolare, si intende valutare se e in che misura le permutazioni influenzino i risultati, individuando eventuali strategie di permutazione in grado di migliorare l'efficienza computazionale.

4.2 Implementazione

In una fase iniziale, sono stati sviluppati metodi per permutare casualmente sia le righe sia le colonne della matrice, al fine di valutare l'impatto generale di tali trasformazioni. A tal fine, sono stati integrati nella classe `Solver` due nuovi metodi:

- `permute_rows`: applica una permutazione casuale delle righe.
- `permute_columns`: applica una permutazione delle colonne, con supporto a diverse strategie.

4.2.1 Analisi preliminare dei risultati

L'algoritmo è stato testato su tutte le matrici contenute nella cartella `benchmarks1`, imponendo un limite massimo di 16 colonne (dopo l'eliminazione di quelle nulle e di quelle duplicate) e un tempo massimo di esecuzione di 120 secondi per ciascun test. Sono stati considerati quattro scenari distinti:

1. Nessuna permutazione.
2. Permutazione casuale delle righe.
3. Permutazione casuale delle colonne.
4. Permutazione casuale di righe e colonne.

Per ogni scenario, i risultati ottenuti sono stati confrontati con quelli della configurazione originale per verificare la correttezza delle soluzioni. In particolare, si è controllato che:

- Il numero di soluzioni ottenute fosse invariato.
- Le soluzioni coincidessero bit a bit (confronto binario).

Il codice per fare questo controllo si trova nel file *perm_result_check.py*
In caso di corrispondenza, il test è stato considerato superato; diversamente, è stato segnalato come non consistente. È importante sottolineare che nessuna esecuzione ha prodotto risultati errati, in quanto le permutazioni di righe e colonne non alterano le soluzioni trovate, bensì hanno influenza soltanto sulle performance di esecuzione dell'algoritmo.
Dall'analisi delle esecuzioni con input permutato è emerso che:

- La permutazione delle righe non ha prodotto impatti significativi sulle prestazioni.

- La permutazione delle colonne ha, invece, evidenziato differenze rilevanti in numerosi casi, migliorando in modo consistente i tempi di esecuzione.

A seguito di questa osservazione, l'attenzione è stata focalizzata sulle strategie di permutazione delle colonne.

4.2.2 Metodo `permute_columns`

È stato quindi progettato un metodo flessibile per permutare le colonne secondo tre strategie distinte:

- **Permutazione casuale**: le colonne vengono rimescolate utilizzando una permutazione casuale.
- **Permutazione decrescente**: le colonne vengono ordinate in base al numero di elementi uguali a 1 in ordine decrescente, con l'obiettivo di raggruppare le colonne più dense nella parte iniziale della matrice.
- **Permutazione crescente**: le colonne vengono ordinate in modo crescente in base al numero di 1, posizionando le colonne meno dense all'inizio.

Il comportamento del metodo è controllato da due attributi:

- `randomize`: determina se applicare una permutazione casuale.
- `decrescent`: indica se ordinare le colonne in modo crescente o decrescente.

4.2.3 Valutazione delle strategie di permutazione

Successivamente, per valutare l'impatto delle diverse permutazioni di colonna, sono stati eseguiti ulteriori test mirati. In particolare, sono state analizzate le performance dell'algoritmo nei seguenti casi:

- Generazione di tutti i MHS.
- Ricerca del primo e del secondo MHS.

Anche in questa fase, i test sono stati eseguiti sui file della cartella `benchmarks1`, mantenendo i limiti di 16 colonne (senza considerare le colonne interamente composte da zeri e le colonne duplicate) e un tempo massimo di 120 secondi per ciascuna esecuzione.

I risultati ottenuti sono illustrati nella sezione successiva, con grafici e tabelle che evidenziano l'efficacia delle varie strategie di permutazione nell'ottimizzazione dell'algoritmo di hitting set minimo.

4.2.4 Risultati sperimentali

Enumerazione completa delle soluzioni Sono stati eseguiti test per l'enumerazione di tutti i MHS, con un limite di 120 secondi e un massimo di 16 colonne, considerando quattro strategie: nessuna permutazione, permutazione casuale, ordinamento crescente e decrescente delle colonne.

- I tempi medi di esecuzione mostrano un leggero miglioramento nel caso di permutazione crescente delle colonne (3.35 s), rispetto alla configurazione senza permutazioni (3.41 s).

- Anche il consumo medio di memoria presenta un lieve decremento, passando da 9102.96 KB (senza permutazione) a 9093.08 KB (crescente).

Strategia	**Tempo medio (s)**	**Variazione %**	**Memoria (KB)**	**Variazione %**
No Permutation	3.41	0.00%	9102.96	0.00%
Random	3.41	-0.10%	9102.48	-0.01%
Decrescent	3.39	-0.49%	9094.68	-0.09%
Crescent	**3.35**	**-1.71%**	**9093.08**	**-0.11%**

Tabella 1: Tempi e memorie medie per l'enumerazione completa

I risultati indicano che l'ordinamento crescente delle colonne, sebbene con variazioni marginali, può contribuire a un miglioramento complessivo dell'efficienza durante l'enumerazione completa.

Ricerca della prima e seconda soluzione È stata successivamente valutata l'efficacia delle strategie di permutazione nel caso in cui si desideri individuare solo i primi due MHS, imponendo un limite di 16 colonne e di 30 secondi per caso. I risultati sono stati i seguenti:

- La strategia decrescente ha mostrato un miglioramento significativo nelle performance:
  - Tempo medio per la prima soluzione: 0.76 s (contro 1.40 s nel caso normale, +84%).
  - Tempo medio per la seconda soluzione: 0.88 s (contro 1.86 s nel caso normale, +110%).
- Al contrario, la strategia crescente ha peggiorato notevolmente le performance nel recupero delle prime soluzioni.

Strategia	**1ª Soluzione (s)**	**Var. %**	**2ª Soluzione (s)**	**Var. %**
Normal	1.403	+84.20%	1.869	+110.73%
Random	1.095	+43.75%	1.248	+40.72%
Decrescent	**0.762**	**0.00%**	**0.887**	**0.00%**
Crescent	2.306	+202.75%	2.458	+177.16%

Tabella 2: Tempi medi per la prima e seconda soluzione

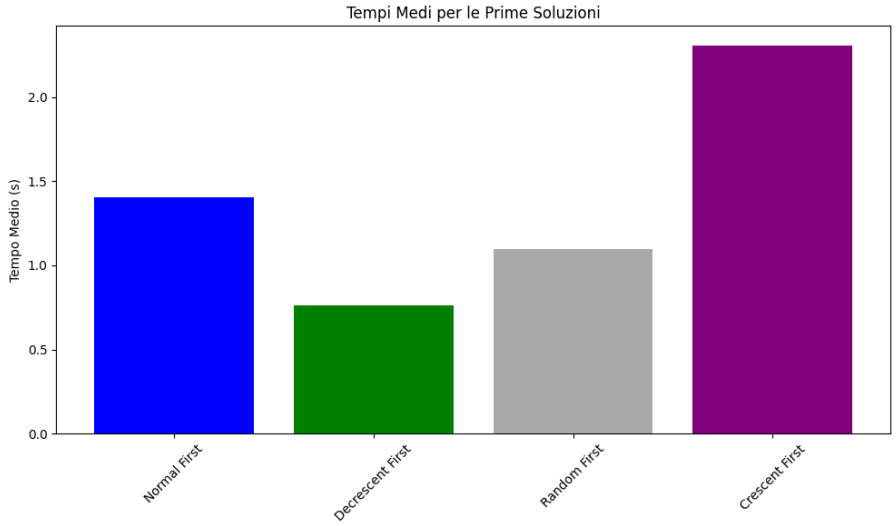

Figura 2: Tempi medi per la prima soluzione nei vari casi

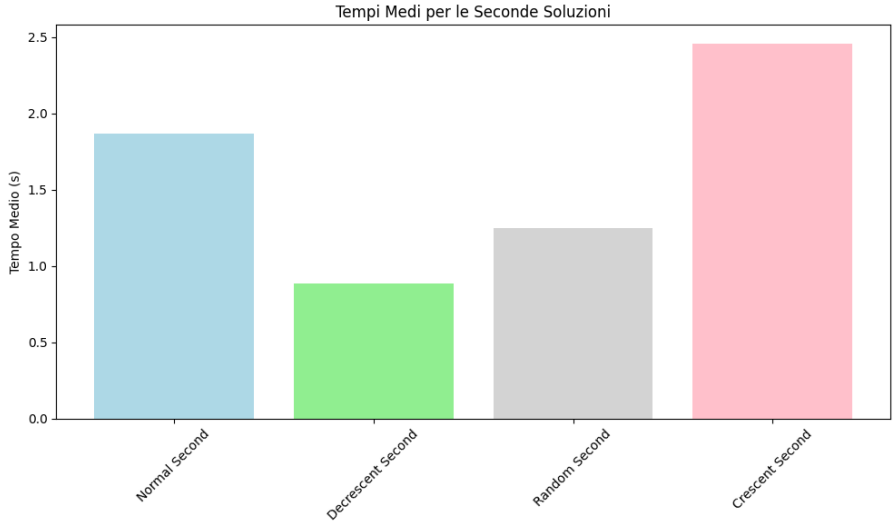

Figura 3: Tempi medi per la seconda soluzione nei vari casi

4.3 Analisi dei risultati

L'analisi sperimentale condotta ha evidenziato come l'ordine delle colonne nella matrice di input possa influenzare in modo significativo le prestazioni dell'algoritmo di generazione degli hitting set minimi, pur mantenendo inalterata la correttezza dei risultati.

In particolare, sono emerse due tendenze distinte:

- **Enumerazione completa**: nell'ambito della generazione di tutte le soluzioni (MHS), l'ordinamento crescente delle colonne — ovvero la disposizione delle colonne meno dense nella parte iniziale della matrice — ha prodotto lievi miglioramenti in termini di tempo di esecuzione e utilizzo della memoria. Sebbene le variazioni siano contenute, questo approccio potrebbe comunque rappresentare una strategia valida in contesti in cui è necessario ottimizzare l'efficienza su grandi istanze del problema.

- **Ricerca delle prime soluzioni**: in scenari in cui l'obiettivo è individuare rapidamente una o poche soluzioni (ad esempio per applicazioni interattive o in pipeline computazionali), l'ordinamento decrescente delle colonne si è dimostrato nettamente più efficace. Questo tipo di permutazione, che colloca le colonne più dense all'inizio, ha comportato una riduzione significativa dei tempi di calcolo — circa dimezzato rispetto alla configurazione originale — rendendola una strategia fortemente raccomandata in casi simili.

È importante sottolineare che tutte le strategie analizzate preservano la correttezza del risultato: le permutazioni applicate, agendo unicamente sull'ordine delle righe e delle colonne, non modificano la struttura logica del problema, ma ne influenzano esclusivamente l'efficienza computazionale. Nessuna delle esecuzioni effettuate ha infatti prodotto soluzioni errate o inconsistenti.

Questi risultati suggeriscono che un'analisi preliminare della struttura della matrice di input potrebbe guidare la scelta di una permutazione iniziale più favorevole all'algoritmo, aprendo la strada all'integrazione di tecniche euristiche o adattive nella fase di pre-processing.

In prospettiva, ulteriori approfondimenti potrebbero includere:

- l'esplorazione di strategie di permutazione basate su metriche più complesse.
- valutazione dell'impatto su classi di istanze più ampie e diversificate.

Questa fase del progetto ha quindi dimostrato come anche semplici trasformazioni dell'input, se ben progettate, possano contribuire in modo rilevante all'ottimizzazione del tempo di esecuzione dell'algoritmo di MHS.

5 Ulteriori migliorie

5.1 Rimozione colonne duplicate

Come discusso in precedenza, un primo miglioramento delle prestazioni è stato ottenuto tramite la rimozione delle colonne contenenti unicamente zeri, in quanto esse non contribuiscono in alcun modo alla generazione di soluzioni valide.

Un'ulteriore ottimizzazione può essere raggiunta tramite l'eliminazione delle colonne duplicate, ovvero colonne che presentano la stessa configurazione binaria. Tuttavia, questa operazione risulta più delicata rispetto alla precedente: rimuovendo colonne duplicate, infatti, si rischia di escludere temporaneamente delle possibili soluzioni, che dovranno essere successivamente ricostruite o riconsiderate in una fase successiva del processo di calcolo.

Per implementare questa strategia, è stato sviluppato il seguente metodo:

```
def remove_same_columns(self):
    counter = {}
    self.duplication_list = {}
    for i in range(self.matrix.shape[1]):
        column = self.matrix[:, i].reshape(-1)
        column_str = ''.join('1' if x else '0' for x in column)

        if column_str not in counter:
            counter[column_str] = []

        counter[column_str].append(i)

    tmp = []
    for key in counter:
        if len(counter[key]) <= 1:
            continue
        x = counter[key][1:]
        tmp.extend(x)
        self.duplication_list[counter[key][0]] = x
    tmp.sort()

    for i in range(len(tmp) - 1, -1, -1):
        self.matrix = np.delete(self.matrix, tmp[i], 1)

    return self.matrix.shape[1]
```

Il metodo `remove_same_columns` agisce nel modo seguente:

- Per ogni colonna della matrice, viene generata una rappresentazione binaria come stringa (`column_str`).
- Le colonne vengono raggruppate in base alla loro rappresentazione: quelle che condividono la stessa stringa sono considerate duplicate.
- Per ciascun gruppo di colonne duplicate, viene mantenuta solo la prima, mentre le altre vengono rimosse dalla matrice.
- Le informazioni sulle colonne eliminate vengono salvate nella struttura `duplication_list`, utile per un eventuale ripristino delle colonne originali in fase di post-processing.

Questo approccio consente di ridurre la dimensione del problema da risolvere, migliorando l'efficienza computazionale, pur richiedendo una gestione attenta delle informazioni rimosse per garantire la correttezza delle soluzioni finali.

5.2 Ripristino delle colonne duplicate e generazione delle soluzioni equivalenti

Dopo aver ottenuto una o più soluzioni sulla matrice ridotta (ovvero, priva delle colonne duplicate), è necessario ricostruire le soluzioni originali includendo anche le colonne precedentemente rimosse. A tale scopo viene utilizzata la struttura `duplication_list`, costruita in fase di pre-processing.

I seguenti frammenti di codice mostrano come vengono reinserite le colonne eliminate e come si generano le possibili combinazioni equivalenti di soluzioni:

```
tmp = []
for key in self.duplication_list:
    tmp.extend(self.duplication_list[key])
tmp.sort()

for i in tmp:
    solution = np.insert(solution, i, [False], axis=1)
```

In questa prima fase, per ogni colonna eliminata, viene reinserita una colonna di zeri (`False`) nella posizione originaria, ripristinando così la dimensione originale della soluzione.

Successivamente, vengono identificate le colonne attivate (`True`) nella prima riga della soluzione (che corrisponde a una delle soluzioni trovate):

```
original_indexes = []
for i in range(len(solution[0])):
    if solution[0][i]:
        original_indexes.append(i)
```

A questo punto, si costruisce una lista di insiemi di indici che rappresentano tutte le possibili combinazioni delle colonne originarie, tenendo conto delle colonne duplicate:

```
arr = []
for idx in range(len(solution[0])):
    if solution[0][idx] and idx in self.duplication_list:
        indexes = []
        indexes.append(idx)
        indexes.extend(self.duplication_list[idx])
        arr.append(indexes)
    elif solution[0][idx] and idx not in self.duplication_list:
        arr.append([idx])
```

Per ogni colonna attiva nella soluzione, se essa ha colonne duplicate, viene generato un insieme comprendente la colonna originale e tutte le sue copie. Viene quindi costruito il prodotto cartesiano tra questi insiemi per generare tutte le combinazioni possibili di colonne attive equivalenti:

```
from itertools import product
combinations = list(product(*arr))

combinations.remove(tuple(original_indexes))
```

Infine, ogni combinazione viene trasformata in una nuova riga della soluzione, mantenendo inalterata la semantica del problema:

```
for combination in combinations:
    new_row = np.zeros(len(solution[0]), dtype=bool)
    for i in range(len(combination)):
        new_row[combination[i]] = True
    solution = np.vstack([solution, new_row])
```

Questo passaggio assicura che tutte le soluzioni logicamente equivalenti (ossia, che differiscono solo per l'utilizzo di colonne duplicate) vengano correttamente ricostruite e aggiunte alla soluzione finale. In caso di debug attivo, viene anche stampato il numero di soluzioni generate:

```
if self.option.debug:
    print(f'Added {len(combinations)} new solutions by combining
    the duplicated columns')
```

Questo approccio permette di bilanciare l'efficienza del pre-processing con l'accuratezza delle soluzioni finali, assicurando che nessuna combinazione valida venga persa a causa della rimozione delle colonne duplicate.

5.3 Risultati ottenuti

Definizione dei metodi `dz` e `dz_dd`

Nel contesto di questa analisi, vengono confrontati due approcci di pre-elaborazione delle matrici: `DZ` e `DZ_DD`. Il metodo `dz` si occupa esclusivamente della rimozione delle **colonne composte interamente da zeri**. Al contrario, il metodo `dz_dd` elimina sia le **colonne nulle** che le **colonne duplicate**, ottenendo una compressione più efficace.

L'analisi dei risultati sperimentali evidenzia la superiorità del metodo `DZ_DD` rispetto a `DZ` sotto il profilo dell'efficienza. Nelle istanze completate con successo, `DZ_DD` richiede in media meno tempo e meno memoria, dimostrandosi più leggero in termini di risorse computazionali. Inoltre, il numero di istanze terminate nei limiti imposti è maggiore per `DZ_DD`, indicando una migliore capacità di gestione dei casi complessi.

È importante sottolineare che le istanze non completate non sono attribuibili a errori interni ai metodi, ma esclusivamente al superamento dei vincoli imposti per l'esecuzione. In particolare, entrambi i metodi sono stati testati con gli stessi limiti:

- `TIME_LIMIT = 120` secondi
- `MAX_COLUMNS = 16`

Oltre a questi, è stato introdotto un ulteriore vincolo di sicurezza per limitare la complessità computazionale legata al numero di combinazioni da esplorare durante il ripristino delle righe a partire dalle colonne eliminate. Il processo viene interrotto preventivamente nel caso in cui il numero di combinazioni superi il valore soglia di un milione (`1,000,000`). Il seguente frammento di codice evidenzia tale meccanismo di interruzione:

```
num_of_combinations = 1
for element in arr:
    num_of_combinations *= len(element)

if num_of_combinations > 1000000:
    print(f'Warning: number of combinations is too high ({
    num_of_combinations}), skipping this step.')
    self.stopped = True
```

Questa precauzione si è resa necessaria per evitare un'esplosione combinatoria potenzialmente ingestibile in termini di tempo e memoria.

Tali evidenze rendono *DZ_DD* una scelta preferibile nell'ambito delle strategie di pre-elaborazione analizzate. Di seguito sono riportati alcuni dati aggregati, ricavati dalle esecuzioni del metodo sulle diverse istanze del problema.

Metodo	Completion Rate
DZ	4.76%
DZ_DD	17.43%

Tabella 3: Percentuale di istanze terminate nei limiti imposti per ciascun metodo.

Metodo	Tempo Medio (s)	Numero Istanze
DZ	5.27	118
DZ_DD	3.48	432

Tabella 4: Statistiche sulle istanze completate correttamente: tempo medio per ciascun metodo, con relativo numero di casi.

Parametro	Valore	Note
Numero totale di file analizzati	2478	
Colonne dopo eliminazione zeri		
Media numero colonne dopo `dz`	296.45	
Media numero colonne dopo `dz_dd`	54.51	
Riduzione media	443.81%	
Esecuzioni non effettuate (superamento vincoli)		
File non eseguiti con `dz`	2360	
File non eseguiti con `dz_dd`	2046	
File eseguiti solo da `dz_dd`	314	*(con stessi vincoli)*
Tempi medi di esecuzione		
Tempo medio con `dz`	5.27 s	
Tempo medio con `dz_dd`	0.10 s	(solo per file eseguiti con `dz`)
Speedup medio	4935.66%	
Memoria media utilizzata		
Memoria media con `dz`	138.64 KB	
Memoria media con `dz_dd`	25.97 KB	(solo per file eseguiti con `dz`)
Riduzione memoria	433.84%	

Tabella 5: Confronto tra le strategie `dz` e `dz_dd` in termini di riduzione dimensionale e prestazioni

6 Gestione parametrica dell'esecuzione

L'adozione di un approccio parametrico per la gestione dell'esecuzione consente di aumentare significativamente la flessibilità e la riusabilità del sistema. In particolare, il controllo tramite opzioni a riga di comando permette di adattare dinamicamente il comportamento dell'algoritmo a seconda delle caratteristiche delle istanze da risolvere, senza dover modificare il codice sorgente.

Questo approccio è risultato particolarmente vantaggioso nelle fasi di test e di esecuzioni multiple in quanto è aperto ai possibili scenari:

- **Esecuzione batch**: consente di eseguire automaticamente grandi quantità di istanze con configurazioni differenti, facilitando l'automazione di campagne sperimentali.
- **Testing e debugging mirato**: attivando la modalità `debug` è possibile ottenere un output dettagliato sull'elaborazione di specifiche istanze, utile per l'analisi del comportamento interno dell'algoritmo.
- **Sperimentazione di strategie**: opzioni come la permutazione delle righe o colonne permettono di confrontare diverse strategie di ordinamento e preprocessing, valutandone l'impatto sul tempo di esecuzione e sulla memoria utilizzata.
- **Controllo dei tempi computazionali**: l'opzione `max_time` consente di impostare un limite massimo di tempo per ciascuna esecuzione, evitando blocchi su istanze particolarmente complesse e migliorando la robustezza del sistema.

6.1 Struttura delle opzioni

La classe `Option` rappresenta un contenitore per i parametri di configurazione attivi durante l'esecuzione. Essa raccoglie flag booleani per attivare o disattivare funzionalità specifiche, tra cui:

- `debug`: attiva la stampa di informazioni utili per il debugging.
- `delete_zeros`: consente di rimuovere le colonne composte interamente da zeri, ottimizzando la dimensione della matrice.
- `delete_duplicates`: permette di eliminare colonne duplicate, riducendo la ridondanza e semplificando la ricerca.
- `max_time`: imposta un tempo massimo di esecuzione (in secondi), utile per limitare la computazione su istanze particolarmente complesse.
- `permute_rows`, `permute_columns`, `permute_columns_asc`, `permute_columns_desc`: attivano il riordinamento delle righe o colonne secondo criteri specifici, potenzialmente utili a migliorare l'efficienza dell'esplorazione dello spazio di ricerca.

6.2 Parsing degli argomenti

La funzione `handle_menu(args)` si occupa del parsing degli argomenti passati da linea di comando. Riconosce i flag definiti e li traduce in un oggetto `Option`, che viene poi passato al `Solver`. In particolare:

- I flag vengono interpretati e mappati nei corrispondenti attributi booleani.
- In caso di opzione `-m`, il tempo massimo viene letto come parametro numerico aggiuntivo.
- Vengono distinti i file di input (nomi dei file da elaborare) dai flag di controllo.

6.3 Utilizzo delle opzioni nel Solver

All'interno della classe `Solver`, l'oggetto `Option` viene utilizzato per controllare le fasi di preprocessing e output. Ad esempio, nella funzione `parse_matrix()`, vengono eseguite operazioni come la rimozione di colonne vuote o duplicate solo se i relativi flag sono attivi. Inoltre, se è attiva la modalità debug, la matrice preprocessata viene stampata a console.

Questa struttura modulare e parametrica consente di adattare dinamicamente il comportamento dell'algoritmo in base alle esigenze dell'utente o alla complessità delle istanze da risolvere, rendendo il sistema flessibile e facilmente estendibile.

7 Guida all'uso

7.1 Preparazione dell'ambiente

Per evitare conflitti tra diverse versioni delle librerie Python, è consigliato creare un *virtual environment*. Per creare e attivare un ambiente virtuale, seguire questi passaggi:

1. Creare l'ambiente virtuale lanciando il seguente comando:

   ```
   python -m venv nome_ambiente
   ```

   Questo comando creerà una nuova cartella chiamata `nome_ambiente` (scegliere un nome significativo per l'ambiente).

2. Attivare l'ambiente virtuale. Il comando per l'attivazione dipende dal sistema operativo:

   Su Linux/macOS:

   ```
   source nome_ambiente/bin/activate
   ```

   Su Windows:

   ```
   .\nome_ambiente\Scripts\activate
   ```

   Quando l'ambiente virtuale è attivo, il nome dell'ambiente apparirà nel terminale (ad esempio, `(nome_ambiente)`).

3. Una volta attivato l'ambiente, è possibile installare le librerie necessarie utilizzando il file `requirements.txt` come descritto nella sezione precedente.

   ```
   pip install -r requirements.txt
   ```

Per disattivare l'ambiente virtuale, eseguire il comando:

```
deactivate
```

L'ambiente virtuale può essere riattivato in qualsiasi momento seguendo i passaggi sopra descritti.

7.2 Esecuzione del file `main.py` con parametri

Il file `main.py` accetta una serie di parametri da riga di comando per modificare il comportamento del programma. Di seguito viene riportata una guida ai vari comandi disponibili e al loro utilizzo.

Sintassi generale

```
python main.py [opzioni] file1 file2 ...
```

Opzioni disponibili

- `-h` Mostra l'elenco dei comandi disponibili.
- `-v` Abilita l'output verboso.
- `-dz` Elimina righe contenenti solo zeri.
- `-dd` Elimina righe duplicate.
- `-pr` Permuta le righe in ordine casuale.
- `-pcr` Permuta le colonne in ordine casuale.
- `-pcd` Permuta le colonne in ordine decrescente.
- `-pcc` Permuta le colonne in ordine crescente.
- `-m <secondi>` Imposta il tempo massimo di esecuzione.

Esempi di utilizzo

- Mostrare la guida:

```
python main.py -h
```

- Eseguire in modalità debug:

```
python main.py -v input1.matrix input2.matrix
```

- Eliminare righe con zeri e permutare le righe:

```
python main.py -dz -pr dataset.matrix
```

- Impostare tempo massimo di esecuzione a 60 secondi:

```
python main.py -m 60 input.matrix
```

- Combinazione di più opzioni:

```
python main.py -dd -pcr -m 120 dati.matrix
```

Nota: L'ordine delle opzioni non è importante, ma se si utilizza `-m`, il numero che specifica i secondi deve seguire immediatamente il parametro.
Nota: Se non viene fornito alcun file da riga di comando il programma diventa interattivo e viene chiesto all'utente se desidera eseguire tutti i benchmark disponibili. Se l'utente risponde `y`, il programma esegue tutti i file presenti in entrambe le directory [benchmarks1, benchmarks2]. Se l'utente risponde `n`, il programma chiede di inserire il nome di un file da eseguire. Negli altri casi il programma termina senza eseguire nulla.