UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

Università degli Studi di Brescia
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
Corso di Laurea Magistrale in Ingegneria Informatica

Progetto Minimal Hitting Set

Mastroianni Axel

Contents

Chapter 1

Introduzione Al Problema

1.1 Problema

Il problema da affrontare in questo elaborato consiste nel calcolo dei minimum hitting sets (MHS) a partire da una collezione finita di insiemi denominata N i cui elementi stanno nel dominio M

Andiamo quindi per ordine e vediamo cosa si intende per minimum hitting set

1.1.1 Minimal Hitting Set

Per **hitting set** si intende un insieme di elementi appartenenti a M che presenta una intersezione non vuota con ciascuno degli insiemi in N

Se ciascun sottoninsieme di un HS non è un HS allora si parla di **minimal hitting set**

1.1.2 Esempio

Consideriamo:

1. N = {{B1, B3}, {A1, A2, B4}, {A2, A5, B3, B4}}
2. M = {A1, A2, A5, B3, B4}

Avremo che MHS = {{B4}, {A1, B3}, {A2, B3}}

Infatti si può notare che ognuno di questi sottoinsiemi di M interseca ciascun elemento di N e se si prende un sottoinsieme quest'ultimo non è un MHS

Avuta quindi una panoramica del problema iniziamo a vedere come sono fatti i dati di input ad alto livello, siccome ancora non è stato scelto alcun linguaggio di programmazione

1.2 Input

In input al problema viene fornita una matrice le cui colonne sono in numero pari al numero di elementi di M mentre le righe sono in numero pari al numero di elementi di N
L'elemento i,j di tale matrice vale 1 se il j-esimo elemento di M sta nell'i-esimo insieme N, altrimenti vale 0

La figura 1.1 riporta un esempio

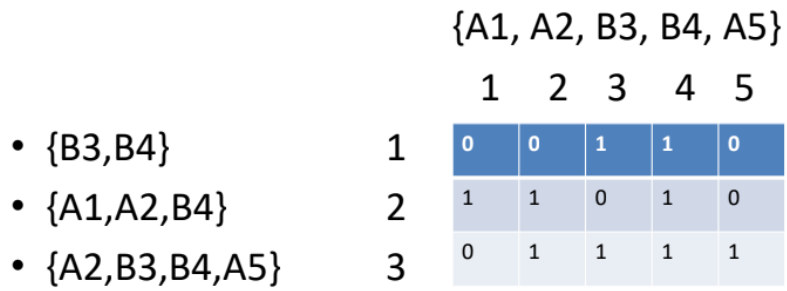

Figure 1.1: Esempio Matrice di Input

1.3 Scelta Del Linguaggio Di Programmazione

L'ultimo step prima di approcciare il problema a livello implementativo è ovviamente la scelta di un liguaggio di programmazione da utilizzare

Per svolgere questo lavoro ho deciso di optare per **Python**

Le ragioni di questa scelta sono principalmente di natura pratica siccome ho individuato i seguenti vantaggi nell'utilizzo di tale linguaggio:

1. Velocità e Facilità di implementazione
2. Maggiore chiarezza del codice
3. Possibilità di utilizzare notebook che permettono il test riga per riga favorendo quindi uno sviluppo incrementale dell'algoritmo senza necessità di una struttura base da cui partire
4. Facilità di manipolazione delle matrici grazie alla libreria NumPy

Ovviamente il suo utilizzo comporta l'ovvio svantaggio dell'inefficienza siccome Python è un linguaggio interpretato, rispetto a Java o C che sono indubbiamente più efficienti

Ho tuttavia deciso di sorvolare questa obiezione per due motivi (per lo più Machiavellici):

1. Il problema di decisione da affrontare è NP-Hard per cui avere una maggiore efficienza non aiuta di certo a diminuire la sua complessità ma solamente a risolvere istanze più grandi cosa che non ho reputato sufficiente per abbracciare una soluzione con un liguaggio più efficiente ma che richiede implementazioni più elaborate rispetto a Python
2. L'algoritmo sviluppato non deve partecipare ad alcuna competizione né essere utilizzato in applicazioni critiche dal punto di vista delle performance né tantomeno supportare un progetto di ricerca; è un esercizio da svolgere e quindi ancora una volta mi sono sentito libero di adottare il linguaggio che mi complicasse le cose il meno possibile

Chapter 2

Preparazione delle Istanze

2.1 Leggere la Matrice di Input

In questa sezione vedremo come viene letto il file contenente la matrice e come viene creata una matrice adatta all'algoritmo. In particolare saranno trattate le seguenti 3 funzioni:

1. ReadMatrix
2. InitializeStringMatrix
3. InitializeOnesMatrix

2.1.1 Funzione ReadMatrix

In figura 2.1 è presentato il codice della funzione

```
def read_matrix(file_name):
  with open(file_name, 'r') as f:
    lines = f.readlines()
  string_mat = initialize_string_matrix(lines)
  matrix = initialize_ones_matrix(string_mat)
  return np.array(matrix)
```

Figure 2.1: Lettura Matrice

Possiamo vedere come non faccia altro che leggere il file che contiene la matrice, riga per riga con la funzione "readlines()" e successivamente effettua due chiamate ad altre due procedure in modo da avere il dato pronto per essere utilizzato come input all'algoritmo

2.1.2 Funzione InitializeStringMatrix

Il codice è mostrato in figura 2.2 Tale procedura serve per estrarre la parte

```
def initialize_string_matrix(lines):
  string_mat = []
  for line in lines:
    if line[0] == '0' or line[0] == '1':
      string_mat.append(line[:-3])
  return string_mat
```

Figure 2.2: Funzione InizializeStringMatrix

del file che contiene la matrice, come suggerito dai check sul primo elemento della variabile string line
Se tale carattere è uno 0 oppure un 1 allora all'array di stringhe stringMat viene aggiunta in coda la variabile line
Il risultato è quindi un array con tanti elementi quante sono le righe della matrice di input ma questi elementi sono stringhe, non altri array
Qui entra in gioco la prossima funzione

2.1.3 Funzione InitializeOnesMatrix

Il codice è riportato in figura 2.3 In questa procedura vengono iterate tutte le stringhe dell'array creato con la funzione precedente e per ognuna di esse si itera lungo tutti i suoi caratteri. Siccome le stringhe possiedono caratteri 1 o 0 in posizione pari e spazi bianchi in posizione dispari viene effettuato il controllo sulla posizione letta che deve essere pari
Se tale condizione è soddisfatta si aggiunge in coda all'array row il cast in intero ('0'-> numero 0) del carattere letto
Finito di iterare su tutti i caratteri l'array row è aggiunto ad un'altro array, matrix

```
def initialize_ones_matrix(string_mat):
  matrix = []
  for string_row in string_mat:
    row = []
    for i in range(len(string_row)):
      if i % 2 == 0:
        row.append(int(string_row[i]))
    matrix.append(row)
  return matrix
```

Figure 2.3: Funzione InizializeOnesMatrix

In questo modo viene completata la procedura di creazione della matrice di input le cui colonne sono elementi di M e le cui righe sono gli insiemi di N

Chapter 3

Algoritmo MBase

3.1 Preparazione All'Algoritmo

3.1.1 Creazione di M

Data la matrice, l'insieme è costruito semplicemente con un numero di elementi pari alle colonne della matrice con valore "z + indice della colonna j-esima" come mostrato in figura 3.1

```
array(['z1', 'z2', 'z3', 'z4', 'z5', 'z6', 'z7', 'z8', 'z9', 'z10', 'z11',
       'z12', 'z13', 'z14', 'z15', 'z16', 'z17', 'z18', 'z19', 'z20',
       'z21', 'z22', 'z23', 'z24', 'z25', 'z26', 'z27', 'z28', 'z29',
       'z30', 'z31', 'z32', 'z33', 'z34', 'z35', 'z36', 'z37', 'z38',
       'z39', 'z40', 'z41', 'z42', 'z43', 'z44', 'z45', 'z46', 'z47',
       'z48', 'z49', 'z50', 'z51', 'z52', 'z53', 'z54', 'z55', 'z56',
       'z57', 'z58', 'z59', 'z60', 'z61', 'z62', 'z63', 'z64', 'z65',
       'z66'], dtype='<U3')
```

Figure 3.1: Insieme M

3.1.2 Sets Dict

Questa hash table contiene come chiave un elemento di M e come valore una keyword che indica se la colonna relativa a quell'elemento è:

1. ANALYZE: la colonna possiede sia 0 che 1

2. MHS: la colonna della matrice contiene solo 1
3. KO: la colonna della matrice contiene solo 0

Nelle due figure 3.2 e 3.3 vengono mostrate la funzione che crea tale struttura dati e un esempio di quest'ultima

```
def initialize_sets_dict(matrix, M):
  sets_dict = {}
  for i in range(matrix.shape[1]):
    col = np.unique(matrix[:, i])
    if len(col) > 1:
      sets_dict[M[i]] = ANALYZE
    elif col == 1:
      sets_dict[M[i]] = MHS
    elif col == 0:
      sets_dict[M[i]] = KO
  return sets_dict
```

Figure 3.2: Funzione Per Creare SetsDict

```
{'z1': 'KO',
 'z10': 'KO',
 'z11': 'ANALYZE',
 'z12': 'ANALYZE',
 'z13': 'KO',
 'z14': 'KO',
 'z15': 'KO',
 'z16': 'ANALYZE',
 'z17': 'KO',
 'z18': 'KO',
 'z19': 'ANALYZE',
 'z2': 'KO',
 'z20': 'KO',
 'z21': 'KO',
 'z22': 'ANALYZE',
 'z23': 'ANALYZE',
 'z24': 'ANALYZE',
 'z25': 'KO',
```


Figure 3.3: SetsDict

3.1.3 Indices

Quest'altra hash table viene utilizzata per sapere in quale posizione di M si trova una lettera come mostrato nella figure 3.4 e 3.5

```
def initialize_indices(M):
  indices = {}
  index = 0
  for letter in M:
    indices[letter] = index
    index += 1
  return indices
```

Figure 3.4: Funzione Per Creare Indices

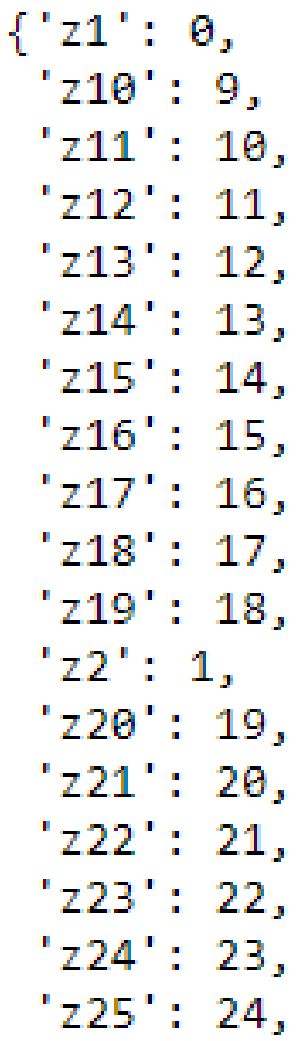

Figure 3.5: Indices

3.1.4 Initialize Queue

Questa è l'ultima funzione che prepara all'esecuzione di MBase ed è riportata in figura 3.6

```
def initialize_queue(M, sets_dict, indices):
  queue = []
  minimum_sets = [letter for letter in M if sets_dict[letter] == MHS]
  old_queue = [letter for letter in M if sets_dict[letter] == ANALYZE and letter != M[-1]]
  # print(old_queue)
  for letter1 in old_queue:
    for letter2 in M:
      arr = [letter1]
      if sets_dict[letter2] == ANALYZE and indices[letter1] < indices[letter2]:
        arr.append(letter2)
        queue.append(arr)
  return queue, minimum_sets
```

Figure 3.6: Funzione InitializeQueue

Questa procedura prende innanzitutto nota di quali insiemi sono già MHS, in quel caso non ha infatti senso farli riprocessare dall'algoritmo dato che darebbero sicuramente luogo a KO.
Viene poi inizializzata una lista "oldQueue" con tutte le lettere che sono ANALYZE e non sono l'ultimo elemento di M
Ora per ogni lettera di "oldQueue" e per ogni lettera di M (variabile letter2) che è ANALYZE e viene lessicograficamente dopo la lettera corrente di "oldQueue" (nel codice "letter1") viene messa in append alla lista "arr" che viene poi aggiunta a "queue"
I risultati ritornati sono rispettivamente la queue con ora tutti insiemi da 2 elementi e la lista dei minimal sets trovati fino ad ora

3.2 Esecuzione di MBase

In figura 3.7 viene riportato il codice di MBase e nei prossimi sottoparagrafi sarà spiegato come funziona

```python
import time
from tqdm.auto import tqdm
processed_times = []
for matrix_file in tqdm(matrices_files_2[295:]):
  start = time.time()
  isInterrupted = False
  matrix = read_matrix(benchmark_2 + "/" + matrix_file)
  M = np.array(["z" + str(i + 1) for i in range(matrix.shape[1])])
  sets_dict = initialize_sets_dict(matrix, M)
  new_matrix = initialize_new_matrix(matrix, M)
  indices = initialize_indices(M)
  minimum_sets = []
  queue, minimum_sets = initialize_queue(M, sets_dict, indices)
  while len(queue) > 0:
    lambda_set = queue.pop(0)
    elapsed_time = time.time() - start

    if elapsed_time > 600:
      print("Maximum time reached, processing the next entity")
      isInterrupted = True
      processed_times.append(elapsed_time)
      write_results(f"./benchmarks_2_mhs/{matrix_file}", minimum_sets, matrix, elapsed_time, isSolved=False)
      break
    # print("Lambda Set: ", lambda_set)

    res = check(lambda_set)

    # print(res)

    if res == "OK" and lambda_set[-1][-1] != M[-1]:
      new_sets = [lambda_set + [letter] for letter in M if indices[letter] > indices[lambda_set[-1]] and sets_dict[letter] == ANA
      # print("New Sets: ", new_sets)
      for new_set in new_sets:
        queue.append(new_set)
    if res == MHS:
      # print("MHS: ", lambda_set)
      minimum_sets.append(lambda_set)
  # print(minimum_sets)
  if not isInterrupted:
    end = time.time() - start
    processed_times.append(end)
    write_results(f"./benchmarks_2_mhs/{matrix_file}", minimum_sets, matrix, elapsed_time, isSolved=True)
```

Figure 3.7: Algoritmo MBase

3.2.1 Inizializzazione

Dalla variabile start, che prende il tempo di inizio dell'elaborazione di un'istanza, a "queue, minimumSets" vengono eseguiti tutti i passaggi descriti nelle sezioni precedenti
Da notare la presenza della variabile isInterrupted che serve al momento della scrittura dell'output per sapere se l'elaborazione è terminata prima o dopo il limite di 10 minuti che ho deciso di imporre

3.2.2 Ciclo While

Il ciclo while e le relative istruzioni sono eseguite fintantoché esistono ancora insiemi candatidati ad essere MHS
Si inizia estraendo un candidato dalla coda, con un pop(), denominato lambdaSet e si calcola poi quanto tempo è passato con la variabile elapsedTime.
Se sono passati 10 minuti viene interrotta l'esecuzione di MBase su questa istanza e si scrivono i risultati fino ad ora ottenuti su un file di testo la cui stuttura, piuttosto criptica, sarà illustrata in un capitolo dedicato con l'ausilio della libreria pandas di Python
Se i 10 minuti non sono ancora trascorsi viene lanciata la procedura check() sul lambdaSet e il risultato viene memorizzato dentro la varibile "res"
Sulla base del suo valore vegono svolte le seguenti operazioni:

1. res="OK": e l'ultimo elemento del lambdaSet non è l'ultimo elemento di M viene creata la lista newSets che è semplicemente una lista di insiemi in cui la radice è sempre il lambdaSet e la desinenza è l'elemento di M successivo all'ultimo elemento della radice, a meno che tale desinenza non sia già un MHS o un KO

2. res="KO": non viene fatto nulla

3. res="MHS": il lambdaSet viene aggiunto alla lista degli MHS trovati

Una volta fuori dal ciclo while viene controllato se si è usciti per aver raggiunto il limite di tempo (isInterrupted=True) oppure no
Se non si è usciti per questo motivo ma perché non c'erano più lambdaSet i risultati vengono scritti su file

```python
def check(lambda_set):
  lambda_indices = [indices[letter] for letter in lambda_set]
  # print(lambda_indices)
  filtered_matrix = new_matrix[:, lambda_indices]
  # print(filtered_matrix)
  vec = ["".join(col) for col in filtered_matrix]
  # print(vec)
  new_vec = [elem for elem in vec if elem.count('z') <= 1]
  # print(new_vec)
  new_vec = np.array(new_vec)
  proj_vec = np.unique(new_vec)
  # print(proj_vec)
  if proj_vec.tolist() == lambda_set:
    return "MHS"
  elif proj_vec.tolist() == [''] + lambda_set:
    return "OK"
  return "KO"
```

Figure 3.8: Modulo Check

3.2.3 Modulo Check

Il modulo check viene riportato in figura 3.8 Innanzitutto viene costruito un array di indici date le lettere del lambdaSet
Dopodich° sfruttando l'array slicing vengono ricavate le colonne della matrice di input corrispondenti agli indici indicati (newMatrix è semplicemente la matrice di input con le lettere j-esime di M se nella j-esima posizione di una riga c'è un 1)
A questo punto viene inizializzato un nuovo array, vec, che consente di trasformare le colonne in un'unica stringa
Di queste stringhe di vec, newVec mantiene solo gli elementi con una "z", una lettera, in quanto, gli altri indicano che quel lambdaSet produce una "X", come illustrato sulle slide
In figura 3.9 sono mostrate queste due liste Ora newVec viene convertito in un numpy array in modo da effettuarne la proiezione con il metodo "unique()"
Se tale proiezione è identica al lambdaSet allora tale elemento è un MHS
Se invece tale proiezione è il lambdaSet + degli spazi vuoti, che sarebbero degli zeri, allora l'insieme è "OK"
Se non vale nessuna di queste casistiche allora l'insieme è "KO"

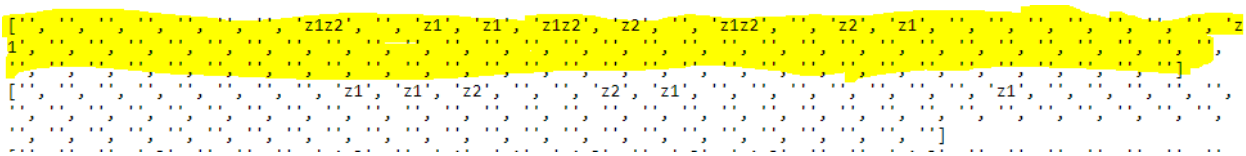

Figure 3.9: In giallo Vec mentre newVec non è evidenziato

Chapter 4

Analisi Delle Performance

4.1 Scrivere i risultati su File

Nel capitolo precedente è stata volontariamente tralasciata la spiegazione della funzione writeResults che verrà illustrata in questa sezione e che costituirà il fondamento per l'analisi delle performance di MBase
La figura 4.1 ne illustra il codice

```
def write_results(filename, minimum_sets, matrix, elapsed_time, isSolved):
  with open(filename, "w") as f:
    f.write(str(len(minimum_sets)) + "\n")
    f.write(str(elapsed_time) + "\n")
    f.write(str(matrix.shape[1]) + "\n")
    f.write(str(matrix.shape[0]) + "\n")
    f.write(str(isSolved) + "\n")
    f.write(str(matrix.shape[0] * matrix.shape[1]) + "\n")
    for minimum_set in minimum_sets:
      for single_set in minimum_set:
        f.write(f"{single_set}, ")
      f.write("\n")
    f.close()
```

Figure 4.1: Funzione Write Results

Come si può vedere vengono scritti su file, in ordine, i seguenti parametri

1. Numero di Minimal Hitting Set trovati
2. Tempo di elaborazione

3. Numero di Colonne della Matrice
4. Numero di Righe della Matrice
5. Se l'istanza è stata risolta oppure se l'esecuzione è stata interrotta
6. Numero totale di elementi della Matrice
7. Tutti i minimal hitting set trovati

La struttura del file è mostrata in figura 4.2 in cui, per questioni di praticità, non è stato riportato il significato del dato presente

```
154
1.017171859741211
1193
5
True
5965
z23, z382, z577,
z23, z430, z577,
z371, z382, z577,
z371, z430, z577,
z382, z427, z577,
z427, z430, z577,
z23, z35, z368, z577,
z23, z36, z368, z577,
z23, z230, z368, z577,
z23, z326, z382, z576,
z23, z326, z430, z576,
z23, z328, z368, z577,
z23, z376, z382, z576,
z23, z376, z430, z576,
z23, z380, z382, z576,
z23, z380, z430, z576,
z23, z382, z469, z576,
z23, z382, z509, z576,
z23, z430, z469, z576,
z23, z430, z509, z576,
z35, z368, z371, z577,
z35, z368, z427, z577,
z36, z368, z371, z577,
z36, z368, z427, z577,
z117, z371, z382, z576,
z117, z371, z430, z576,
z117, z382, z427, z576,
z117, z427, z430, z576,
```

Figure 4.2: File Di Output

4.2 Preparazione Data Frame

Grazie alla libreria di Python, Pandas, è possibile organizzare tutti questi dati in un struttura tabulare nominando le colonne
In figura 4.3 è mostrato come questi file, contenenti ciascuno l'output di WriteResults, vengono letti

```
results_files = os.listdir("./benchmarks_1_mhs")
useful_results = []
for result_file in tqdm(results_files):
  path = f"./benchmarks_1_mhs/{result_file}"
  with open(path) as f:
    lines = f.readlines()
    file_type = result_file.split(".")[0]
    seconds = float(lines[1][:-1])
    cols = int(lines[2][:-1])
    rows = int(lines[3][:-1])
    solved = lines[4][:-1]
    if solved == 'True':
        solved = True
    else:
        solved = False
    # print(seconds, solved, cols * rows)
    useful_results.append([file_type, seconds, cols, rows, solved, rows * cols])
    f.close()
useful_results
```

Figure 4.3: Lettura File Con Risultati

L'array usefulResults viene poi utilizzato per creare il dataframe come mostrato in figura 4.4

```
df = pd.DataFrame(useful_results, columns=["file_type", "seconds", "cols", "rows", "is_solved", "#elements"])
df
```

	file_type	seconds	cols	rows	is_solved	#elements
0	74181	0.070903	66	3	True	198
1	74181	0.022543	66	2	True	132
2	74181	0.010998	66	2	True	132
3	74181	0.021001	66	2	True	132
4	74181	0.007001	66	2	True	132
...	...	...	...	...	...	...
1366	c880	600.002872	384	47	False	18048
1367	c880	600.000333	384	43	False	16512
1368	c880	600.004869	384	43	False	16512
1369	c880	600.000890	384	36	False	13824
1370	c880	600.001499	384	36	False	13824

1371 rows × 6 columns

Figure 4.4: DataFrame Benchmarks 1

4.3 Data Analysis

In quest'ultima sezione andremo a svolgere delle semplici operazioni di Data Analysis sui risultati ottenuti. In particolare sarà analizzato:

1. Numero di Istanze Risolte vs Non Risolte
2. Relazione tempo-numero elementi matrice
3. Relazione tempo-numero righe matrice
4. Relazione tempo-colonne matrice
5. Ricerca della più grande istanza risolta

4.3.1 Risolte Vs Non Risolte

In figura 4.5 viene mostrato l'istogramma delle istanze risolte del benchmark numero 1
Come si può notare prevalgono nettamente le istanze non risolte (più di 1000) rispetto alle risolte (tra le 300 e le 400)
Questo significa tre cose:

1. L'algoritmo MBase non è sufficientemente efficiente da risolvere la maggior parte delle istanze
2. Il linguaggio di programmazione adottato non permette di risolvere istanze grandi. A questo punto è leggittimo chiedersi se usando Java o C i due grafici sarebbero stati un po' più bilanciati
3. Sono presenti nel codice errori di implementazione

Il terzo caso mi è sembrato opportuno farlo presente siccome, non essendo il problema decidibile, non è possibile sapere al 100% se la propria implementazione è corretta o meno

```
df = df.replace([False, True], [0, 1])
df['is_solved'].hist()
```

```
<AxesSubplot:>
```

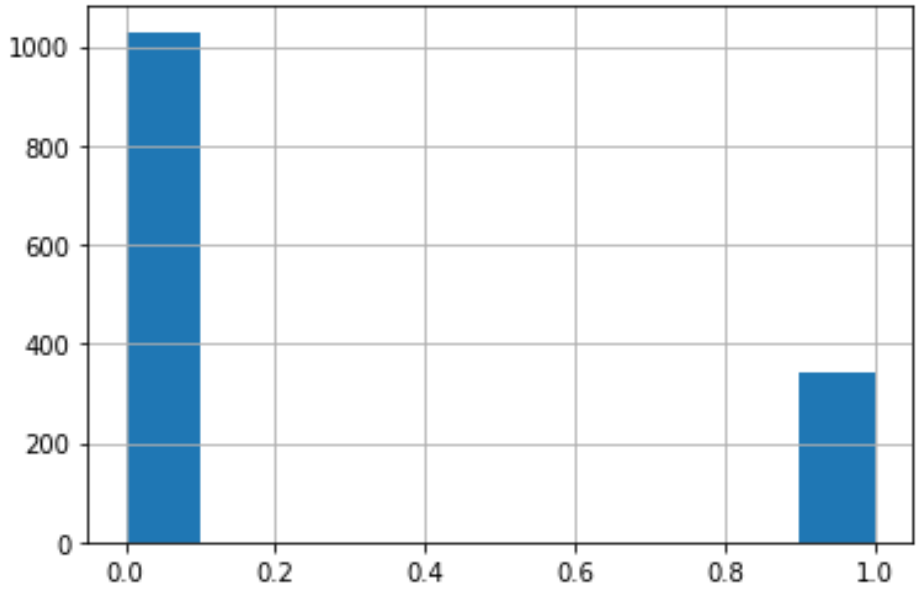

Figure 4.5: Risolte Vs Non Risolte

4.3.2 Tempo Vs Numero Elementi

In figura 4.6 viene riportato il grafico a punti (scatter plot) che mostra la relazione tra il numero di elementi della matrice di ingresso (asse x) e il tempo che si è impiegato a risolvere l'istanza (asse y)

```
seconds_per_elements = df[['seconds', '#elements']]
# print(seconds_per_elements)
X = seconds_per_elements['#elements']
y = seconds_per_elements['seconds']
plt.scatter(X, y)
```

```
<matplotlib.collections.PathCollection at 0x1fa6b03cac0>
```

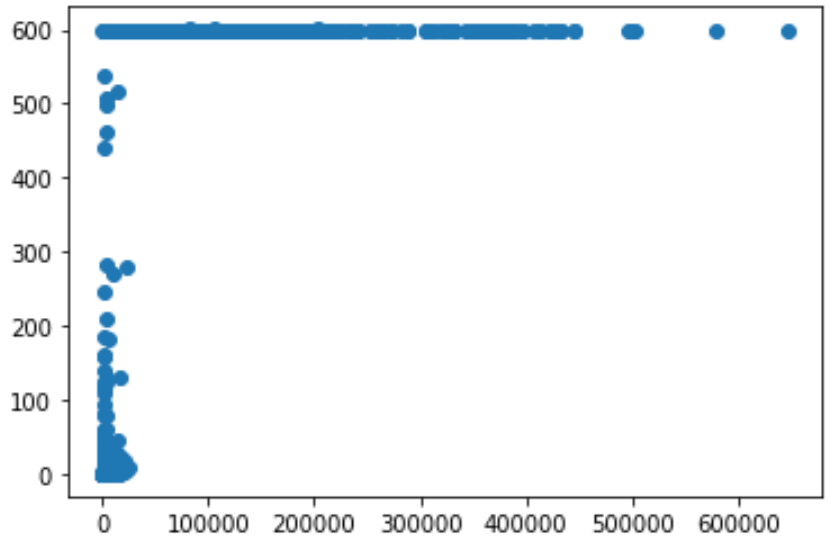

Figure 4.6: Scatter Plot Numero Elementi Vs Tempo Esecuzione

Come si può notare la relazione sembra essere di tipo esponenziale: all'aumentare del numero di istanze il tempo di esecuzione aumenta esponenzialmente. Chiaramente identificare il tipo di relazione che intercorre tra le due grandezze esula dallo scopo di questa relazione, tuttavia non è difficile concludere che più la matrice possiede elementi più è alta la probabilità che l'algoritmo non risolverà l'istanza?
Notare che si parla di probabilità non di certezze: come vedremo infatti

l'istanza più grande risolta presenta ben 25000 elementi andando in contraddizione con quanto affermato

4.3.3 Righe Matrice Vs Tempo Esecuzione

In figura 4.7 viene riportato lo scatter plot della relazione tra righe della matrice e tempo di esecuzione dell'algoritmo mettendo sull'asse x il numero di righe e sull'asse y il tempo di esecuzione

```
seconds_per_rows = df[['seconds', 'rows']]
X = seconds_per_rows['rows']
y = seconds_per_rows['seconds']
plt.scatter(X, y)
```

```
<matplotlib.collections.PathCollection at 0x1fa6b0aaac0>
```

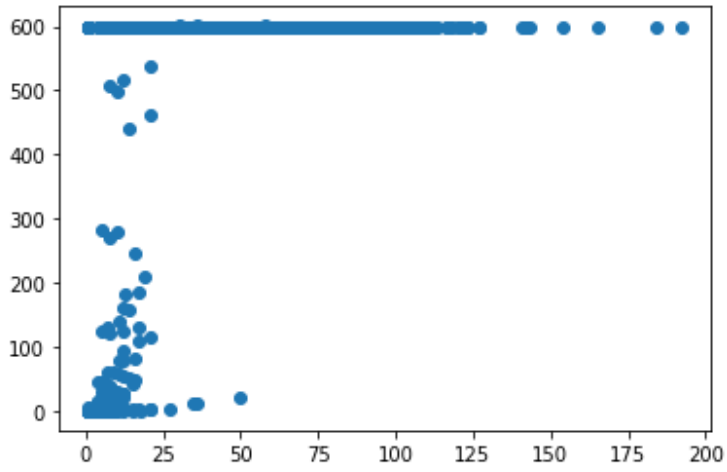

Figure 4.7: Scatter Plot Numero Righe Vs Tempo Esecuzione

Come si può notare anche qui, se non per alcune eccezioni, la relazione sembra essere di tipo esponenziale: più la matrice ha righe più è probabile che l'aglrotimo non sia in grado di risolverla nei 10 minuti di tempo a disposizione

4.3.4 Numero Colonne Vs Tempo Esecuzione

L'ultimo caso di scatter plot che andremo ad analizzare è quello della relazione che sussiste tra colonne della matrice e tempo di esecuzione che viene riportata

in figura 4.8

```
seconds_per_cols = df[['seconds', 'cols']]
X = seconds_per_cols['cols']
y = seconds_per_cols['seconds']
plt.scatter(X, y)
```

```
<matplotlib.collections.PathCollection at 0x1fa6b0ea130>
```

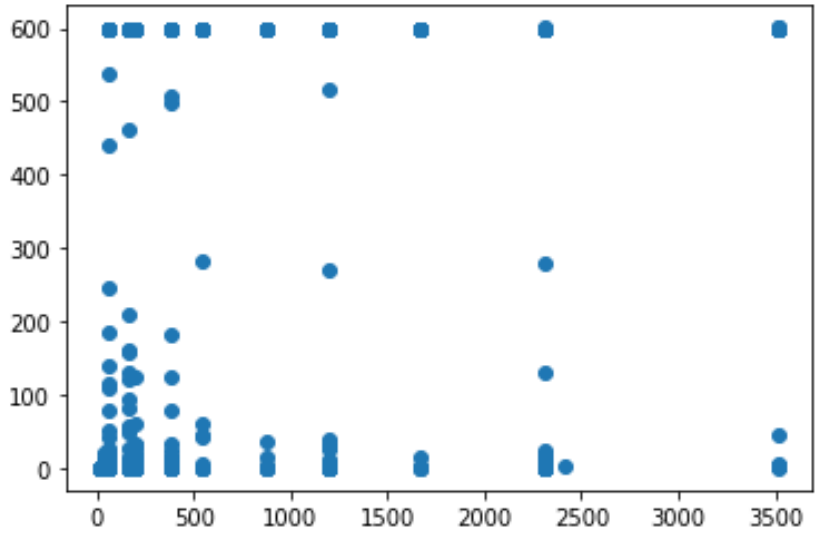

Figure 4.8: Scatter Plot Numero Colonne Vs Tempo Esecuzione

Diversamente dai 2 casi precedenti quest'ultimo non da luogo ad alcuna relazione possibile: ciò che ci viene sugerito dai punti è infatti che a volte un'istanza con 3500 colonne impiega meno di 100 secondi per essere eseguita mentre altre volte più di 10 minuti
Questo indica l'assenza di relazione tra colonne della matrice e tempo di esecuzione e per cui si può affermare che quest'ultima può avere anche un gran numero di colonne ma se ha poche righe l'algoritmo MBase impiegherà molto poco a risolverla, mentre se ha tante righe impiegherà sicuramente tanto, non importa, ovviamente fino ad un certo punto, quante colonne possiede

4.3.5 Ricerca Della Più Grande Istanza Risolta

In figura 4.9 viene riportata la più grande istanza risolta dall'algoritmo

```
#Pick the bigger solved instance
solved_df = df[df['is_solved'] == 1]
solved_df['#elements'].max()
solved_df.sort_values(by="#elements")
```

	file_type	seconds	cols	rows	is_solved	#elements
63	74182.009.matrix	0.007003	20	1	1	20
62	74182.008.matrix	0.007003	20	1	1	20
61	74182.007.matrix	0.007003	20	1	1	20
60	74182.006.matrix	0.007003	20	1	1	20
59	74182.005.matrix	0.007003	20	1	1	20
...	...	...	...	...	...	...
780	c5315.040.matrix	17.902951	2308	9	1	20772
779	c5315.039.matrix	2.147076	2308	9	1	20772
787	c5315.049.matrix	4.417778	2308	10	1	23080
781	c5315.041.matrix	280.701851	2308	10	1	23080
788	c5315.050.matrix	10.001959	2308	11	1	25388

342 rows × 6 columns

Figure 4.9: Istanza Più Grande Risolta

In questo caso vengono innanzitutto prelevate le sole istanze risolte e di queste viene ricercato l'elemento più grande in termini di numero di elementi Quest'ultimo risulta avere:

1. 25388 elementi
2. 11 righe

3. 2308 colonne

4. Aver impiegato 10.001959 secondi per esssere risolta

Ciò ovviamente non significa che questo algoritmo sarà in grado di risolvere tutte le istanze che hanno meno di 11 righe e 2308 colonne
Andiamo quindi a ispezionare come è fatto il file che risolve questa istanza mostrato in figura 4.10

```
26
10.00195860862732
2308
11
True
25388
z11, z18, z194, z209, z225, z575,
z11, z18, z194, z209, z225, z874,
z11, z18, z194, z209, z225, z992,
z11, z18, z194, z209, z575, z602,
z11, z18, z194, z209, z602, z874,
z11, z18, z194, z209, z602, z992,
z11, z18, z194, z209, z225, z342, z995,
z11, z18, z194, z209, z225, z344, z650,
z11, z18, z194, z209, z225, z344, z809,
z11, z18, z194, z209, z225, z344, z995,
z11, z18, z194, z209, z342, z602, z995,
z11, z18, z194, z209, z344, z602, z650,
z11, z18, z194, z209, z344, z602, z809,
z11, z18, z194, z209, z344, z602, z995,
z11, z18, z194, z209, z225, z242, z342, z650,
z11, z18, z194, z209, z225, z242, z342, z809,
z11, z18, z194, z209, z225, z342, z343, z650,
z11, z18, z194, z209, z225, z342, z343, z809,
z11, z18, z194, z209, z225, z342, z574, z650,
z11, z18, z194, z209, z225, z342, z574, z809,
z11, z18, z194, z209, z242, z342, z602, z650,
z11, z18, z194, z209, z242, z342, z602, z809,
z11, z18, z194, z209, z342, z343, z602, z650,
z11, z18, z194, z209, z342, z343, z602, z809,
z11, z18, z194, z209, z342, z574, z602, z650,
z11, z18, z194, z209, z342, z574, z602, z809,
```

Figure 4.10: File Di Risoluzione Istanza Più Grande

Notiamo che il numero di MHS trovati è di solo 26 e quindi possiamo ipotizzare che il fatto che riesca a risolvere o no l'istanza sia legato anche al numero di 0 e 1 presenti nella matrice oltre che al numero di righe?

4.3.6 Altre Grandezze Vs Tempo di Esecuzione

In quest'ultima sottosezione andremo ad analizzare altre 3 possibili relazioni con il tempo di esecuzione:

1. Numero di zeri della matrice
2. Numero di uno della matrice
3. Rango della matrice

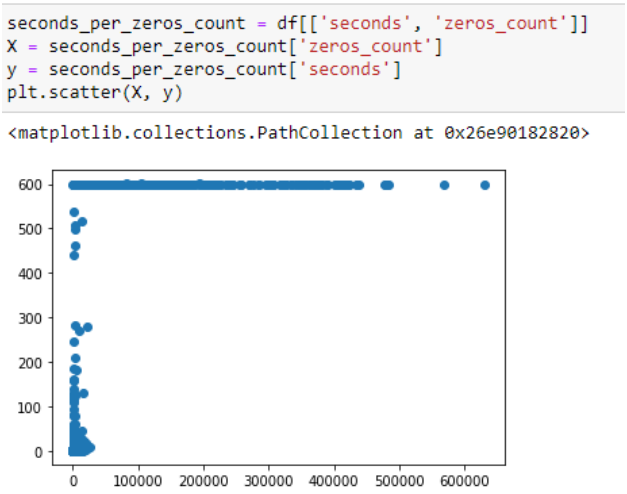

Figure 4.11: Numero Zeri Della Matrice Vs Tempo di Esecuzione

Come si può notare da tutti e 3 i grafici la relazione è facilmente approssimabile ad un esponenziale suggerendo le seguenti osservazioni:

1. Non importa se la matrice ha tanti o pochi zeri perché tale mancanza o eccedenza sarà complementata dagli 1 rendendo il tempo di esecuzione ugualmente esponziale
2. Il tempo di esecuzione aumenta esponenzialmente anche con il rango della matrice ma ciò non sorprende siccome la maggior parte delle matrici di input possedevano un numero di colonne nettamente maggiore rispetto al numero di righe e quindi non è difficile che avessero rango pieno, ossia pari al numero di righe

Con queste ultime 3 relazioni si conclude l'analisi delle performance dell'algoritmo MBase

```
seconds_per_ones_count = df[['seconds', 'ones_count']]
X = seconds_per_ones_count['ones_count']
y = seconds_per_ones_count['seconds']
plt.scatter(X, y)
```

```
<matplotlib.collections.PathCollection at 0x26e902fd910>
```

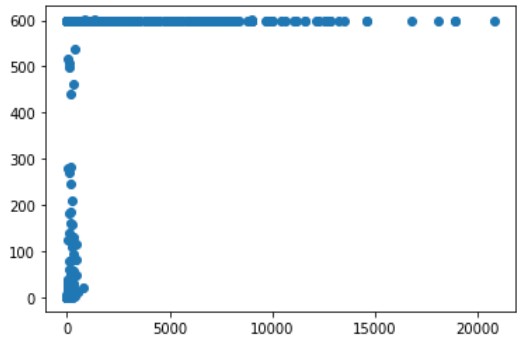

Figure 4.12: Numero Uni Della Matrice Vs Tempo di Esecuzione

```
seconds_per_rank = df[['seconds', 'matrix_rank']]
X = seconds_per_rank['matrix_rank']
y = seconds_per_rank['seconds']
plt.scatter(X, y)
```

```
<matplotlib.collections.PathCollection at 0x26e901f5a90>
```

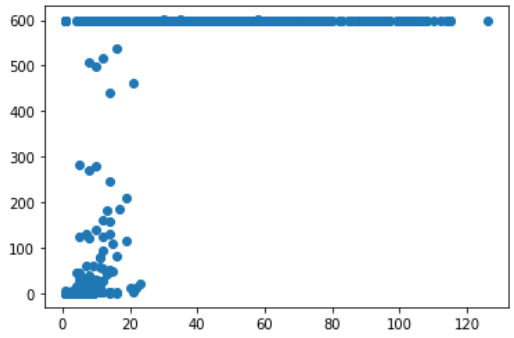

Figure 4.13: Rango Matrice vs Tempo

4.3.7 Altre Possibili Relazioni

Sarebbe comunque interessante continuare ad analizzare le performance dell'algoritmo cercando altre relazioni come ad esempio

1. Numero di KO set trovati alla prima iterazione (preprocessing) Vs

Tempo

2. Numero di MHS trovati alla prima iterazione (preprocessing) Vs Tempo
3. Cardinalità del più grande MHS trovato nella matrice Vs Tempo
4. Rapporto Righe/Colonne
5. Rapporto Colonne/Righe

E tante altre, il limite è solo la nostra creatività
Queste ultime analisi non sono state da me svolte per via di una non sufficiente organizzazione del lavoro e della struttura dei file di output poco orientata all'analisi delle performance ma molto di più a "far funzionare" l'algoritmo e analizzarlo superficialmente
Inoltre l'analisi di alcune relazioni richiederebbe di riesguire l'algoritmo nuovamente su tutte le istanze e questo va oltre sicuramente lo scopo di questo lavoro

Chapter 5

Conclusioni

Nonostante non sia stato possibile analizzare le performance dell'algoritmo MBase in tante sfaccettature la conclusione che posso trarre dal precedente capitolo è che tale procedura, presa nella sua versione basilare, richiede un tempo di esecuzione esponenziale che sembra non dipendere dalla scelta delle strutture dati da me effettuata quanto piuttosto dall'input che, banalmente, più è grande più chiede tempo di esecuzione
Ovviamente numerose sono le ottimizzazioni che possono essere apportate al codice con il solo fine di poter risolvere istanze più grandi, non di certo abbassare la complessità esponenziale
Nel caso si volesse optare per questo aspetto ciò che si può concludere è agire sull'algoritmo MBase cambiandone il codice e ponendo maggiore attenzione sulla fase di preprocessing e sull'evitare di utilizzare tutti gli elementi di un MHS all'interno di un lambda set