UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea in Informatica

Tesi di Laurea in Informatica

Un'analisi sperimentale di algoritmi per Bin Packing

Relatore
Ch.mo Prof. Ugo Vaccaro

Candidato
Francesco Migliaro
Matricola 0512105109

Anno accademico 2019–2020

Indice

Abstract

Nel problema del Bin Packing, degli oggetti di volume diverso devono essere allocati in un numero finito di contenitori (bin), ciascuno di una determinata capacità fissa, in modo da ridurre al minimo il numero di bin utilizzati. Il Bin Packing ha numerose applicazioni. Per esempio l'inserimento di file con dimensioni specifiche in blocchi di memoria di dimensioni fisse o la registrazione di tutta la musica di un compositore, dove la lunghezza dei pezzi da registrare (in bytes) è la dimensione degli oggetti e la capacità del generico bin è la quantità di bytes che può essere memorizzata su un CD audio. Il problema del Bin Packing è un noto problema NP-hard, per cui è irrealistico pensare che si possano trovare algoritmi di complessità polinomiale per la sua soluzione. In questa tesi, analizziamo e valutiamo sperimentalmente due algoritmi per un'importante variante del Bin Packing, in cui è possibile "riempire" ogni bin anche al disopra della sua capacità. L'obiettivo è di minimizzare la somma dei valori assoluti delle differenze tra le capacità dei bin e il volume totale del loro contenuto.

Elenco delle figure

Elenco delle tabelle

Introduzione

Argomento trattato

All'interno di questa tesi si è trattato di algoritmi che risolvono il problema del Bin Packing con bins estendibili. Il problema è quello di disporre un numero grande di oggetti all'interno di un numero molto più piccolo di contenitori che possono essere estesi se la loro capacità viene superata, con lo scopo di minimizzare lo spazio vuoto e in eccesso totale dei contenitori.
Per questo problema non esiste un algoritmo polinomiale che lo risolve all'ottimo, infatti fa parte della classe di problemi NP-hard, di conseguenza bisogna approssimare la soluzione nel miglior modo possibile tramite algoritmi che fanno uso di euristiche.

Obiettivo

L'obiettivo della tesi è quello di analizzare sperimentalmente due algoritmi di approssimazione per il problema del Bin Packing con bins estendibili, per trarre in conclusione qual è il migliore tra i due in base alle prestazioni e risultati per il problema in esame.
Tali algoritmi si basano su due euristiche diverse, il primo opera inserendo ogni volta l'oggetto di volume maggiore all'interno del contenitore più vuoto, mentre il secondo unisce gli oggetti di volume minore fin quando il loro numero non corrisponde a quello dei contenitori e li inserisce in modo arbitrario in essi.

Organizzazione della tesi

La tesi è suddivisa in 3 capitoli, all'interno del primo capitolo si formalizza il problema del Bin Packing classico, si parla della sua complessità e si introducono alcune sue variazioni; all'interno del secondo si formalizza il problema del Bin Packing con bins estendibili e vengono descritti i due algoritmi che saranno analizzati; infine, all' interno del terzo capitolo si discute dell'approccio utilizzato alla sperimentazione, gli strumenti utilizzati e vengono forniti i dati da cui sono state fatte le considerazioni finali, anch'esse presenti all'interno di questo capitolo.

Capitolo 1

Bin Packing Problem

1.1 Il problema

Spesso ci si può ritrovare a dover risolvere problemi in cui si devono disporre degli oggetti in contenitori al fine di utilizzarne il minor numero possibile, ad esempio se si deve prendere un aereo bisogna organizzare tutte le proprie cose in delle valigie da portare con sé, valigie che ovviamente avranno un costo di trasporto, di conseguenza si è interessati ad utilizzare il minimo numero di valigie al fine di spendere il meno possibile per i bagagli. Questa è proprio una classica applicazione del problema del Bin Packing, un problema che è più frequente di quanto si immagini.

1.1.1 Formulazione

Il problema del Bin Packing è così formulato:

> Dati degli oggetti di volume differente, questi devono essere allocati in un numero finito di contenitori (bin), ciascuno di una determinata capacità fissa, in modo da ridurre al minimo la quantità di contenitori utilizzata.

Un altro modo di formulare il problema è attraverso la programmazione lineare intera:

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n} y_i \\
\text{s.t.} \quad & \sum_{j=1}^{n} w_j x_{ij} \leq c y_i && \forall i \in \{1, ..., n\} \\
& \sum_{i=1}^{n} x_{ij} = 1 && \forall j \in \{1, ..., n\} \\
& y_i \in \{0, 1\} && \forall i \in \{1, ..., n\} \\
& x_{ij} \in \{0, 1\} && \forall i, j \in \{1, ..., n\}
\end{aligned}
$$

dove

w_j = peso dell'oggetto j, $w_j \in \mathbb{Z}^+$, $w_j \leq c \; \forall j \in \{1, ..., n\}$

c = capacità di ogni bin, $c \in \mathbb{N}$

$$y_i = \begin{cases} 1 & \text{se il bin } i \text{ è utilizzato} \\ 0 & \text{altrimenti} \end{cases}$$

$$x_{ij} = \begin{cases} 1 & \text{se l'oggetto } j \text{ è assegnato al bin } i \\ 0 & \text{altrimenti} \end{cases}$$

In modo da rendere più chiara la comprensione viene fornito un esempio:

> Sia S un insieme di cardinalità n contenente i vari oggetti, indicati da un numero intero positivo, e $s(i)$ il volume dei vari oggetti $\forall i \in S$ e sia c la capacità di un generico bin.
> Consideriamo la seguente istanza:
>
> $$S = \{1, ..., 10\}, \text{ quindi } |S| = n = 10, \; c = 15$$
>
> $$\begin{array}{cc} s(1) = 15 & s(6) = 7 \\ s(2) = 3 & s(7) = 13 \\ s(3) = 9 & s(8) = 1 \\ s(4) = 5 & s(9) = 2 \\ s(5) = 3 & s(10) = 11 \end{array}$$
>
> Una soluzione ottima per l'istanza è data dai bins:
>
> $$B_1 = \{1\}, \quad B_2 = \{7, 9\}, \quad B_3 = \{4, 5, 6\}, \quad B_4 = \{2, 8, 10\}, \quad B_5 = \{3\}$$
>
> E quindi si può concludere che il numero minimo di bins da utilizzare per quest'istanza è 5.

1.1.2 Difficoltà

Questo problema è chiaramente di natura combinatoria, e in particolare appartiene all'insieme dei problemi NP-hard, mentre la versione di decisione del problema fa parte dell'insieme dei problemi NP-complete ed è così definita [1]:

Istanza:	Insieme finito U di oggetti, una taglia $s(u) \in \mathbb{Z}^+ \; \forall u \in U$, una capacità intera per i bins B e un intero positivo K.
Domanda:	Esiste una partizione di U in insiemi disgiunti $U_1, U_2, ..., U_k$ tale che la somma delle taglie degli oggetti in ogni U_i è $\leq B$.

Nello specifico è un problema strongly NP-complete e questo può essere provato tramite una riduzione polinomiale da un altro problema strongly NP-complete, ovvero il 3-partition.
Nonostante ciò comunque è possibile risolvere il problema del Bin Packing in tempo pseudo-polinomiale per ogni numero di bins fissato ≥ 2 [2], e in

tempo polinomiale per ogni fissata capacità B enumerando tutte le possibili $(k+1)^{p(1)+p(2)+...+p(B)}$ assegnazioni del numero di bins al tipo di partizioni numeriche con cui è possibile descrivere il loro riempimento e cercando poi tramite ricerca esaustiva l'esistenza di una che è coerente con i dati del problema e ne soddisfa le condizioni.
Come conseguenza della classe di complessità di appartenenza del problema è chiaro quindi che bisogna comunque risolverlo in un tempo ragionevole, ossia polinomiale, anche nel caso generale, quindi si ricorre ad algoritmi di approssimazione adoperando diverse euristiche. Esistono anche algoritmi esatti non banali, come l'MTP [3] e il Bin Completion [4] che sono i due algoritmi esatti più efficienti per il problema trovati fin'ora.

1.2 Variazioni e applicazioni

Ci sono diverse variazioni di questo problema, ognuna con varie applicazioni, tra le più famose troviamo quella del Bin Packing a più dimensioni, il packing by weight e il VM packing. Ovviamente ne esistono molte altre ma di seguito verranno descritte quelle precedentemente elencate per dare un'idea della ricorrenza di questo problema, nel prossimo capitolo si parlerà della variante con bins estendibili nel dettaglio.

1.2.1 N-dimensional packing

Questa variante del problema classico occorre ogni volta che bisogna immagazzinare oggetti in contenitori non in base al loro volume ma in base alle loro misure in relazione alle dimensioni considerate nel problema. Ad esempio, se si vogliono riempire dei container con oggetti da spedire e ogni container ha un costo per essere spedito, al fine di minimizzare il costo di spedizione si vuole fare in modo che vengano usati meno container possibili. Questa è una classica applicazione di 3-dimensional packing, dove appunto si vuole minimizzare il numero di container utilizzati e assegnare gli oggetti ai container in base alle loro dimensioni, quali larghezza, lunghezza ed altezza.

1.2.2 Packing by weight

In questa variante anziché assegnare gli oggetti in base al loro volume si assegnano in base al loro peso. Si potrebbe fare un esempio analogo a quello del N-dimensional packing, ovvero se bisogna spedire della merce, e questa merce deve essere trasportata da dei camion, è risaputo che ogni camion può trasportare merce il cui peso totale è al massimo uguale ad una certa soglia, quindi l'obiettivo sarà quello di minimizzare i camion da utilizzare riempendoli con merce il cui peso totale non superi quella di tolleranza del generico camion.

1.2.3 VM packing

Questo problema prende il nome direttamente dal suo caso di utilizzo maggiore, ovvero il packing delle pagine di memoria condivise da delle Virtual Machines su uno stesso server.
In pratica, in questa variante, gli oggetti facenti parte di determinati insiemi

possono condividere il loro spazio quando sono contenuti nello stesso bin, quindi ne occupando di meno che quando sono in bin diversi, poiché aumenterebbero lo spazio occupato per colpa della loro taglia individuale. L'esempio di applicazione è quello già menzionato, ovvero se ci sono più Virtual Machines su un server si vuole ridurre la loro occupazione di memoria, quindi si può pensare di condividere lo spazio riguardante le pagine di memoria che le VMs utilizzano e se più VMs richiedono la stessa pagina questa non avrà bisogno di essere replicata ma solo referenziata, allora l'obiettivo è quello di minimizzare i blocchi di memoria occupata considerando il fatto che determinate pagine possono condividere spazio tra loro.

1.2.4 Nota

Si può notare che se il numero di bins è ristretto a uno solo, e gli oggetti oltre ad avere un peso hanno anche un valore, il problema di massimizzare il valore degli oggetti che possono essere contenuti dal bin è nient'altro che il problema dello zaino 0-1.
In generale tutti i cosiddetti *Knapsack Problems* possono essere visti come casi speciali del Bin Packing.

Capitolo 2

Bin Packing Problem con bins estendibili

2.1 Il problema

Nell'ambito dell'informatica, precisamente quando si parla di sistemi distribuiti e concorrenza, nasce il problema del bilanciamento del carico, ovvero ridurre il carico di lavoro per le risorse distribuendolo tra le altre disponibili, questo non è altro che una variante del problema del Bin Packing di cui segue la formulazione.

2.1.1 Formulazione

In questa versione del Bin Packing Problem abbiamo un numero fissato di bins, un numero fissato di oggetti molto maggiore di quello dei bins e la capacità di ogni singolo bin può essere ecceduta. Inoltre la somma dei volumi dei vari oggetti è pari alla somma delle capacità dei vari bins. L'obiettivo è quello di minimizzare la quantità data dalla somma dei valori assoluti delle differenze tra le capacità del bin e la somma del volume dei vari oggetti che essi contengono. Formalmente:

> Dati m bins $\{B_1, ..., B_m\}$ con capacità c uguale per tutti e n oggetti $\{o_1, ..., o_n\}$ con $n >> m$ e di volume arbitrario (che indicheremo con $s(o_i)\ \forall i \in \{1, ..., n\}$), tali che $\sum_{i=1}^{n} s(o_i) = mc$, bisogna allocare gli n oggetti negli m bins in modo da minimizzare la quantità $\sum_{i=1}^{m} |c - x_i|$ dove $x_i = \sum_{o \in B_i} s(o)\ \forall i \in \{1, ..., m\}$.

Una possibile formulazione come un problema di programmazione lineare intera è la seguente:

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{m} |c - \sum_{j=1}^{n} x_{ij} w_j| & \\
\text{s.t.} \quad & \sum_{i=1}^{n} x_{ij} = 1 & \forall j \in \{1, ..., n\} \\
& x_{ij} \in \{0, 1\} & \forall i, j \in \{1, ..., n\}
\end{aligned}
$$

dove

w_j = volume dell'oggetto j, $w_j \in \mathbb{Z}^+ \; \forall j \in \{1, ..., n\}$

c = capacità di ogni bin, $c \in \mathbb{N}$

$$
x_{ij} = \begin{cases} 1 & \text{se l'oggetto } j \text{ è assegnato al bin } i \\ 0 & \text{altrimenti} \end{cases}
$$

ed è stata assunta soddisfatta la seguente condizione:

$$
\sum_{j=1}^{n} w_j = mc
$$

2.2 Due possibili algoritmi

Di seguito verranno illustrati due algoritmi di approssimazione che risolvono il problema del Bin Packing con bins estendibili dei quali verrà analizzato il comportamento da un punto di vista sperimentale nel prossimo capitolo.

2.2.1 Algoritmo con euristica MBO

Il primo algoritmo utilizza un'euristica nota per i problemi di scheduling multiprocessore, l'euristica Longest Processing Time, ovvero quella secondo cui si assegnano i job alle risorse in base al loro tempo di esecuzione, precisamente vengono assegnati in ordine non decrescente di tempo di esecuzione alla risorsa più "libera" al fine di minimizzare il *makespan*, ovvero il massimo tempo di utilizzo totale di una risorsa tra le varie. Il problema dello scheduling multiprocessore è molto simile, si potrebbe dire equivalente, a questa variante del Bin Packing.
In questo caso, poiché il problema non è di scheduling ma di Bin Packing, potremmo chiamare l'euristica Most Big Object e quindi procedere nel seguente modo per ottenere una soluzione:

1: **function** BINPACKMBO($B, O, s()$)
2: *Ordina gli oggetti in modo non decrescente in base al volume*
3: $Q = B$
4: **for** $i = 1$; $i \leq |O|$; $i = i + 1$ **do**
5: bin-più-vuoto $\leftarrow$ EXTRACTMAX($Q, s()$)
6: bin-più-vuoto $\leftarrow$ bin-più-vuoto $\cup \{O[i]\}$
7: $Q \leftarrow Q \cup \{$bin-più-vuoto$\}$
8: **return** $\sum_{b \in B} |b|$

L'algoritmo prende in input l'insieme di bins B, la lista di oggetti O e la funzione taglia $s()$. Dopo aver ordinato gli oggetti in modo non decrescente, utilizzando il loro volume come metro comparativo, procede con l'assegnazione degli oggetti ai bins in base a quale sia quello più vuoto al momento dell'assegnazione, semplicemente scorrendo la lista di oggetti da sinistra a destra, quindi dal più voluminoso al meno voluminoso, ed estraendo dall'insieme Q il bin di taglia (capacità) maggiore, ovvero quello più vuoto, e modificando quindi implicitamente la capacità di quest'ultimo diminuendola del peso dell'oggetto assegnatogli, per poi rimetterlo in Q. Infine ritorna il valore a cui si è interessati.

2.2.1.1 Analisi della complessità dell'algoritmo

L'istruzione al rigo `2` prende tempo $O(n \log n)$ dove n è la lunghezza della lista di oggetti, il for delle righe `4 - 7` viene eseguito n volte, l'istruzione del rigo `5`, se l'insieme Q è un semplice insieme la ricerca deve essere effettuata in modo esaustivo, quindi l'operazione di estrazione del massimo ha complessità $\Theta(m)$ dove m è la cardinalità di Q (quindi il numero di bins), le altre istruzioni all'interno del `for` prendono tempo costante, e l'espressione nel `return` alla fine prende tempo $\Theta(m)$. Quindi la complessità dell'algoritmo in questo caso è $O(nm)$ (se $m > \log n$) per via della ricerca esaustiva che viene fatta n volte all'interno del for, ma se Q fosse uno heap l'operazione di estrazione del massimo prenderebbe tempo $\Theta(\log m)$, quella di inserimento in Q prenderebbe anch'essa tempo $\Theta(\log m)$ ma se ne trae vantaggio perché la complessità del `for` diventerebbe $O(n \log m)$ che è minore di $O(nm)$ e quindi la complessità dell'algoritmo diventerebbe $O(n \log n)$, ovvero dominata dall'istruzione al rigo `2`. L'algoritmo sarà quindi implementato usando uno heap come struttura per Q.

2.2.1.2 Esempio di esecuzione

Anche se non verrà rispettata la condizione per cui $n >> m$ al fine di rendere più semplice l'esempio questo non rappresenta una perdita di generalità.

> Sia O una lista di lunghezza n contenente i vari oggetti, e $s()$ la funzione di taglia e sia B un insieme di cardinalità m contenente i vari bin ognuno di capacità c.
> Consideriamo la seguente istanza:
>
> $$O = \{o_1, ..., o_{10}\}, \text{ quindi } |O| = n = 10 \text{ e}$$
> $$B = \{B_1, B_2, B_3\}, \text{ quindi } |B| = m = 3,\ c = 18$$
>
> $$\begin{array}{ll} s(o_1) = 4 & s(o_6) = 7 \\ s(o_2) = 5 & s(o_7) = 9 \\ s(o_3) = 4 & s(o_8) = 3 \\ s(o_4) = 5 & s(o_9) = 3 \\ s(o_5) = 5 & s(o_{10}) = 9 \end{array}$$
>
> L'esecuzione dell'algoritmo è la seguente:
>
> $$O = [o_{10}, o_7, o_6, o_5, o_4, o_2, o_1, o_3, o_8, o_9]$$
> $$Q = B$$

Tabella 2.1 Esecuzione per ogni iterazione

Iterazione i	**Volume oggetto** $s(O[i])$	**Bin più vuoto** $\max_{j\in\{1,\dots,m\}} s(B_j)$	**Nuova capacità** $s(B_j) - s(O[i])$
1	9	$B_1,\ s(B_1) = 18$	$s(B_1) = 18 - 9 = 9$
2	9	$B_2,\ s(B_2) = 18$	$s(B_2) = 18 - 9 = 9$
3	7	$B_3,\ s(B_3) = 18$	$s(B_3) = 18 - 7 = 11$
4	5	$B_3,\ s(B_3) = 11$	$s(B_3) = 11 - 5 = 6$
5	5	$B_2,\ s(B_2) = 9$	$s(B_2) = 9 - 5 = 4$
6	5	$B_1,\ s(B_1) = 9$	$s(B_1) = 9 - 5 = 4$
7	4	$B_3,\ s(B_3) = 6$	$s(B_3) = 6 - 4 = 2$
8	4	$B_2,\ s(B_2) = 4$	$s(B_2) = 4 - 4 = 0$
9	3	$B_1,\ s(B_1) = 4$	$s(B_1) = 4 - 3 = 1$
10	3	$B_3,\ s(B_3) = 2$	$s(B_3) = 2 - 3 = -1$

Infine risulterà $B_1 = \{o_{10}, o_2, o_8\}, B_2 = \{o_7, o_4, o_3\}, B_3 = \{o_6, o_5, o_1, o_9\}$ quindi $s(B_1) = 0, s(B_2) = 1, s(B_3) = -1$ ed il valore tornato dall'algoritmo sarà $|0| + |1| + |-1| = 2$.

È da notare che la soluzione ottenuta non è ottima, in quanto assegnando gli oggetti ai bins nel modo seguente si otterrebbe il valore della funzione obiettivo uguale a 0:

$$B_1 = \{o_{10}, o_7\}, B_2 = \{o_5, o_6, o_8, o_9\}, B_3 = \{o_1, o_2, o_3, o_4\}.$$

Ci sono comunque dei casi in cui questo algoritmo trova la soluzione ottima sempre, ovvero quando c'è un oggetto o_k tale che $s(o_k) \geq \dfrac{\sum_{i=1, i\neq k}^{n} s(o_i)}{m-1}$.

Ad esempio se viene data in input un'istanza con m bins di capacità m e $n = m(m-1)+1$ oggetti e per i primi $n-1$ oggetti $s(o_i) = 1\ \forall i \in \{1, ..., n-1\}$ e per l'ultimo oggetto $s(o_n) = m$ l'algoritmo troverà la soluzione ottima. Infatti l'algoritmo assegnerebbe al primo bin l'oggetto o_n e i restanti $m-1$ bins sarebbero riempiti ad ogni iterazione con un oggetto di volume unitario m volte ciascuno, quindi alla fine delle iterazioni si ritroveranno saturati e la soluzione fornita dall'algoritmo avrà valore 0 che inoltre sarà anche quella ottima.

2.2.2 Algoritmo con euristica merging sugli oggetti

Questo secondo algoritmo utilizza un'euristica di merging sugli oggetti che si basa sull'idea di rendere il numero di oggetti e di bins uguale per poi assegnarne ognuno in modo arbitrario a un bin, per avere che i numeri di oggetti e di bins coincidano fonde insieme gli oggetti con volume minimo in modo da crearne uno nuovo il cui volume è la somma di quelli che lo compongono.
Per avere una soluzione quindi si può procedere nel seguente modo:

```
1: function binPackMERGING(B, O, s())
2:     Q = O
3:     while |B| ≠ |Q| do
4:         obj-meno-grande1 ← extractMin(Q, s())
5:         obj-meno-grande2 ← extractMin(Q, s())
6:         nuovo-obj ← obj-meno-grande1 ∪ obj-meno-grande2
7:         Q ← Q ∪ {nuovo-obj}
8:     for i = 1; i ≤ |B|; i = i + 1 do
9:         B_i = B_i ∪ {q_i}
10:    return Σ_{b∈B} |b|
```

L'algoritmo prende in input l'insieme di bins B, la lista di oggetti O e la funzione taglia $s()$. Dopo aver assegnato gli oggetti della lista O all'insieme Q procede con l'operazione di merging estraendo dall'insieme Q i due oggetti con volume minimo per poi unirli creando un nuovo oggetto, che avrà come volume la somma dei volumi degli altri due, e inserirlo in Q, questo fin quando non risulta $n = m$ (n numero di oggetti e m numero di bins). Una volta terminata l'operazione di merging si procede con l'assegnazione degli oggetti ai bin in maniera arbitraria ed infine viene ritornato il valore a cui si è interessati.

2.2.2.1 Analisi della complessità dell'algoritmo

L'istruzione al rigo `2` prende tempo costante, il while delle righe `3 - 7` viene eseguito $n - m$ volte, le istruzioni delle righe `4` e `5`, se l'insieme Q è un semplice insieme la ricerca deve essere effettuata in modo esaustivo, quindi l'operazione di estrazione del minimo ha complessità $\Theta(n)$, le altre istruzioni all'interno del `while` prendono tempo costante, il `for` delle righe `8 - 9` viene eseguito m volte e l'istruzione al suo interno prende tempo costante, infine l'espressione del `return` prende tempo $\Theta(m)$. Quindi la complessità dell'algoritmo in questo caso è $O(n(n-m))$ per via della ricerca esaustiva che viene fatta $n - m$ volte all'interno del `while`, ma se Q fosse uno heap l'operazione di estrazione del minimo prenderebbe tempo $\Theta(\log n)$, quella di inserimento in Q prenderebbe anch'essa tempo $\Theta(\log n)$ ma se ne trae vantaggio perché la complessità del `while` diventerebbe $O((n-m)\log n)$ che è minore di $O(n(n-m))$ e quindi la complessità dell'algoritmo diventerebbe proprio questa, ovvero dominata dalla ripetizione delle istruzioni all'interno del `while`. L'algoritmo sarà quindi implementato usando uno heap come struttura per Q.

2.2.2.2 Esempio di esecuzione

Anche se non verrà rispettata la condizione per cui $n >> m$ al fine di rendere più semplice l'esempio questo non rappresenta una perdita di generalità.

> Sia O una lista di lunghezza n contenente i vari oggetti, e $s()$ la funzione di taglia e sia B un insieme di cardinalità m contenente i vari bin ognuno di capacità c.
> Consideriamo la seguente istanza:
>
> $$O = \{o_1, ..., o_{10}\}, \text{ quindi } |O| = n = 10 \text{ e}$$
> $$B = \{B_1, B_2, B_3\}, \text{ quindi } |B| = m = 3,\ c = 18$$

$$s(o_1) = 4 \quad s(o_6) = 7$$
$$s(o_2) = 5 \quad s(o_7) = 9$$
$$s(o_3) = 4 \quad s(o_8) = 3$$
$$s(o_4) = 5 \quad s(o_9) = 3$$
$$s(o_5) = 5 \quad s(o_{10}) = 9$$

L'esecuzione dell'algoritmo è la seguente:

$$Q = O$$

Tabella 2.2 Esecuzione per ogni iterazione del while

Iterazione	**Oggetto meno grande 1**	**Oggetto meno grande 2**	**Nuovo oggetto** $s(nuovo\,obj)$
1	$o_9, s(o_9) = 3$	$o_8, s(o_8) = 3$	$o_{9\cup 8}, 6$
2	$o_3, s(o_3) = 4$	$o_1, s(o_1) = 4$	$o_{3\cup 1}, 8$
3	$o_2, s(o_2) = 5$	$o_4, s(o_4) = 5$	$o_{2\cup 4}, 10$
4	$o_5, s(o_5) = 5$	$o_{9\cup 8}, s(o_{9\cup 8}) = 6$	$o_{5\cup 9\cup 8}, 11$
5	$o_6, s(o_6) = 7$	$o_{3\cup 1}, s(o_{3\cup 1}) = 8$	$o_{6\cup 3\cup 1}, 15$
6	$o_7, s(o_7) = 9$	$o_{10}, s(o_{10}) = 9$	$o_{7\cup 10}, 18$
7	$o_{2\cup 4}, s(o_{2\cup 4}) = 10$	$o_{5\cup 9\cup 8}, s(o_{5\cup 9\cup 8}) = 11$	$o_{2\cup 4\cup 5\cup 9\cup 8}, 21$

Infine risulterà $B_1 = \{o_{6\cup 3\cup 1}\}, B_2 = \{o_{7\cup 10}\}, B_3 = \{o_{2\cup 4\cup 5\cup 9\cup 8}\}$ quindi $s(B_1) = 3, s(B_2) = 0, s(B_3) = -3$ ed il valore tornato dall'algoritmo sarà $|3| + |0| + |-3| = 6$.

È da notare che la soluzione ottenuta non è ottima, in quanto unendo gli oggetti e assegnandoli ai bins nel modo seguente si otterrebbe il valore della funzione obiettivo uguale a 0:

$$B_1 = \{o_{10\cup 7}\}, B_2 = \{o_{5\cup 6\cup 8\cup 9}\}, B_3 = \{o_{1\cup 2\cup 3\cup 4}\}.$$

Anche per questo algoritmo ci sono casi in cui trova la soluzione ottima, una condizione sufficiente affinché questa cosa si realizzi è quella di avere m bins e n oggetti tale che $n = 2^k, k \geq 2$ e $m = 2^h, 1 \leq h < k$ e che tutti gli oggetti abbiano taglia uguale, in particolare $s(o_i) = cm/n \; \forall i \in \{1, ..., n\}$. Ad esempio se $m = 2$, $c = 16$ e $n = 8$ allora ogni oggetto ha taglia 2, l'algoritmo assegnerà gli oggetti in modo ottimale, avendo valore della funzione obiettivo pari a 0.

Capitolo 3

Risultati sperimentali

3.1 Il metodo

Per confrontare i due algoritmi procedendo in modo sperimentale, si è implementato un software che producesse un'istanza per il problema per poi essere eseguita su entrambi gli algoritmi e produrre un grafico al fine di analizzare i risultati dati.

3.1.1 Generazione degli oggetti e dei bins

Come prima cosa, dopo aver fissato il numero n di oggetti e il numero m di bins, è stato necessario generare le rispettive taglie in base a qualche criterio in modo che rispettassero le condizioni del problema, ovvero che la somma delle taglie degli oggetti deve essere uguale alla somma delle taglie dei bins.

3.1.1.1 Una prima soluzione

La prima soluzione sviluppata per fare in modo che gli oggetti e i bins generati rispettassero le condizioni del problema è stata quella di scegliere la taglia degli m bins uguale a $1/m$ in modo che la loro somma fosse pari a 1, mentre per gli oggetti si è proceduto nel seguente modo:

1. Si generano n numeri $x_i \, \forall i \in \{1, ..., n\}$ in modo che ogni numero sia preso con probabilità uniforme in un intervallo di interi $[1, M]$.
2. Ogni oggetto o_i avrà taglia pari a $\frac{x_i}{\sum_{j=1}^{n} x_j} \, \forall i \in \{1, ..., n\}$.

Ovvero si è fatto in modo che la somma degli oggetti fosse normalizzata ad 1.

I problemi di questa soluzione
Nonostante questo criterio di generazione sembri essere buono ai fini degli esperimenti esso comporta dei problemi una volta realizzato su calcolatore, in quanto così facendo le taglie degli oggetti assumono valori molto piccoli, valori che non vengono sentiti nella somma tra le varie taglie, e di conseguenza falsano i risultati. Non è possibile fare un confronto tra i due algoritmi in quanto non effettuano le operazioni nello stesso ordine, un numero potrebbe essere sommato prima o

dopo ad un altro in base all' algoritmo eseguito, e quindi potrebbe amplificare o meno errori di calcolo, inoltre i numeri generati con questo metodo sono anche soggetti ad arrotondamenti e troncamenti vari al crescere di n e quindi si perde precisione.

3.1.1.2 Una seconda soluzione

Visti i problemi della prima soluzione illustrata si è cercato di ridurre gli errori dovuti a numeri troppo piccoli adoperando una tecnica diversa per la generazione degli oggetti ma simile alla prima, ovvero anziché normalizzare ad 1 la somma dei bins e quella degli oggetti si è preferito dare una capacità intera fissata ai bins uguale per tutti e generare le taglie degli oggetti nel seguente modo:

1. Si generano n numeri $x_i \, \forall i \in \{1, ..., n\}$ in modo che ogni numero sia preso con probabilità uniforme in un intervallo di interi $[1, M]$.
2. Ogni oggetto o_i avrà taglia pari a $\frac{x_i cm}{\sum_{j=1}^{n} x_j} \, \forall i \in \{1, ..., n\}$.

Ovvero si è fatto in modo che la somma degli oggetti fosse normalizzata a cm dove c è la capacità fissata di ogni bin e m il numero di bins.

I problemi di questa soluzione
Nonostante in questo caso si lavori con numeri più grandi anche questa soluzione comporta problemi relativi alle operazioni effettuate tramite calcolatore con valori "problematici" e di conseguenza anche questa tecnica falsa i risultati e non è possibile confrontare i risultati degli algoritmi.

3.1.1.3 Una terza soluzione

Al fine di eliminare totalmente errori dovuti a calcoli effettuati con numeri non sentiti nella somma su calcolatore si è cercato di trovare un modo di generare le taglie degli oggetti e dei bins per far sì che fossero solo interi, ovvero, una volta fissati n ed m:

1. Si generano n numeri $o_i \, \forall i \in \{1, ..., n\}$ in modo che ogni numero sia preso con probabilità uniforme in un intervallo di interi $[1, M]$.
   Sia $sumobj$ la somma delle taglie degli oggetti.
2. Se $sumobj$ non è un multiplo di m allora si calcola il resto della divisione tra $sumobj$ ed m, lo si sottrae ad m e il risultato di questa operazione lo si aggiunge alla taglia di un oggetto scelto a caso, se invece è un multiplo si esegue il prossimo passo direttamente.
3. La taglia dei bins diventa $sumobj/m$.

In questo modo si lavora solo con numeri interi e gli errori dovuti a numeri non sentiti nella somma con altri sono totalmente eliminati, di conseguenza i dati sono corretti ed è possibile confrontare gli algoritmi.

I problemi di questa soluzione
Anche se sono stati eliminati del tutto gli errori sui calcoli sorge un altro problema, ovvero quello dell'overflow, infatti se le taglie degli oggetti sono troppo grandi, o comunque al crescere di n, la somma di questi potrebbe provocare un'overflow. Questi problemi sono presenti anche nelle due soluzioni precedenti al crescere di n, anzi le due precedenti potrebbero portare anche ad underflow, ma ciò è trascurabile per la riuscita dell'esperimento in quanto non è una cosa che può essere risolta in generale sotto queste condizioni.

3.1.1.4 La strategia utilizzata

Si è scelto di utilizzare l'ultima soluzione descritta per generare gli oggetti e i bins in quanto non presenta nessun problema ai fini dell'esperimento e produce dati corretti, per cui è possibile confrontare i risultati degli algoritmi.

3.1.2 Raccolta dati

Per la parte di raccolta dati semplicemente si è definito un numero di esperimenti da eseguire e salvato i valori prodotti dagli algoritmi corrispondenti alle istanze generate dal metodo illustrato precedentemente basato sul numero di oggetti e bins relativi all'esperimento. Una volta prodotti questi dati essi sono stati riportati in una tabella che mette in corrispondenza i valori delle istanze e dei risultati degli algoritmi eseguiti su di esse.

3.1.3 Analisi grafica

Una volta prodotti i dati dell'esperimento, al fine di renderne più facile e immediata l'analisi, essi sono stati riportati su un piano tridimensionale dove sull'asse delle x si trova il numero di oggetti da immagazzinare, sull'asse delle y il numero di bins, e sull'asse delle z il valore risultato dall'esecuzione degli algoritmi. In questo modo è immediato confrontare i valori delle esecuzioni su una stessa istanza.

3.2 Aspetti tecnici dell'implementazione

Per realizzare il software desiderato sono state fatte scelte precise riguardo le tecnologie da utilizzare spiegate di seguito.

3.2.1 Linguaggio e librerie utilizzate

Il linguaggio utilizzato per la realizzazione del software è *Python*, le motivazioni di tale scelta sono da cercare nella semplicità di scrittura del codice data dall'alto livello di astrazione e dalla presenza di numerose librerie a supporto dell'utilizzo scientifico del linguaggio. Inoltre non ci sono particolari caratteristiche dell'esperimento che avrebbero prediletto altri linguaggi, ad esempio se si avesse dovuto utilizzare la concorrenza si sarebbe optato per un linguaggio come *Rust* o uno puramente funzionale come *Haskell*, se il tempo di esecuzione fosse stato cruciale si sarebbe optato per un linguaggio a basso livello come *C*, ma poiché non c'è da preoccuparsi di queste questioni *Python* va più che bene

per lo scopo.
Le librerie utilizzate all'interno del software sono:

heapq
Una libreria nativa del linguaggio che permette di utilizzare una lista come uno heap ed eseguire i classici algoritmi relativi ad esso sulla lista in modo efficiente. Usata per sfruttare il costo computazionale ridotto di uno heap per determinate operazioni al fine di ottimizzare gli algoritmi che ne fanno uso.

matplotlib
Una libreria di plotting per Python orientata agli oggetti, che permette di creare grafici di ogni tipo e visualizzarli interattivamente, esponendo una API molto simile a quella del linguaggio *MATLAB*.
Usata all'interno del software sviluppato per creare le tabelle e i grafici relativi agli esperimenti, in particolare si sono utilizzati i moduli `pyplot` e `Axes3D` da `mpl_toolkits.mplot3d`.

numpy
Libreria di supporto al calcolo scientifico per il linguaggio, che espone numerose funzioni matematiche di alto livello al fine di poterle utilizzare in modo efficiente, utilizzata anche all'interno di `matplotlib` per operare efficientemente con matrici e vettori di grandi dimensioni.
Usata nel software per creare liste di numeri generati casualmente tramite distribuzione uniforme, infatti si è utilizzato in particolare il modulo `random`.

3.2.2 Codice

Per la consultazione del codice si rimanda alla sezione dedicata nell'appendice.

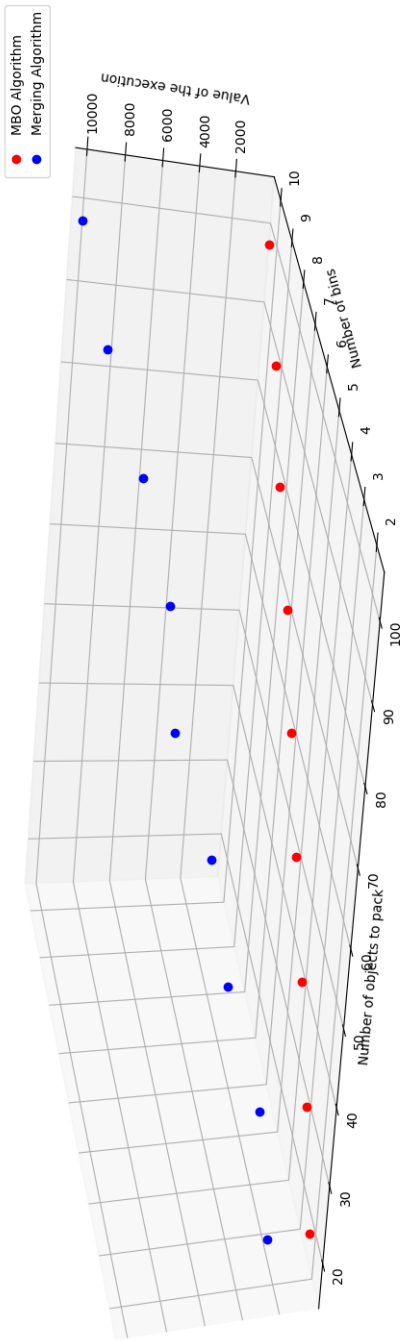
3.3 Dati sperimentali e interpretazione

Una volta delineato il metodo da utilizzare per l'esperimento e realizzato il software che permette di eseguirlo sono stati prodotti i dati relativi ad esso per essere analizzati.

3.3.1 Dati

Di seguito varie immagini raffiguranti gli esiti degli esperimenti basati su istanze generate casualmente con distribuzione uniforme negli intervalli specificati.

Figura 3.1: 9 esperimenti con $n = 10m$ in $[1, 1000]$



	MBO Algorithm	Merging Algorithm
Bins: 2, Objects: 20	156	2492
Bins: 3, Objects: 30	44	2660
Bins: 4, Objects: 40	48	4130
Bins: 5, Objects: 50	112	4752
Bins: 6, Objects: 60	94	6484
Bins: 7, Objects: 70	44	6462
Bins: 8, Objects: 80	206	7652
Bins: 9, Objects: 90	178	9294
Bins: 10, Objects: 100	290	10378

Figura 3.2: 9 esperimenti con $n = 100m$ in $[1, 1000]$

	MBO Algorithm	Merging Algorithm
Bins: 2, Objects: 200	2	17518
Bins: 3, Objects: 300	6	22038
Bins: 4, Objects: 400	6	31784
Bins: 5, Objects: 500	6	37392
Bins: 6, Objects: 600	10	46358
Bins: 7, Objects: 700	12	54026
Bins: 8, Objects: 800	2	61180
Bins: 9, Objects: 900	14	66022
Bins: 10, Objects: 1000	20	74404

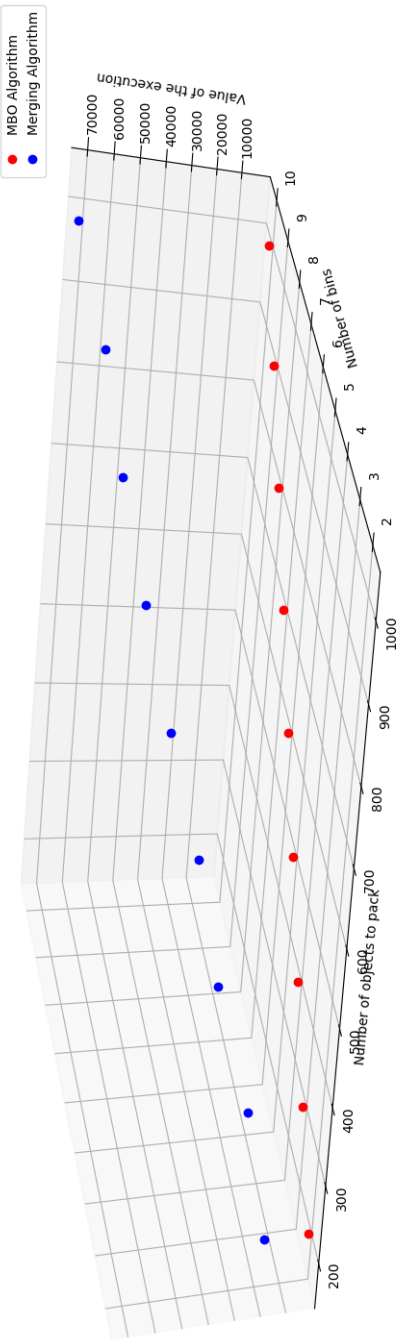
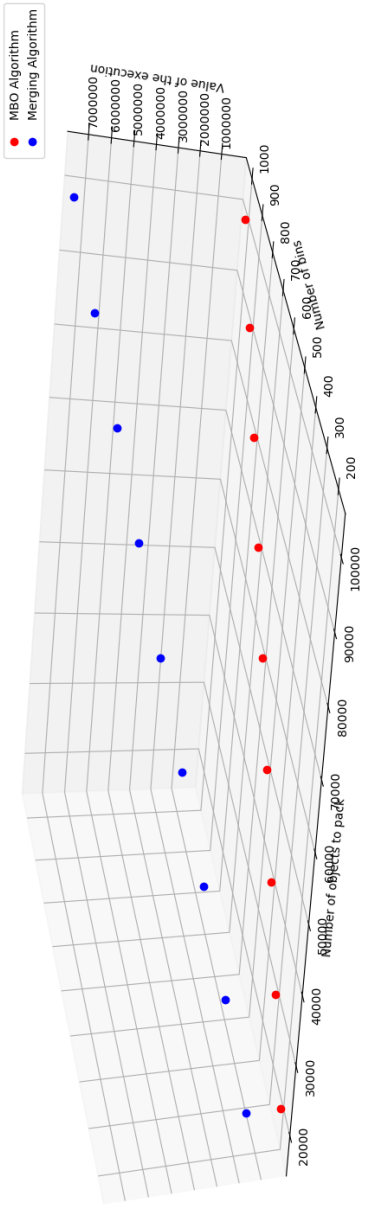


Figura 3.2.2: Altri 9 esperimenti con $n = 100m$ in $[1, 1000]$



	MBO Algorithm	Merging Algorithm
Bins: 200, Objects: 20000	214	1562560
Bins: 300, Objects: 30000	448	2328284
Bins: 400, Objects: 40000	308	3102120
Bins: 500, Objects: 50000	408	3882470
Bins: 600, Objects: 60000	298	4657464
Bins: 700, Objects: 70000	848	5446860
Bins: 800, Objects: 80000	848	6203094
Bins: 900, Objects: 90000	562	6997848
Bins: 1000, Objects: 100000	1496	7742372

Figura 3.3: 9 esperimenti con $n = 1000m$ in $[1, 1000]$

	MBO Algorithm	Merging Algorithm
Bins: 2, Objects: 2000	0	147996
Bins: 3, Objects: 3000	0	234944
Bins: 4, Objects: 4000	0	319114
Bins: 5, Objects: 5000	0	418114
Bins: 6, Objects: 6000	0	484920
Bins: 7, Objects: 7000	0	543256
Bins: 8, Objects: 8000	0	660508
Bins: 9, Objects: 9000	0	719610
Bins: 10, Objects: 10000	0	809170

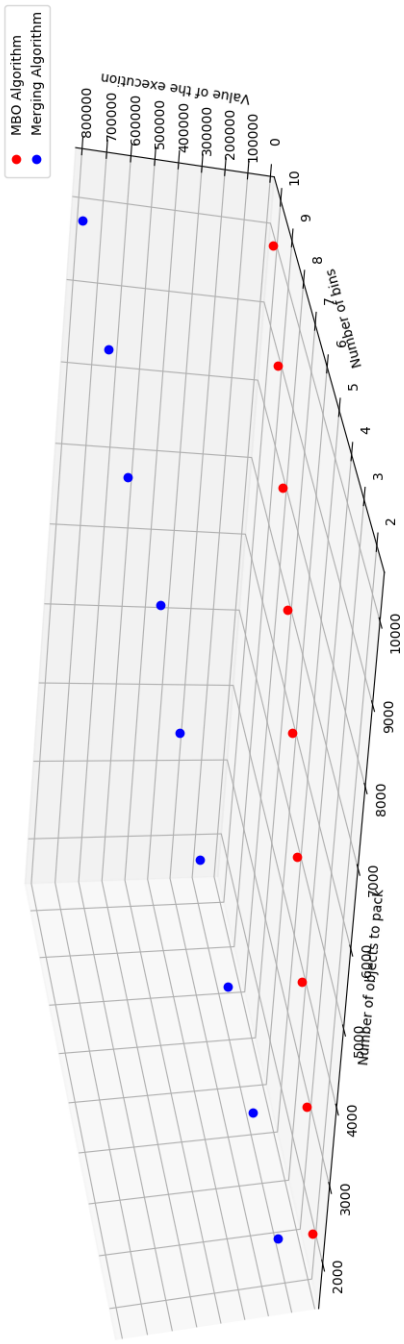


Figura 3.3.2: Altri 9 esperimenti con $n = 1000m$ in $[1, 1000]$

	MBO Algorithm	Merging Algorithm
Bins: 200, Objects: 200000	0	15989048
Bins: 300, Objects: 300000	0	23918126
Bins: 400, Objects: 400000	0	31957016
Bins: 500, Objects: 500000	0	39870144
Bins: 600, Objects: 600000	0	47993114
Bins: 700, Objects: 700000	0	55900914
Bins: 800, Objects: 800000	0	63986802
Bins: 900, Objects: 900000	0	71958760
Bins: 1000, Objects: 1000000	0	79952494

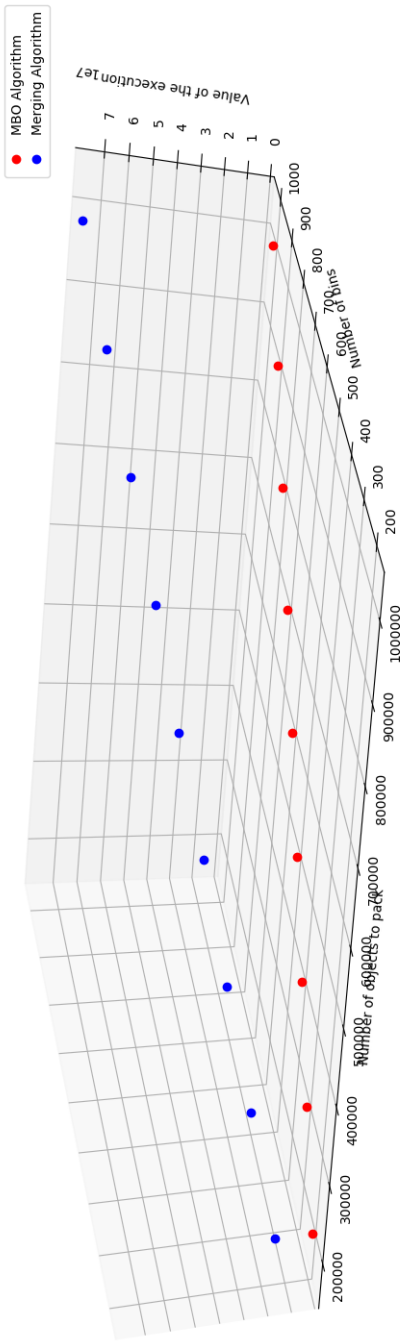


Figura 3.4: 5 esperimenti con $n = 10m$ progressivamente in $[1, 1000]$

	MBO Algorithm	Merging Algorithm
Bins: 3, Objects: 30	88	2576
Bins: 30, Objects: 300	450	29638
Bins: 300, Objects: 3000	1840	298720
Bins: 3000, Objects: 30000	5700	3032898
Bins: 30000, Objects: 300000	51996	30221016

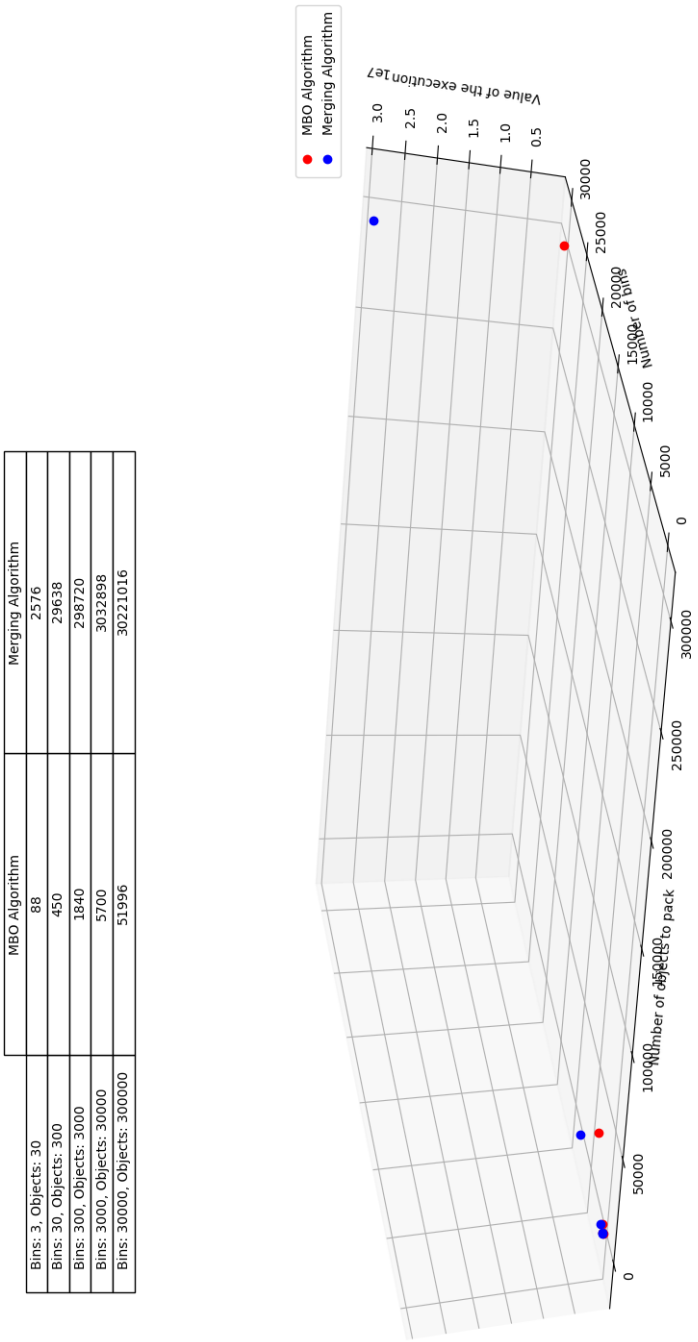
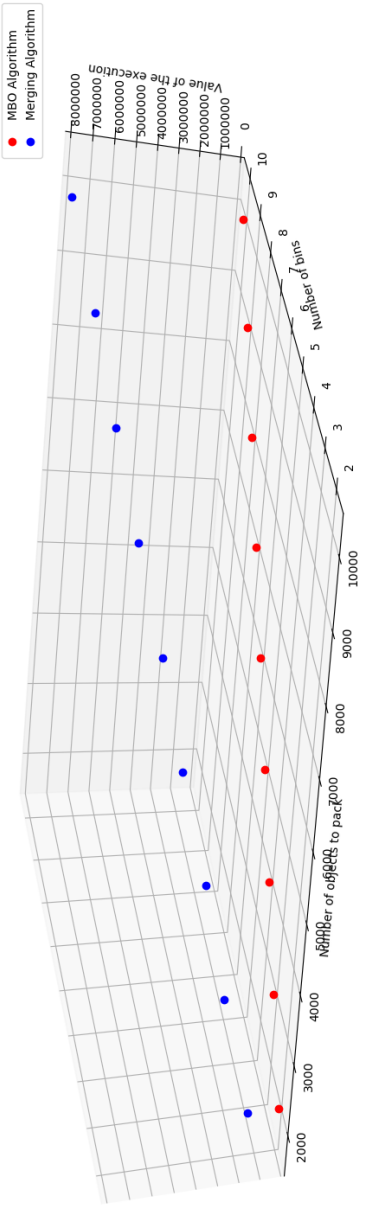


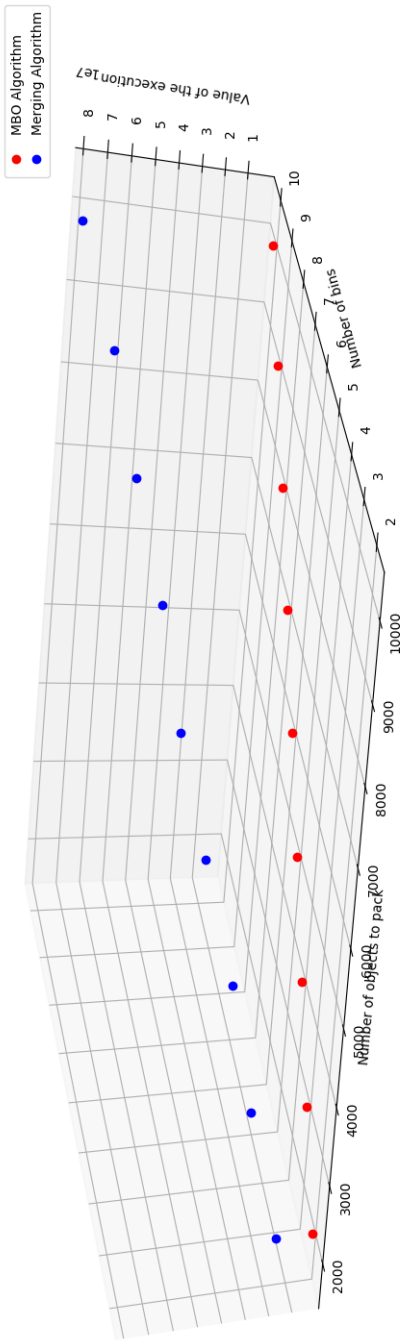
Figura 3.5: 9 esperimenti con $n = 1000m$ in $[1, 10000]$



	MBO Algorithm	Merging Algorithm
Bins: 2, Objects: 2000	0	1494528
Bins: 3, Objects: 3000	4	2394002
Bins: 4, Objects: 4000	4	3011228
Bins: 5, Objects: 5000	6	3941588
Bins: 6, Objects: 6000	10	4641004
Bins: 7, Objects: 7000	16	5568616
Bins: 8, Objects: 8000	10	6420134
Bins: 9, Objects: 9000	10	7174222
Bins: 10, Objects: 10000	24	8068218

Figura 3.6: 9 esperimenti con $n = 1000m$ in $[1, 100000]$

	MBO Algorithm	Merging Algorithm
Bins: 2, Objects: 2000	190	15815570
Bins: 3, Objects: 3000	18	24427756
Bins: 4, Objects: 4000	70	30126042
Bins: 5, Objects: 5000	30	39481964
Bins: 6, Objects: 6000	30	48228760
Bins: 7, Objects: 7000	48	53963996
Bins: 8, Objects: 8000	80	63042930
Bins: 9, Objects: 9000	136	70298646
Bins: 10, Objects: 10000	118	81434598



3.3.2 Interpretazione

Ciò che si nota a primo impatto dai risultati degli esperimenti è che l'algoritmo con euristica MBO, su istanze generate in modo casuale, dà sempre un risultato di gran lunga migliore rispetto all'algoritmo con euristica di merging sugli oggetti in ogni caso.
È possibile notare come al crescere del numero di oggetti rispetto al numero di bins, fissato l'intervallo di generazione, l'algoritmo MBO migliora sempre la propria soluzione (visibile analizzando questi esperimenti: Figura 1, Figura 2, Figura 3) arrivando addirittura a disporli perfettamente all'interno dei contenitori senza il minimo spreco o eccesso di spazio. Al contrario l'algoritmo di merging ha un comportamento opposto, ovvero peggiora sempre la propria soluzione.
Inoltre si nota come prendendo un intervallo più grande di generazione (Figura 5, Figura 6) l'algoritmo MBO non dispone sempre perfettamente gli oggetti nei contenitori ma dà comunque risultati migliori rispetto a quello di merging.

3.4 Conclusioni

Dopo aver effettuato i vari esperimenti ed analizzato i dati si può concludere dicendo che in generale, se gli oggetti hanno taglie "varie" e quindi l'istanza del problema non ha una particolare forma, l'algoritmo MBO dà risultati migliori rispetto a quello di merging. Quindi poiché la complessità temporale asintotica è praticamente la stessa si può affermare che conviene utilizzare il primo algoritmo menzionato in quanto ci darà una soluzione nettamente migliore rispetto al secondo.
È comunque da tenere in considerazione la natura dell'istanza, in quanto ce ne potrebbero essere alcune per cui gli algoritmi danno la stessa soluzione, oppure per cui la "distanza" tra le due soluzioni non sia così grande come si vede dagli esperimenti, o che addirittura, come visto in precedenza, producano quella ottima.

Bibliografia

[1] Michael R. Garey e David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. A Series of Books in the Mathematical Sciences. W. H. Freeman e Company, 1979. Cap. A4.1. ISBN: 0-7167-1045-5.

[2] Klaus Jansen et al. «Bin packing with fixed number of bins revisited». In: *Journal of Computer and System Sciences* 79 (2013), pp. 39–49. DOI: https://doi.org/10.1016/j.jcss.2012.04.004.

[3] Silvano Martello e Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990. Cap. 8.5. ISBN: 0-471-92420-2.

[4] Ethan L. Schreiber e Richard E. Korf. «Improved Bin Completion for Optimal Bin Packing and Number Partitioning». In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence* (2013), pp. 651–658. URL: https://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6851.

Appendice

Codice

```
import heapq
import matplotlib.pyplot as plt
import numpy as np
from copy import deepcopy
from mpl_toolkits.mplot3d import Axes3D


def bin_pack_mbo(objects, bins, size_of_bins):
    sorted_objects = sorted(objects)[::-1]
    bins_heap = deepcopy(bins)

    heapq.heapify(bins_heap)

    for obj in sorted_objects:
        most_empty_bin = heapq.heappop(bins_heap)
        most_empty_bin += obj
        heapq.heappush(bins_heap, most_empty_bin)

    return sum(abs(size_of_bins - a_bin) for a_bin in bins_heap)


def bin_pack_merging(objects, bins, size_of_bins):
    bins_list = deepcopy(bins)
    bins_length = len(bins_list)
    objects_heap = deepcopy(list(objects))

    heapq.heapify(objects_heap)

    while len(objects_heap) != bins_length:
        first_smaller_object, second_smaller_object =
        ↪  heapq.heappop(objects_heap),
        ↪  heapq.heappop(objects_heap)
        merged_object = first_smaller_object +
        ↪  second_smaller_object
        heapq.heappush(objects_heap, merged_object)

    for i, _ in enumerate(bins_list):
        bins_list[i] += heapq.heappop(objects_heap)

    return sum(abs(size_of_bins - a_bin) for a_bin in bins_list)


def calculate_size_of_bins(objects, number_of_bins,
↪  number_of_objects):
    sum_of_objects_size = sum(objects)

    if sum_of_objects_size % number_of_bins != 0:
        rest = sum_of_objects_size % number_of_bins
```

```
45          i = np.random.randint(0, number_of_objects)
46          objects[i] += number_of_bins - rest
47          sum_of_objects_size += number_of_bins - rest
48
49      size_of_bins = sum_of_objects_size // number_of_bins
50
51      return size_of_bins
52
53
54  def display_chart(objects_history, bins_history, results):
55      fig, ax = plt.subplots(2, 1, figsize=(16, 9),
        ↪  subplot_kw={"projection": "3d"})
56      fig.canvas.set_window_title("Bin Packing Problem Analysis")
57      ax[0].axis("off")
58      row_labels = [f"Bins: {n_bins}, Objects: {n_objects}" for
        ↪  n_bins, n_objects in zip(bins_history, objects_history)]
59
60      ax[0].table(cellText=[(mbo, merging) for mbo, merging in
        ↪  results],
61                  rowLabels=row_labels,
62                  colLabels=("MBO Algorithm", "Merging Algorithm"),
63                  cellLoc="center",
64                  loc="center",
65                  colWidths=[.2] * 3)
66      ax[1].plot(objects_history, bins_history, list(map(lambda
        ↪  element: element[0], results)),
67                 linestyle="None", marker="o", color="red",
                   ↪  label="MBO Algorithm")
68      ax[1].plot(objects_history, bins_history, list(map(lambda
        ↪  element: element[1], results)),
69                 linestyle="None", marker="o", color="blue",
                   ↪  label="Merging Algorithm")
70
71      ax[1].set_xlabel("Number of objects to pack")
72      ax[1].set_ylabel("Number of bins")
73      ax[1].set_zlabel("Value of the execution")
74
75      plt.legend()
76      plt.tight_layout()
77      plt.show()
78
79
80  def main():
81      results = []
82      objects_history = []
83      bins_history = []
84
85      number_of_experiments = int(input("How many experiments do
        ↪  you want to do? "))
86
```

```
87      for experiment in range(number_of_experiments):
88          print(f"\nExperiment number {experiment + 1}")
89
90          number_of_bins = int(input("Insert number of bins: "))
91          number_of_objects = int(input("Insert number of objects
            ↪  to pack: "))
92
93          objects = np.random.randint(1, 1001,
            ↪  size=number_of_objects)
94
95          size_of_bins = calculate_size_of_bins(objects,
            ↪  number_of_bins, number_of_objects)
96          bins = [0] * number_of_bins
97
98          mbo_algorithm_result = bin_pack_mbo(objects, bins,
            ↪  size_of_bins)
99          merging_algorithm_result = bin_pack_merging(objects,
            ↪  bins, size_of_bins)
100
101         objects_history += [number_of_objects]
102         bins_history += [number_of_bins]
103         results += [(mbo_algorithm_result,
            ↪  merging_algorithm_result)]
104
105     display_chart(objects_history, bins_history, results)
106
107
108 if __name__ == "__main__":
109     main()
```