Algoritmi Euristici
One Dimensional Bin Packing Problem

Università degli Studi Di Milano

Marco Odore

26 giugno 2017

Indice

1 Introduzione

Lo scopo del lavoro è quello di proporre una possibile implementazione in C di diversi metodi euristici applicati al problema del *One Dimensional Bin Packing*, per la ricerca di soluzioni ottime o che comunque vi si avvicinano.

1.1 One Dimensional Bin Packing

Dato un multiset di n oggetti $O = \{o_1, o_2, o_3 \dots o_n\}$, ognuno con dimensione d_i, lo scopo è quello di minimizzare il numero di contenitori b_j (bin) $M = \{b_1, b_2, b_3 \dots b_n\}$, ognuno con dimensione fissata B, che contengono tali oggetti.

Il problema è soggetto a diversi vincoli:

- Ogni oggetto deve essere inserito in un solo contenitore.
- La somma delle dimensioni d_i degli oggetti o_i, nel contenitore b_j, non deve superare la dimensione del contenitore.
$$\sum_{o_i \in b_j} d_i \leq B$$
- Il numero dei contenitori b_j deve essere il minimo possibile. Si cercherà quindi di minimizzare tale funzione:
$$min \sum_{j=1}^{n} y_j$$
In cui y_i è una variabile binaria associata agli n possibili contenitori b_j (il caso peggiore contempla un contenitore per ogni oggetto presente nel multi insieme).

  Secondo la teoria della complessità, tale problema ha complessità *NP-hard*. Per tale motivo sono state studiate diverse tecniche euristiche, con lo scopo di ottenere un trade-off tra velocità di esecuzione e ottimalità delle soluzioni generate.

2 Euristiche implementate

Per la risoluzione del problema sono state implementate due principali euristiche costruttive greedy:

- FirstFit
- Minimum Bin Slack (MBS)

Che poi sono servite da base per altre due meta euristiche:

- MBS Sampling
- Variable Neighbour Search (VNS)

2.1 FirstFit

Tale algoritmo è molto banale, e si basa sull'idea greedy che, scorrendo iterativamente la lista di oggetti, se nel contenitore b_j corrente c'è abbastanza spazio, allora vi si inserisce l'oggetto corrente o_i. Altrimenti, se non c'è spazio tra i contenitori attualmente presenti, se ne genera uno nuovo.

Algorithm 1 FirstFit

```
1:  for obj in objectList do
2:      for bin in binList do
3:          if obj fit in bin then
4:              Pack object in bin
5:              break
6:          end if
7:      end for
8:      if obj did not fit in any available bin then
9:          Create new bin and pack object in it
10:     end if
11: end for
```

Nel caso peggiore (quando ogni oggetto può essere inserito in un solo contenitore) tale algoritmo ha complessità $O(n^2)$.

Una variante di questo algoritmo, il *FirstFit Decreasing*, prende in considerazione l'idea che posizionare oggetti grandi sia più difficile che posizionarne di piccoli, e consiste nell' ordinamento decrescente della lista di oggetti del dataset, prima dell'esecuzione del FirstFit.

2.2 Minimum Bin Slack

L'MBS è un'euristica greedy orientata sui contenitori. L'algoritmo consiste nel mantenere, ad ogni passo, una lista di oggetti Z non ancora inseriti, con un ordinamento decrescente, ricercando tra questi l'insieme di oggetti che meglio riempiono il contenitore corrente (idealmente non lasciando spazio libero).

La ricerca del sottoinsieme di oggetti da inserire nel contenitore corrente, avviene tramite una procedura ricorsiva, la quale testa tutti i possibili sottoinsiemi della lista Z. Se durante la ricerca si trova una soluzione che riempie totalmente il contenitore, questa viene interrotta, poiché non ci può essere una soluzione migliore per il contenitore corrente, ma al massimo equivalente (da notare comunque che il sottoinsieme generato potrebbe non essere ottimo per la soluzione globale).

Di seguito lo pseudocodice dell'algoritmo utilizzato nel lavoro[1] :

Algorithm 2 MBSsearch

```
   Procedure MBSsearch(q)
 2: for int r = q to n do
       o_r = Z[r]
 4:    if size(o_r) ≤ slack(A) then
          A = A ∪ {o_r}
 6:       MBSsearch(r + 1)
          A = A − {o_r}
 8:       if slack(A*) = 0 then
             exit
10:       end if
       end if
12: end for
   if slack(A) < slack(A*) then
14:    A* = A
   end if
```

Dato che in linea teorica questa procedura cerca tutte le possibili combinazioni di elementi da inserire in un contenitore, questa ha complessita $O(2^n)$. Nella pratica è possibile velocizzare l'algoritmo con alcuni accorgimenti. Ad esempio, se lo slack del sottoinsieme corrente è più piccolo del più piccolo elemento dell'insieme di oggetti, cioè che $slack(A) < size(obj_{min})$, allora il ciclo for viene saltato, in quanto per il sottoinsieme A non esiste miglioramento possibile.

Una variante di questo algoritmo è il *modified MBS* (definito semplicemente come MBS'), che crea dei sottoinsiemi per i contenitori del problema alla stessa maniera dell' MBS *base*, con la differenza che, prima di eseguire la procedura di ricerca, un oggetto preso dalla lista Z viene sempre fissato all'interno del contenitore corrente.

A differenza dell'MBS *base* che rischia di sfruttare subito gli oggetti di piccola dimensione, l'MBS' si forza nel fissare almeno un oggetto grande (il più grande tra i rimanenti nella lista Z, al momento corrente), generando soluzioni differenti, e potenzialmente migliori.

2.3 MBS Sampling

Questo algoritmo invoca diverse volte l'euristica MBS', cambiando l'ordine degli oggetti nella lista Z ad ogni esecuzione, e adottando la soluzione migliore trovata.

L'ordinamento del vettore Z è basato probabilisticamente sull'ordine decrescente delle dimensioni degli oggetti, con la probabilità di selezionare un oggetto o_i e di riempire il vettore ordinato ad ogni step, pari a:

$$p_i = \frac{size(o_i)^2}{\sum_{o_j \in A} size(o_j)^2}$$

Dove A in questo caso rappresenta il multiset di oggetti o_j non ancora inseriti nella lista.

[1] A e A^* indicano, rispettivamente, gli insiemi corrente e il migliore. q è l'indice da cui si parte per la ricerca ricorsiva nella lista Z. $slack(A)$ è una funzione che ci dice di quanto dista la somma delle dimensioni degli oggetti di A rispetto alla capacità del contenitore, mentre $size(o_i)$ ci dice la dimensione dell'oggetto o_i.

2.4 Variable Neighbourhood Search

Questo tipo di meta euristica sfrutta la variazione sistematica ed incrementale dell'intorno di una soluzione di partenza, applicando poi degli algoritmi di ricerca del minimo, come ad esempio l'algoritmo di *discesa del gradiente*.

La tecnica si compone di tre fasi principali, iterate:

1. Shaking
2. Local Search
3. Move or not

Shaking - Questa fase consiste nel perturbare una soluzione x di partenza (presa ad esempio dall'MBS') per muoversi nel suo intorno $N(x)$ generando una nuova soluzione x'. L'intorno incrementale della soluzione x di partenza $N_k(x)$ viene definito dal numero di mosse k utilizzate per ottenere la perturbazione.

Le mosse utilizzate possono essere di due tipi, e cioè di *swap* o di *transfer*. Lo swap consiste nello scambiare due oggetti presenti in due contenitori differenti, mentre il trasferimento porta allo spostamento di un oggetto da un contenitore ad un altro. Chiaramente queste mosse devono essere sensate e consentite. Cioè devono portare ad una effettiva variazione della soluzione e non ne devono violare i vincoli[2].

Local Search - La ricerca locale di un minimo nell'intorno avviene tramite la *discesa del gradiente*. In questo caso specifico è utile pensare alla funzione obiettivo come ad una funzione da massimizzare piuttosto che da minimizzare (numero di contenitori). Viene infatti massimizzata la somma quadratica della dimensione degli oggetti in un contenitore:

$$max\ f(x) = \sum_{a=1}^{m} (l(a))^2$$

Dove $l(a)$ indica la somma delle dimensioni degli oggetti presenti nel bin a. La discesa avviene tramite una ricerca iterativa ad ogni step della miglior mossa possibile di swap o di transfer, che migliora la funzione obiettivo. La ricerca termina se non esistono mosse possibile che migliorano la soluzione.

Move or not - Se la ricerca locale ha portato ad un effettivo miglioramento della soluzione, ci si muove verso la nuova soluzione, facendola diventare quella corrente e si prova ad esplorare il suo intorno, altrimenti si prova ad allargare l'intorno della soluzione originale. L'allargamento dell'intorno avviene tramite l'aumento del numero di mosse utilizzate per perturbare la soluzione ($k = k+1$). Quando si trova una soluzione migliore invece, si ritorna ad un intorno più piccolo ($k = 1$). Chiaramente bisogna settare un confine all'intorno, fissando un k_{max}, che se raggiunto termina la ricerca.

3 Dataset di test

Il dataset utilizzato[1] per testare i diversi algoritmi è suddiviso in 2 classi principali. La prima, che va dal *binpack*1 al *binpack*4, consiste di oggetti uniformemente distribuiti in $(20, 100)$ (interi), che devono essere inseriti in contenitori di dimensione 150. La seconda classe, composta dai file che vanno da *binpack*5 a *binpack*8, consiste invece di oggetti uniformemente distribuiti in $(25, 50)$(reali), che devono essere inseriti in contenitori di dimensione 100.

Sia per la prima classe che per la seconda classe vi sono 80 problemi differenti, i quali variano per il numero di oggetti totali da inserire nei contenitori.

[2]Ad esempio scambiare due oggetti uguali non modifica la soluzione.

4 Valutazione degli algoritmi

Per la valutazione dei diversi algoritmi si sono utilizzate diverse metriche. In particolar modo, per valutarne l'efficacia si sono utilizzate:

- Media
- Varianza

Come statistiche descrittive classiche.
Mentre per verificare la robustezza dell'algoritmo sulle diverse istanze, si è utilizzata la Solution Quality Distribution, la quale ci ha permesso inoltre di confrontare le diverse performarce.

4.1 Metriche Classiche

4.2 SQD

Riferimenti bibliografici

[1] E.Falkenauer, *"A Hybrid Grouping Genetic Algorithm for Bin Packing"*, Working paper CRIF Industrial Management and Automation, CP 106 - P4, 50 av. F.D.Roosevelt, B-1050 Brussels (1994)