2D bin packing

Lorenzo Santina
Tommaso Toscano

Indice

1. Bin Packing Problem
    a. 2D bin packing
    b. Utilizzi reali
2. BSA
    a. Soluzione iniziale Greedy
    b. Intorni intra bin
        i. Spostare un rettangolo inserito in un rettangolo libero
    c. Intorni inter bin
        i. Spostare scaffali da un bin a un altro
        ii. Spostare i rettangoli dai bin piu' vuoti a quelli piu' pieni
    d. Tabu list
    e. Ricerca locale
3. Test

Bin packing

Un problema di packing è caratterizzato da:

- contenitori
- un insieme di oggetti

L'obiettivo dei problemi di packing può essere:

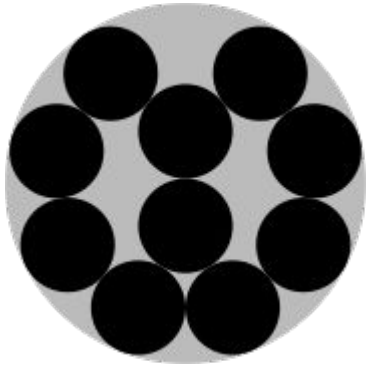
- Riempire il contenitore il più densamente possible oppure
- Inserire tutti gli oggetti utilizzando il minor numero di contenitori possibili.

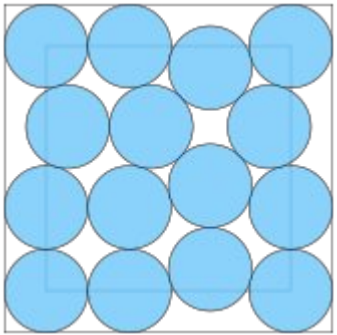
In particolare i Bin Packing problem, inseriscono oggetti di dimensioni diverse all'interno di un numero finito di contenitori, in maniera tale da minimizzare il numero di contenitori.

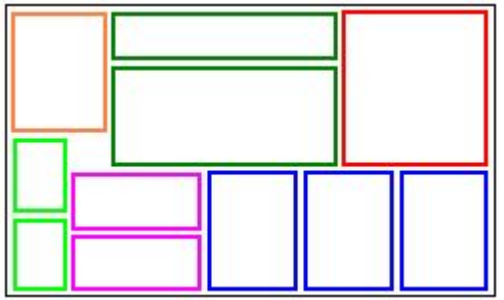
2D bin packing

I contenitori hanno 2 dimensioni.

Noi abbiamo approfondito il packing di rettangoli diversi in bin rettangolari perche' siamo interessato alle sue applicazioni nel taglio dei vetri.



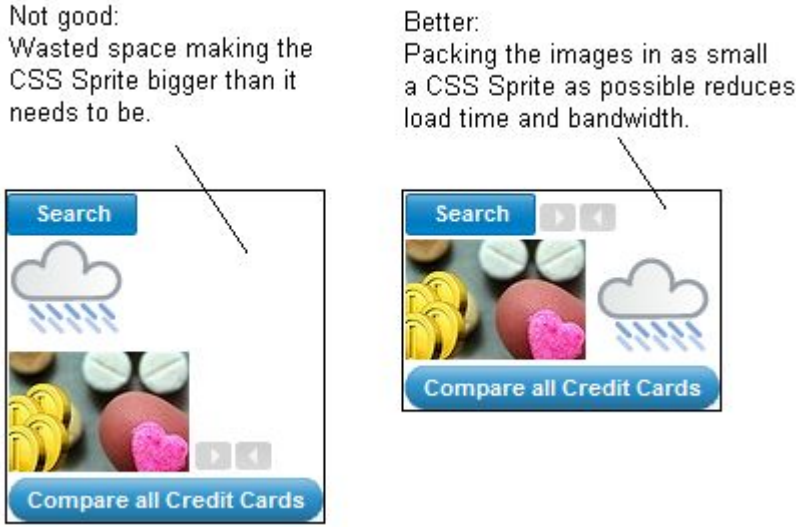




Utilizzi reali

Ottimizzazione immagini web

Per rendere più veloce il caricamento delle immagini nei siti web, possono essere combinate in una sola grande immagine.
Per ottimizzare il caricamento dell'immagine risultante, bisogna disporre le immagini in modo da minimizzare le sue dimensioni.



Cutting stock problem

Il Bin Packing viene anche molto utilizzato nelle industrie per tagliare: vetri, cavi e carta.

Soluzione iniziale Greedy

1) Tutti i rettangoli vengono orientati in verticale e ordinati per altezza e larghezza
2) Vengono inseriti in scaffali di altezza uguale al prossimo rettangolo più alto da inserire.
3) Si tiene traccia dello spazio inutilizzato ad ogni scaffale con una lista di “rettangoli liberi”

Intorni

- Intra bin
  1) Spostare un rettangolo inserito in un rettangolo libero
  2) Scambiare i rettangoli da uno scaffale all'altro
  3) Ruotare un rettangolo inserito
- Inter bin
  4) Spostare scaffali da un bin a un altro
  5) Spostare i rettangoli dai bin piu' vuoti a quelli piu' pieni
  6) Scambiare i rettangoli da un bin all'altro

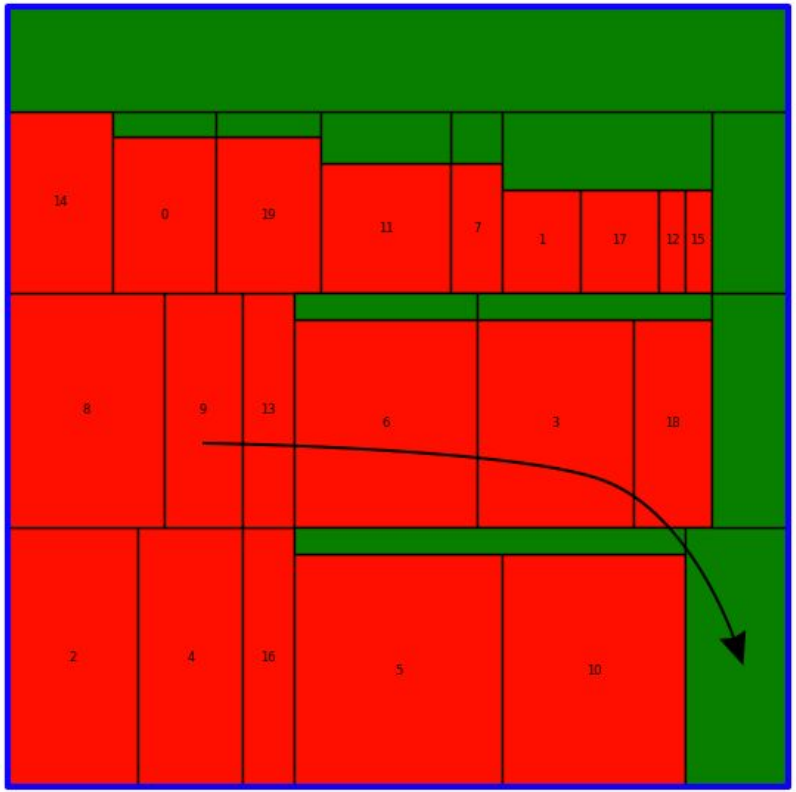
Abbiamo implementato gli intorni 1, 4 e 5

Spostare un rettangolo inserito in un rettangolo libero

Si cerca un rettangolo che possa essere spostato nello scaffale sottostante cosi da:

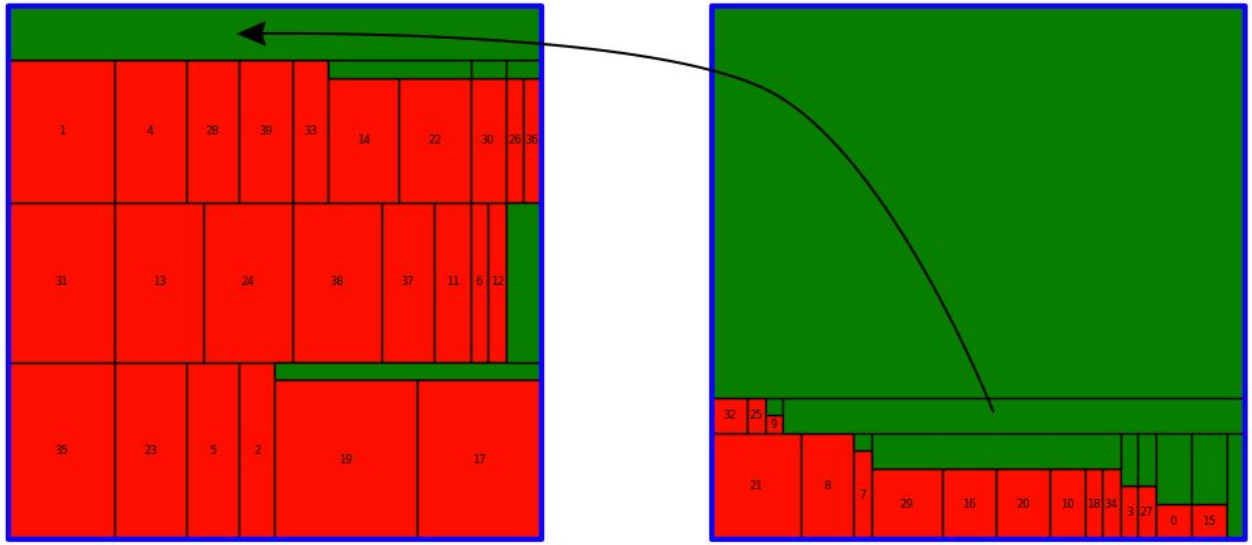
- riempire meglio lo scaffale sotto
- liberare spazio nel suo scaffale

I rettangoli dello scaffale di origine vengono fatti scorrere verso sinistra



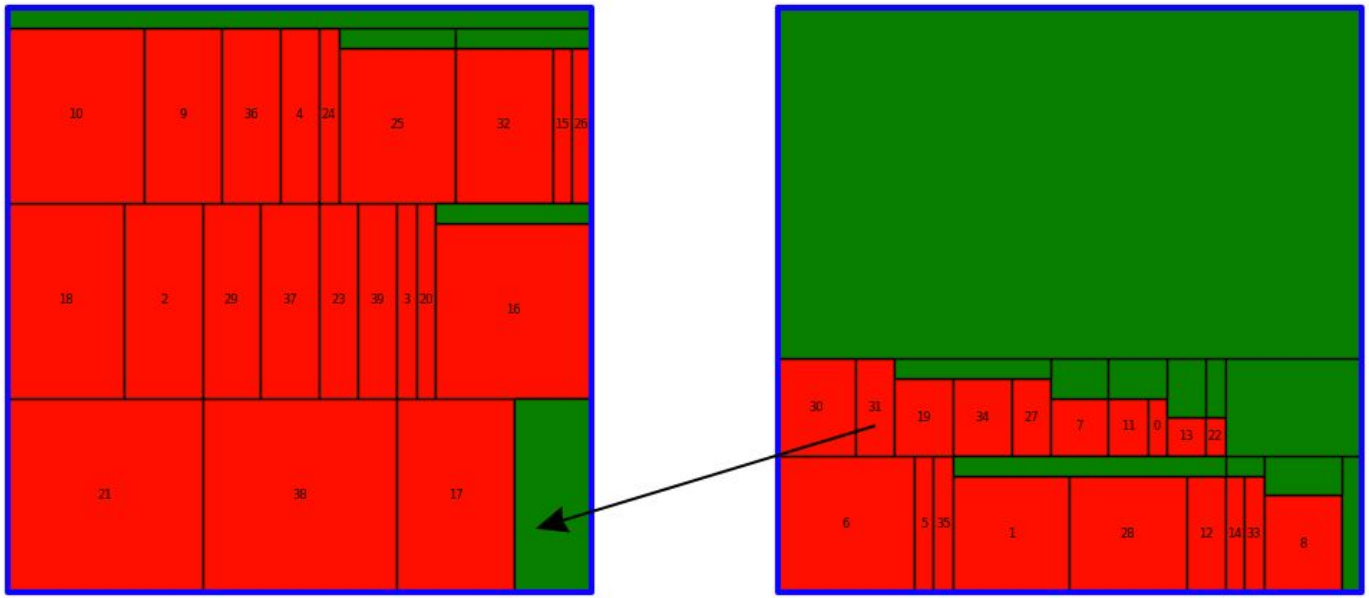
Spostare scaffali da un bin a un altro

1) I container vengono riordinati per spazio diposnibile in altezza (crescente)
2) Gli scaffali vengono riordinati per altezza (crescente)
3) Si considerano solo i container pieni piu' della meta'
4) Partendo dal container con minor spaizo libero, si cerca lo scaffale piu' alto che ci stia dentro (cosi da riempire il container il piu' possibile)



Spostare i rettangoli dai bin piu' vuoti a quelli piu' pieni

1) I container vengono ordinati in base al rapporto area_occupata/numero_item (per trovare quelli con tanti item piccoli)
2) Gli item vengono ordinati per area (crescente)
3) Partendo dall'item piu' piccolo, lo si cerca di spostare in un altro bin con sufficiente area libera
4) La tabulist permette di non ripetere le mosse appena fatte



Tabu list

La tabulist permette di non ripetere le mosse appena fatte

Ad ogni mossa si controlla che:

- non sia gia' presente nella tabu list
- non sia il "reverse" di una mossa gia' in tabu list

```
class TabooList:
    def __init__(self):
        self.moves=dict()

    def insert(self, move):
        if move not in self.moves:
            self.moves[move]=15
        self.reduce()

    def reduce(self):
        deleted=None

        for move, count in self.moves.items():
            self.moves[move]-=1
            if(self.moves[move]==0):
                deleted=move

        if(deleted):
            self.moves.pop(deleted)

    def contains(self, move):
        for m in self.moves:
            if(move.compare(m)):
                return True
        return False
```

Ricerca Locale

Vengono applicati gli intorni in sequenza finche non e' piu' possibile fare spostamenti

Inoltre e' stato imposto un limite di 1000 iterazioni per evitare che un ciclo di mosse continui all'infinito.

```
def bsa(instance):
    instance.reset()
    greedyShelf(instance)

    n1=True
    n2=True
    n3=True

    cont = 1000
    while( (n1 or n2 or n3) and cont > 0):
        n1=intraNeighborhood(instance)
        if(not n1):
            n2=interShelfborhood(instance)
            if(not n2):
                n3=interRectangle(instance)
        cont -= 1
```

Codice Soluzione iniziale

1) Orientamento in verticale e ordinamento per altezza e larghezza

```
rectangles = rotateWide(rectangles)
rectangles.sort(key=lambda x: (x.height,x.width), reverse=True)
```

2) Inserimento in scaffale

```
sh=Shelf(cont.width, shelf_height, cont.height-available_height)
cont.shelves.append(sh)
available_height-=shelf_height
while(i < len(rectangles) and sh._item_fits_shelf(rectangles[i])):
    sh.insert(rectangles[i])
    cont.items.append(rectangles[i])
    i+=1
```

3) Spazio inutilizzato

```
if available_height < shelf_height or i >= len(rectangles):
    freeRect = FreeRectangle(cont.width, available_height, cont.x, cont.height-available_height)
    cont.wastemap.freerects.add(freeRect)
```

Test

Abbiamo usato i dataset presi da questi link:

http://or.dei.unibo.it/sites/or.dei.unibo.it/files/instances/MV_2bp.zip

http://or.dei.unibo.it/sites/or.dei.unibo.it/files/instances/BW_2bp.zip

Migliori risultati conosciuti

http://or.dei.unibo.it/general-files/best-known-solution-and-lower-bound-each-instance