Euristiche costruttive (1/2)

Indice

Perché le euristiche costruttive?

Euristiche costruttive

I problemi e le ipotesi

Il paradigma greedy

Il problema dello zaino 0/1: quale ordinamento?

Il problema del bin packing: prima ordinare, poi riempire
- First Fit Decreasing
- Best Fit Decreasing
- Che cosa dà l'ordinamento, e che cosa non dà

Scheduling su macchine parallele: prima i lavori più lunghi
- List scheduling
- Longest processing time first

Obiettivi

Alla fine di questo capitolo saremo in grado di:

- applicare l'algoritmo greedy al KP con i tre ordinamenti (profitto, peso, efficienza);
- eseguire FF, FFD e BFD per il BPP, e LIST e LPT per $P \,||\, C_{\max}$;
- costruire un giro del TSP con il nearest neighbour e con le euristiche di inserimento;
- colorare un grafo con la colorazione greedy e con *DSATUR*;
- applicare la costruzione randomizzata, l'inserimento con regret e il metodo pilota;
- confrontare i risultati con i bound duali, e spiegare quando un'euristica costruttiva fallisce.

Prerequisiti

- Da questo modulo, il capitolo «Algoritmi euristici: definizioni»: bound primali e duali, il gap e le quattro istanze di riferimento.
- Dal modulo *Problemi di ottimizzazione combinatoria*, i problemi: il KP, il BPP, $P \,||\, C_{\max}$, il TSP e il VCP.

Perché le euristiche costruttive?

Introduzione: Perché le euristiche costruttive?

- I problemi su cui lavoriamo sono NP-difficili: per istanze grandi una soluzione ottima può essere fuori portata, ma serve comunque una buona soluzione ammissibile
- Un'euristica costruttiva è il modo più naturale per ottenerne una: costruisce la soluzione stessa, un pezzo alla volta
- Le sue soluzioni sono anche i punti di partenza dei metodi che migliorano una soluzione completa, che vedremo più avanti nel corso

Costruire una soluzione passo dopo passo

- Un'**euristica costruttiva** costruisce una soluzione ammissibile passo dopo passo: parte da una soluzione vuota e la estende con un elemento alla volta, finché la soluzione non è completa.
- In questo capitolo studiamo come si sceglie l'elemento da aggiungere, e perché questa scelta decide la qualità del risultato.

Riepilogo

Euristiche costruttive:

- costruiscono una soluzione ammissibile, un elemento alla volta
- partono da una soluzione vuota e si fermano quando la soluzione è completa

La domanda del capitolo:

- come si sceglie, a ogni passo, l'elemento da aggiungere?
- perché questa scelta decide la qualità del risultato?

Euristiche costruttive

Introduzione: Euristiche costruttive

- In questa sezione diamo la definizione formale di euristica costruttiva
- Fissa le parole usate in tutto il capitolo: **soluzione parziale** ed **elemento**
- Il resto del capitolo studia come si sceglie l'elemento da aggiungere

Euristica costruttiva

Definizione 1: Euristica costruttiva

*Un'**euristica costruttiva** è un algoritmo euristico che costruisce una soluzione ammissibile passo dopo passo: parte dalla **soluzione parziale** vuota e la estende con un elemento alla volta, finché la soluzione non è completa.*

- Gli elementi dipendono dal problema: gli oggetti selezionati per lo zaino, gli assegnamenti degli oggetti ai contenitori e dei lavori alle macchine, gli archi di un giro.
- La qualità del risultato dipende da come si sceglie, a ogni passo, l'elemento da aggiungere (il paradigma greedy, nelle prossime sezioni).

Riepilogo

Euristica costruttiva:

- parte dalla soluzione parziale vuota
- la estende con un elemento alla volta, finché la soluzione non è completa

Elementi:

- oggetti, assegnamenti degli oggetti ai contenitori e dei lavori alle macchine, archi di un giro
- come si sceglie l'elemento decide la qualità del risultato

I problemi e le ipotesi

Introduzione: I problemi e le ipotesi

- In questa sezione richiamiamo i problemi su cui illustriamo le euristiche
- Fissiamo la notazione usata da tutte le euristiche del capitolo
- Richiamiamo le quattro istanze di riferimento, di cui sono noti i valori ottimi: ci permettono di misurare quanto è buona ogni euristica

I problemi di questo capitolo

- Illustriamo le euristiche sui quattro problemi del capitolo «Algoritmi euristici: definizioni»:
  - il problema dello zaino 0/1 (KP),
  - il problema del bin packing (BPP),
  - il problema del commesso viaggiatore (TSP),
  - $P \,||\, C_{\max}$;
- la colorazione greedy e *DSATUR* si applicano al problema della colorazione dei vertici (VCP).
- Le loro definizioni e ipotesi sono nelle rispettive pagine; qui richiamiamo la notazione.

Notazione

- Come nelle altre note, $\mathscr{A}(I)$ indica il valore della soluzione calcolata dall'euristica $\mathscr{A}$ sull'istanza I, e $z(I)$ il valore ottimo.
- Per $P \,||\, C_{\max}$, $L_h = \sum_{j \in J_h} t_j$ è il **carico** della macchina h, e il vettore *mach* memorizza la macchina $mach_j$ di ogni lavoro j.
- Per il BPP, il vettore *bin* memorizza il contenitore bin_j di ogni oggetto j, e *rc* indica una capacità residua.

Le istanze di riferimento

Gli esempi usano le quattro **istanze di riferimento** del capitolo «Algoritmi euristici: definizioni», di cui sono noti i valori ottimi:

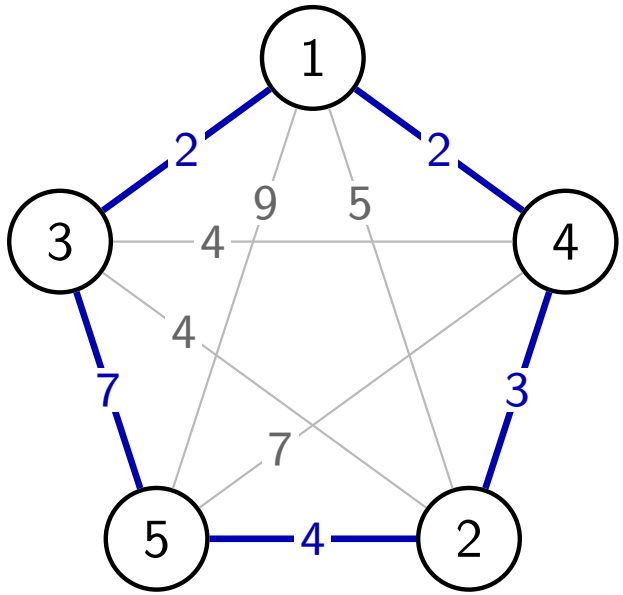
problema	istanza di riferimento	ottimo
KP	$p = (10, 7, 6, 4)$, $w = (5, 4, 3, 3)$, $c = 9$	$\{1, 2\}$, $z(I) = 17$
BPP	$w = (4, 4, 5, 3, 2, 3)$, $c = 7$	$\{1, 4\}, \{2, 6\}, \{3, 5\}$, $z(I) = 3$
$P \mid\mid C_{\max}$	$t = (5, 5, 4, 4, 3, 3, 3)$, $m = 3$	$\{1, 3\}, \{2, 4\}, \{5, 6, 7\}$, $z(I) = 9$
TSP	5 città, distanze qui sotto	$\sigma = (1, 3, 5, 2, 4)$, $z(I) = 18$

L'istanza di riferimento del TSP (1/2)

- Nelle figure, le cinque città dell'istanza di riferimento del TSP sono disegnate sempre nelle stesse posizioni, su un pentagono, nell'ordine $(1, 3, 5, 2, 4)$ del giro ottimo.
- Il giro ottimo è quindi il bordo del pentagono, e ogni giro che usa una diagonale ne è visibilmente diverso.
- La prossima figura riporta le distanze dell'istanza e, in grassetto, il suo giro ottimo.

L'istanza di riferimento del TSP (2/2)

d_{ij}	1	2	3	4	5
1	—	5	2	2	9
2	5	—	4	3	4
3	2	4	—	4	7
4	2	3	4	—	7
5	9	4	7	7	—



Riepilogo

I problemi:

- il KP, il BPP, il TSP e $P \,||\, C_{\max}$; il VCP per la colorazione greedy e *DSATUR*
- $\mathscr{A}(I)$ è il valore dell'euristica $\mathscr{A}$, $z(I)$ il valore ottimo

Le istanze di riferimento:

- quattro istanze con valori ottimi noti: 17, 3, 9 e 18
- le cinque città dell'istanza del TSP sono disegnate su un pentagono, il cui bordo è il giro ottimo

Il paradigma greedy

Introduzione: Il paradigma greedy

- In questa sezione descriviamo il modo più semplice di scegliere l'elemento da aggiungere: prendere quello che sembra migliore adesso
- Scriviamo uno schema di pseudocodice che ogni euristica del capitolo specifica
- Discutiamo l'ordine delle scelte, i criteri statici e dinamici, la miopia, il tempo di esecuzione, e quando il greedy è esatto

Tre domande a ogni passo

- Una soluzione di ciascuno dei quattro problemi è fatta di **elementi**: gli oggetti selezionati per lo zaino, gli assegnamenti degli oggetti ai contenitori e dei lavori alle macchine, gli archi di un giro. Un'euristica costruttiva parte dalla **soluzione parziale** vuota e aggiunge un elemento alla volta.

- A ogni passo bisogna rispondere a tre domande:
  - **quali elementi si possono aggiungere** (i candidati),
  - **quale si aggiunge** (il criterio di selezione),
  - **se il risultato si può ancora completare in una soluzione ammissibile** (il test di ammissibilità).

- Un'euristica **greedy** risponde alla seconda domanda nel modo più semplice: aggiunge il candidato che sembra migliore *adesso*, secondo un criterio che misura il beneficio immediato dell'elemento.

Algoritmo greedy

Definizione 2: Algoritmo greedy

*Un **algoritmo greedy** è un'euristica costruttiva che costruisce una soluzione selezionando ripetutamente, fra gli elementi candidati, il migliore secondo un criterio di selezione; l'elemento si aggiunge alla soluzione parziale se il test di ammissibilità lo consente, e si scarta altrimenti.*

In un algoritmo greedy ogni decisione si prende una volta per tutte:

una scelta, una volta fatta, non si rivede mai.

Lo pseudocodice che segue è uno **schema** più che un algoritmo.

L'insieme dei candidati, il criterio e il test sono le tre parti che ogni euristica del capitolo definisce in modo specifico.

Algorithm: Schema greedy

Input : un'istanza I, l'insieme C degli elementi delle soluzioni

Output : una soluzione ammissibile S

```
1  function Greedy(I)
2      S := ∅                          // soluzione parziale vuota
3      while C ≠ ∅ and S non è completa do
4          e := il migliore elemento di C secondo il criterio di
               selezione
5          C ← C \ {e}    // e si esamina una sola volta
6          if S ∪ {e} supera il test di ammissibilità then
7              S ← S ∪ {e}      // la scelta è definitiva
8          end
9      end
10     return S
11 end
```

L'ordine decide tutto

- Lo schema esamina gli elementi nell'ordine indotto dal criterio di selezione.
- Un elemento rifiutato o accettato presto non si può riconsiderare dopo.
- Due euristiche greedy con gli stessi candidati e lo stesso test di ammissibilità, ma con criteri diversi, possono produrre soluzioni molto diverse.
- Cambiare il criterio è la principale scelta di progetto.

Criteri statici e dinamici

- Un criterio è **statico** se il valore di un elemento non dipende dalla soluzione parziale:
  - gli elementi si possono allora ordinare una sola volta all'inizio, e l'euristica scorre la lista ordinata;
  - esempi: efficienza degli oggetti del KP, peso degli oggetti del BPP, tempo di esecuzione dei lavori.

- Un criterio è **dinamico** se il valore di un elemento cambia man mano che la soluzione cresce:
  - va ricalcolato a ogni passo;
  - esempi: distanza dall'ultima città di un giro parziale, capacità residua dei contenitori, carico delle macchine.

Miopia e tempo di esecuzione

- **Miopia.** Il criterio guarda solo all'effetto immediato di una scelta, non alle sue conseguenze sulle scelte ancora da fare.
  - È ciò che rende veloci le euristiche greedy, ed è anche ciò che le fa fallire: una scelta buona adesso può costringere a scelte cattive dopo.

- **Tempo di esecuzione.**
  - Con un criterio statico il tempo di esecuzione è di solito dominato dall'ordinamento, $O(n \log n)$, più il costo dei test di ammissibilità.
  - Con un criterio dinamico è il numero di passi per il costo di rivalutare i candidati, di solito $O(n^2)$ o $O(n^3)$.

Quando il greedy è esatto

- Per alcuni problemi un algoritmo greedy trova sempre una soluzione ottima:
  - l'algoritmo di Kruskal per il problema dell'albero ricoprente di costo minimo esamina gli archi per costo non decrescente e accetta un arco se non chiude un ciclo;
  - il problema dello zaino frazionario si risolve all'ottimo prendendo gli oggetti per efficienza non crescente.

- Per i quattro problemi NP-difficili di questo capitolo nessun algoritmo greedy è esatto, e la domanda diventa *quale* criterio dà buone soluzioni.

Lo schema per i quattro problemi (1/2)

La tabella mostra come si specifica lo schema per i quattro problemi; le sezioni successive li studiano uno per uno.

problema	elemento	criterio di selezione	test di ammissibilità
KP	un oggetto	profitto, peso o efficienza (statico)	l'oggetto entra nella capacità residua
BPP	un oggetto e un contenitore	oggetto: ordine di input o peso (statico); contenitore: il primo, il più pieno, il più vuoto (dinamico)	l'oggetto entra nel contenitore

Lo schema per i quattro problemi (2/2)

problema	elemento	criterio di selezione	test di ammissibilità
$P \mid\mid C_{\max}$	un lavoro e una macchina	lavoro: ordine di input o tempo di esecuzione (statico); macchina: la meno carica (dinamico)	sempre ammissibile
TSP	una città e la sua posizione	la più vicina, la più lontana, la più economica (dinamico)	sempre ammissibile

Riepilogo

Il paradigma greedy:

- tre domande a ogni passo: i candidati, il criterio di selezione, il test di ammissibilità
- un algoritmo greedy aggiunge il candidato che sembra migliore adesso; una scelta non si rivede mai

Progetto:

- l'ordine indotto dal criterio decide tutto
- criteri statici (si ordina una volta) e criteri dinamici (si ricalcola a ogni passo)
- miopia: veloce, ma una scelta buona adesso può costringere a scelte cattive dopo
- per i quattro problemi NP-difficili nessun algoritmo greedy è esatto

Il problema dello zaino 0/1: quale ordinamento?

Introduzione: Il problema dello zaino 0/1: quale ordinamento?

- In questa sezione applichiamo il paradigma greedy al KP
- Il test di ammissibilità è fissato, quindi l'euristica è determinata dall'ordine degli oggetti: confrontiamo tre ordinamenti naturali
- Vediamo che ogni ordinamento può essere arbitrariamente cattivo, e come combinare le soluzioni ci protegge

Euristiche greedy per il KP

- Per il KP gli elementi sono gli oggetti, e il test di ammissibilità è il vincolo di capacità: un oggetto si seleziona se il suo peso non supera la capacità residua rc.
- Poiché il test è fissato, un'euristica greedy per il KP è completamente determinata dall'**ordine** in cui si esaminano gli oggetti.
- Tre ordinamenti sono naturali: per profitto, per peso e per efficienza.

Tre ordinamenti

per **profitto** (GR_p):	p_j non crescente	prima gli oggetti di maggior valore
per **peso** (GR_w):	w_j non decrescente	quanti più oggetti possibile
per **efficienza** (GR):	$\frac{p_j}{w_j}$ non crescente	miglior valore per unità di peso

I pareggi si risolvono a favore dell'oggetto con l'indice minimo.

- Il greedy per efficienza si indica con GR; qui lo confrontiamo con gli altri due ordinamenti.

Algoritmo greedy con ordinamento ρ

- L'*algoritmo greedy con ordinamento* ρ è un'euristica costruttiva per il KP.
- Esamina gli oggetti nell'ordine $\rho = (\rho_1, \ldots, \rho_n)$, una permutazione di $\mathscr{I}$, partendo dallo zaino vuoto.
- Seleziona ogni oggetto se e solo se entra nella capacità residua.

Algorithm: Greedy con ordinamento (1/2)

Input : istanza del KP con n oggetti, profitti $p[1..n]$, pesi $w[1..n]$, capacità c; ordinamento $\rho[1..n]$

Output: soluzione ammissibile del KP $x[1..n]$

```
1 function GreedyOrderKP(n, c, w, p, ρ)
2 |   for j := 1 to n do
3 |   |   x[j] ← 0   // all'inizio l'oggetto j non è
  |   |    selezionato
4 |   end
5 |   rc := c                          // capacità residua
```

Algorithm: Greedy con ordinamento (2/2)

function *GreedyOrderKP*(n, c, w, p, ρ) (continua)

```
6   |   for k := 1 to n do
7   |   |   j := ρ[k]                    // k-esimo oggetto
    |   |    nell'ordinamento
8   |   |   if w[j] ≤ rc then
9   |   |   |   x[j] ← 1      // selezionare l'oggetto j
10  |   |   |   rc ← rc − w[j]          // aggiornare la
    |   |   |    capacità residua
11  |   |   end
12  |   end
13  |   return x
14  end
```

Ogni criterio coglie un aspetto

- Le tre euristiche differiscono solo per l'ordinamento ρ, calcolato una sola volta ordinando gli oggetti in tempo $O(n \log n)$.
- Ogni criterio coglie un aspetto di una buona soluzione e ignora l'altro:
  - il profitto ignora quanta capacità consuma un oggetto;
  - il peso ignora quanto vale un oggetto;
  - l'efficienza bilancia i due, ma ignora quanto bene gli oggetti riempiono lo zaino.

Esempio: tre ordinamenti sull'istanza di riferimento del KP (1/3)

Sull'istanza di riferimento del KP ($c = 9$) i tre ordinamenti sono

criterio	valori	ordinamento ρ
profitto	$p = (10, 7, 6, 4)$	$(1, 2, 3, 4)$
peso	$w = (5, 4, 3, 3)$	$(3, 4, 2, 1)$
efficienza	$\frac{p}{w} = \left(\frac{10}{5}, \frac{7}{4}, \frac{6}{3}, \frac{4}{3}\right) = (2,\ 1.75,\ 2,\ 1.33)$	$(1, 3, 2, 4)$

(gli oggetti 3 e 4 hanno lo stesso peso, gli oggetti 1 e 3 la stessa efficienza, e i pareggi si risolvono con l'indice).

Esempio: tre ordinamenti sull'istanza di riferimento del KP (2/3)

Le esecuzioni, con la capacità residua dopo ogni passo, sono:

	passo 1	passo 2
GR_p	oggetto 1: dentro, $rc = 4$	oggetto 2: dentro, $rc = 0$
GR_w	oggetto 3: dentro, $rc = 6$	oggetto 4: dentro, $rc = 3$
GR	oggetto 1: dentro, $rc = 4$	oggetto 3: dentro, $rc = 1$

	passo 3	passo 4
GR_p	oggetto 3: fuori	oggetto 4: fuori
GR_w	oggetto 2: fuori	oggetto 1: fuori
GR	oggetto 2: fuori	oggetto 4: fuori

Esempio: tre ordinamenti sull'istanza di riferimento del KP (3/3)

Quindi

$$GR_p(I) = p_1 + p_2 = 17,$$
$$GR_w(I) = p_3 + p_4 = 10,$$
$$GR(I) = p_1 + p_3 = 16.$$

- Il greedy per profitto trova la soluzione ottima $\{1, 2\}$, che riempie esattamente lo zaino.
- Il greedy per efficienza, dopo gli oggetti 1 e 3, resta con $rc = 1$, troppo poco per qualunque altro oggetto.
- Il greedy per peso prende i due oggetti leggeri, che sono anche quelli di minor valore.

Nessun ordinamento è il migliore (1/2)

- Su questa istanza vince il criterio "ingenuo". Non è una regola generale: nessun ordinamento è il migliore su ogni istanza, e **ciascuno dei tre ordinamenti può essere arbitrariamente cattivo**. Sia $c \geq 3$:
  - **profitto**: un oggetto con $p_1 = 2$, $w_1 = c$ e c oggetti con $p_j = 1$, $w_j = 1$. GR_p prende l'oggetto 1, che riempie lo zaino: $GR_p(I) = 2$, mentre i c oggetti leggeri insieme danno $z(I) = c$;
  - **peso**: due oggetti con $p_1 = 1$, $w_1 = 1$ e $p_2 = c$, $w_2 = c$. GR_w prende l'oggetto 1 e l'oggetto 2 non entra più: $GR_w(I) = 1$, mentre $z(I) = c$;

Nessun ordinamento è il migliore (2/2)

Sempre con $c \geq 3$:

- **efficienza**: due oggetti con $p_1 = 2$, $w_1 = 1$ e $p_2 = c$, $w_2 = c$: qui $GR(I) = 2$ e $z(I) = c$.

In tutti e tre i casi il rapporto

$$\frac{z(I)}{\mathscr{A}(I)}$$

cresce con c.

Combinare due soluzioni

- Un rimedio è **combinare** due soluzioni: l'algoritmo Ext-Greedy GR^e restituisce la migliore fra la soluzione di GR e il miglior oggetto singolo.
- Questa semplice combinazione garantisce

$$\frac{z(I)}{GR^e(I)} \leq 2.$$

- Eseguire più euristiche greedy e tenere la soluzione migliore è un modo generale ed economico per proteggersi dalla debolezza di ogni singolo criterio.

Riepilogo

Greedy per il KP:

- il test di ammissibilità è fissato: l'euristica è determinata dall'ordine degli oggetti
- tre ordinamenti: profitto (GR_p), peso (GR_w), efficienza (GR)
- sull'istanza di riferimento: $GR_p(I) = 17$, $GR_w(I) = 10$, $GR(I) = 16$

Nessun ordinamento è il migliore:

- ciascuno dei tre ordinamenti può essere arbitrariamente cattivo
- Ext-Greedy, la migliore fra GR e il miglior oggetto singolo, garantisce $\frac{z(I)}{GR^e(I)} \leq 2$

Il problema del bin packing: prima ordinare, poi riempire

Introduzione: Il problema del bin packing: prima ordinare, poi riempire

- In questa sezione scegliamo per il BPP l'ordine degli oggetti, non solo il contenitore
- Studiamo due euristiche, First Fit Decreasing e Best Fit Decreasing, e le eseguiamo sull'istanza di riferimento
- Vediamo che cosa dà l'ordinamento (una garanzia migliore nel caso peggiore) e che cosa non dà (una soluzione migliore su ogni istanza)

Euristiche greedy nell'ordine di input

- Quattro euristiche greedy classiche per il BPP elaborano gli oggetti **nell'ordine di input** e differiscono per la regola con cui scelgono il contenitore:
  - Next Fit (NF) tiene aperto un solo contenitore;
  - First Fit (FF), Best Fit (BF) e Worst Fit (WF) tengono tutti i contenitori e scelgono, fra quelli in cui l'oggetto entra, il primo aperto, il più pieno o il più vuoto.

- Sull'istanza di riferimento del BPP ($c = 7$, $w = (4, 4, 5, 3, 2, 3)$, $z(I) = 3$) danno

$$NF(I) = 5, \qquad FF(I) = 4, \qquad BF(I) = 3, \qquad WF(I) = 4.$$

Prima ordinare, poi riempire

- In queste euristiche il *contenitore* si sceglie con un criterio, ma l'*oggetto* no: gli oggetti arrivano nell'ordine dell'input, che può essere un cattivo ordine.
- Quando l'intera istanza è nota in anticipo — il caso *offline* — niente ci obbliga ad accettare quell'ordine.
- L'idea di questa sezione è scegliere anche l'ordine degli oggetti:

  prima ordinare, poi riempire.

Perché prima gli oggetti grandi

- Gli oggetti difficili di un'istanza del BPP sono quelli grandi: due oggetti più grandi di $\frac{c}{2}$ non stanno mai nello stesso contenitore, e un oggetto grande entra solo in un contenitore ancora quasi vuoto.
- Gli oggetti piccoli, al contrario, possono riempire i vuoti lasciati in qualunque contenitore.
- Se gli oggetti piccoli arrivano prima, si distribuiscono sui contenitori e occupano lo spazio che servirà dopo agli oggetti grandi.
- Se gli oggetti grandi arrivano prima, ciascuno di essi ha il proprio contenitore, e gli oggetti piccoli servono poi a riempire i vuoti.

First Fit Decreasing

- Il *First Fit Decreasing Algorithm* (indicato con FFD) è un'euristica costruttiva per il BPP.
- Prima ordina gli oggetti per peso non crescente, poi li inserisce in quest'ordine con First Fit:
  - ogni oggetto va nel primo contenitore, nell'ordine in cui i contenitori sono stati aperti, in cui entra;
  - se non entra in nessun contenitore, se ne apre uno nuovo.

Algorithm: First Fit Decreasing (1/3)

Input : istanza I del BPP con n oggetti, pesi $w[1..n]$, capacità c

Output: assegnamento oggetti-contenitori $bin[1..n]$ e numero di contenitori usati nb

```
1 function FirstFitDecreasing(n, w, c)
2 |   ρ := gli oggetti ordinati per peso non crescente
  |    // prima ordinare
3 |   bin := nuovo vettore[1..n]          // assegnamento
  |    oggetti-contenitori
4 |   rc := nuovo vettore[1..n]      // capacità residue
5 |   nb := 0                      // numero di contenitori
```

Algorithm: First Fit Decreasing (2/3)

function *FirstFitDecreasing*(n, w, c) (continua)

```
6   |   for k := 1 to n do
7   |   |   j := ρ[k]          // poi riempire: k-esimo
    |   |    oggetto più grande
8   |   |   s := −1            // contenitore selezionato
9   |   |   for b := 1 to nb do
10  |   |   |   if w[j] ≤ rc[b] then
11  |   |   |   |   s ← b
12  |   |   |   |   break
13  |   |   |   end
14  |   |   end
```

Algorithm: First Fit Decreasing (3/3)

```
function FirstFitDecreasing(n, w, c) (continua)
    for k := 1 to n do (continua)
15      if s = −1 then
16          nb ← nb + 1            // aprire un nuovo
              contenitore
17          rc[nb] ← c
18          s ← nb
19      end
20      bin[j] ← s                 // assegnare l'oggetto
21      rc[s] ← rc[s] − w[j]
22  end
23  return bin, nb
24 end
```

FFD e FF

- Il ciclo di riempimento è esattamente il ciclo di FF: l'unica differenza è la riga che calcola l'ordinamento ρ.
- L'ordinamento costa $O(n \log n)$, e FFD richiede lo stesso tempo di FF, $O(n^2)$ con la semplice scansione dei contenitori.

Esempio: FF e FFD sull'istanza di riferimento del BPP (1/4)

FF, nell'ordine di input:

- gli oggetti 1, 2 e 3 (pesi $4, 4, 5$) aprono un contenitore ciascuno, perché nessuna coppia di essi entra insieme;
- l'oggetto 4 (peso 3) va nel contenitore 1 ($rc[1] = 0$);
- l'oggetto 5 (peso 2) va nel contenitore 2 ($rc[2] = 1$);
- l'oggetto 6 (peso 3) non entra in nessun contenitore, le cui capacità residue sono $0, 1, 2$, e apre il contenitore 4.

Quindi $FF(I) = 4$.

Esempio: FF e FFD sull'istanza di riferimento del BPP (2/4)

FFD. I pesi ordinati per valore non crescente sono $5, 4, 4, 3, 3, 2$, cioè $\rho = (3, 1, 2, 4, 6, 5)$ (pareggi per indice). Quindi:

1. Oggetto 3 (peso 5): nessun contenitore è aperto, FFD apre il contenitore 1; $rc[1] = 7 - 5 = 2$.
2. Oggetto 1 (peso 4): $4 > rc[1] = 2$, FFD apre il contenitore 2; $rc[2] = 7 - 4 = 3$.
3. Oggetto 2 (peso 4): $4 > rc[1] = 2$, $4 > rc[2] = 3$, FFD apre il contenitore 3; $rc[3] = 7 - 4 = 3$.

Esempio: FF e FFD sull'istanza di riferimento del BPP (3/4)

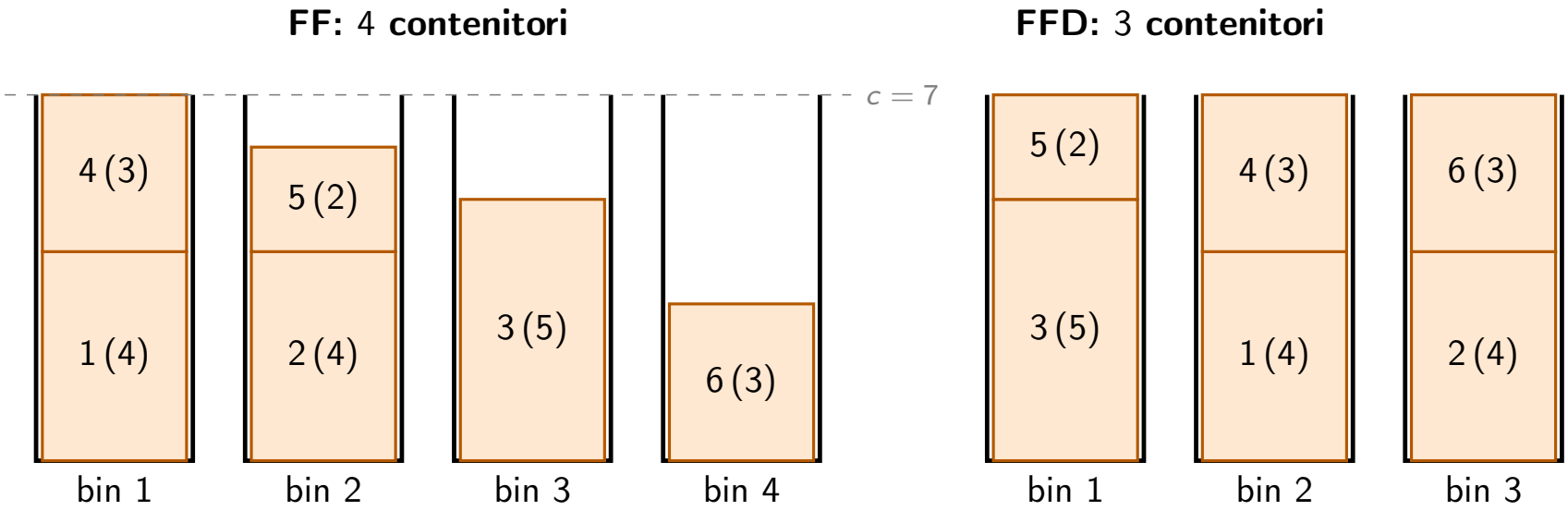
4. Oggetto 4 (peso 3): $3 > rc[1] = 2$, $3 \leq rc[2] = 3$: contenitore 2; $rc[2] = 0$.
5. Oggetto 6 (peso 3): $3 > rc[1] = 2$, $3 > rc[2] = 0$, $3 \leq rc[3] = 3$: contenitore 3; $rc[3] = 0$.
6. Oggetto 5 (peso 2): $2 \leq rc[1] = 2$: contenitore 1; $rc[1] = 0$.

Quindi $bin = (2, 3, 1, 2, 1, 3)$, i pattern di riempimento sono

$$P_1 = \{3, 5\}, \qquad P_2 = \{1, 4\}, \qquad P_3 = \{2, 6\},$$

e $FFD(I) = 3 = \left\lceil \frac{21}{7} \right\rceil$: la soluzione è ottima, e tutti e tre i contenitori sono pieni.

Esempio: FF e FFD sull'istanza di riferimento del BPP (4/4)



- In FF il piccolo oggetto 5 arriva prima del grande oggetto 6 e prende lo spazio che servirebbe all'oggetto 6 nel contenitore 2.
- In FFD l'oggetto 5 arriva per ultimo e riempie il vuoto del contenitore 1.

Best Fit Decreasing

- Il *Best Fit Decreasing Algorithm* (indicato con BFD) è un'euristica costruttiva per il BPP.
- Prima ordina gli oggetti per peso non crescente, poi li inserisce in quest'ordine con Best Fit:
  - ogni oggetto va nel contenitore con la capacità residua minima fra quelli in cui entra (in caso di pareggio, il contenitore aperto per primo);
  - se non entra in nessun contenitore, se ne apre uno nuovo.

Algorithm: Best Fit Decreasing (1/3)

Input : istanza I del BPP con n oggetti, pesi $w[1..n]$, capacità c

Output: assegnamento oggetti-contenitori $bin[1..n]$ e numero di contenitori usati nb

```
1 function BestFitDecreasing(n, w, c)
2 |   ρ := gli oggetti ordinati per peso non crescente
  |    // prima ordinare
3 |   bin := nuovo vettore[1..n]          // assegnamento
  |    oggetti-contenitori
4 |   rc := nuovo vettore[1..n]      // capacità residue
5 |   nb := 0                      // numero di contenitori
```

Algorithm: Best Fit Decreasing (2/3)

function *BestFitDecreasing*(n, w, c) (continua)

```
6   |   for k := 1 to n do
7   |   |   j := ρ[k]          // poi riempire: k-esimo
    |   |    oggetto più grande
8   |   |   s := −1            // contenitore selezionato
9   |   |   for b := 1 to nb do
10  |   |   |   if w[j] ≤ rc[b] and (s = −1 or rc[b] < rc[s])
    |   |   |    then
11  |   |   |   |   s ← b      // contenitore più pieno
    |   |   |   |    trovato finora
12  |   |   |   end
13  |   |   end
```

Algorithm: Best Fit Decreasing (3/3)

```
function BestFitDecreasing(n, w, c) (continua)
    for k := 1 to n do (continua)
14      if s = −1 then
15          nb ← nb + 1                  // aprire un nuovo
              contenitore
16          rc[nb] ← c
17          s ← nb
18      end
19      bin[j] ← s                       // assegnare l'oggetto
20      rc[s] ← rc[s] − w[j]
21  end
22  return bin, nb
23 end
```

Anche qui il ciclo di riempimento è quello di BF,

preceduto dall'ordinamento.

Esempio: BFD sull'istanza di riferimento del BPP

- L'ordinamento è $\rho = (3, 1, 2, 4, 6, 5)$, come per FFD.
- Gli oggetti 3, 1 e 2 aprono i contenitori 1, 2, 3, con capacità residue $2, 3, 3$.
- L'oggetto 4 (peso 3) entra nei contenitori 2 e 3, entrambi con capacità residua 3: il pareggio si risolve a favore del contenitore 2, e $rc[2] = 0$.
- L'oggetto 6 (peso 3) entra solo nel contenitore 3, e $rc[3] = 0$.
- L'oggetto 5 (peso 2) entra solo nel contenitore 1, e $rc[1] = 0$.

BFD restituisce gli stessi pattern di riempimento di FFD, $P_1 = \{3, 5\}$, $P_2 = \{1, 4\}$, $P_3 = \{2, 6\}$, e $BFD(I) = 3$.

Una garanzia migliore (1/2)

- Ordinare gli oggetti migliora il caso peggiore. FF e BF hanno rapporto di prestazione nel caso peggiore $\frac{17}{10}$.
- Per FFD è noto che

$$FFD(I) \leq \frac{11}{9} z(I) + \frac{6}{9} \qquad \text{per ogni istanza (del BPP) } I,$$

e che il bound è tight.

Una garanzia migliore (2/2)

- Un bound della forma $\frac{11}{9}z(I) + 4$ fu dimostrato per la prima volta sia per FFD sia per BFD da D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey e R.L. Graham (*Worst-case performance bounds for simple one-dimensional packing algorithms*, SIAM Journal on Computing 3 (1974) 299–325).

- La costante additiva tight è dovuta a G. Dósa (*The tight bound of First Fit Decreasing bin-packing algorithm is* $FFD(I) \leq 11/9\, OPT(I) + 6/9$, ESCAPE 2007, LNCS 4614, pp. 1–11).

Non una soluzione migliore su ogni istanza

- Una garanzia nel caso peggiore confronta l'algoritmo con l'ottimo, non con un altro algoritmo.
- Su alcune istanze l'ordine di input è più fortunato di quello ordinato.

Il prossimo esempio mostra un'istanza su cui
FF trova l'ottimo e FFD no.

Esempio: l'ordinamento può peggiorare (1/2)

Siano $c = 7$ e $w = (2, 2, 3, 2, 2, 3)$, con peso totale 14, così che

$$z(I) \geq \left\lceil \frac{14}{7} \right\rceil = 2.$$

FF mette gli oggetti $1, 2, 3$ nel contenitore 1 $(2 + 2 + 3 = 7)$ e gli oggetti $4, 5, 6$ nel contenitore 2 $(2 + 2 + 3 = 7)$: $FF(I) = 2$, che è ottimo.

FFD esamina gli oggetti nell'ordine $\rho = (3, 6, 1, 2, 4, 5)$:

- gli oggetti 3 e 6 vanno nel contenitore 1 ($rc[1] = 1$);
- gli oggetti 1, 2, 4 non entrano nel contenitore 1 e vanno nel contenitore 2 ($rc[2] = 1$);
- l'oggetto 5 non entra in nessuno dei due contenitori e apre il contenitore 3: $FFD(I) = 3$.

Esempio: l'ordinamento può peggiorare (2/2)

- I due oggetti di peso 3, messi insieme da FFD, sprecano un'unità di capacità nel contenitore 1.
- L'ordine di input li accoppiava invece ciascuno con due oggetti di peso 2, riempiendo esattamente entrambi i contenitori.

Riepilogo

Prima ordinare, poi riempire:

- offline, l'ordine degli oggetti si può scegliere: prima gli oggetti grandi, i piccoli riempiono i vuoti
- FFD e BFD: i cicli di FF e BF, preceduti dall'ordinamento
- sull'istanza di riferimento $FF(I) = 4$, $FFD(I) = BFD(I) = 3 = z(I)$

Che cosa dà l'ordinamento, e che cosa non dà:

- un caso peggiore migliore: $FFD(I) \leq \frac{11}{9} z(I) + \frac{6}{9}$, contro il rapporto $\frac{17}{10}$ di FF e BF
- non una soluzione migliore su ogni istanza: FFD può essere peggiore di FF

Scheduling su macchine parallele: prima i lavori più lunghi

Introduzione: Scheduling su macchine parallele: prima i lavori più lunghi

- In questa sezione applichiamo la stessa idea a $P \,||\, C_{\max}$: una regola sceglie la macchina, e l'ordine dei lavori è una seconda scelta
- Studiamo il list scheduling ($LIST$), che prende i lavori in un ordine qualunque dato, e LPT, che prende prima i lavori più lunghi
- Confrontiamo le loro garanzie nel caso peggiore, dovute a Graham, e vediamo che il bound di LPT è raggiunto dall'istanza di riferimento

La macchina meno carica

- Per $P \,||\, C_{\max}$ gli elementi sono gli assegnamenti dei lavori alle macchine, e ogni assegnamento è ammissibile.
- Una regola greedy naturale assegna ogni lavoro alla macchina che in quel momento è **la meno carica**, così che i carichi crescano nel modo più uniforme possibile.
- Come per il BPP, la regola sceglie la macchina, e l'ordine dei lavori è una seconda scelta, indipendente.

List scheduling

- Il *List Scheduling Algorithm* (indicato con *LIST*) è un'euristica costruttiva per $P \,||\, C_{\max}$.
- Esamina i lavori nell'ordine di una lista data ρ.
- Assegna ogni lavoro a una macchina con il carico corrente minimo (in caso di pareggio, la macchina con l'indice minimo).

Algorithm: List Scheduling (1/2)

Input : istanza di $P \,||\, C_{\max}$ con n lavori, tempi $t[1..n]$, m macchine; lista $\rho[1..n]$

Output: assegnamento lavori-macchine $mach[1..n]$ e makespan $C_{\max}$

```
1 function ListScheduling(n, m, t, ρ)
2 |   mach := nuovo vettore[1..n]        // assegnamento
  |    lavori-macchine
3 |   for h := 1 to m do
4 |   |   L[h] ← 0                      // carichi delle macchine
5 |   end
```

Algorithm: List Scheduling (2/2)

function *ListScheduling*(n, m, t, ρ) (continua)

```
6     for k := 1 to n do
7         j := ρ[k]      // k-esimo lavoro della lista
8         s := una macchina con L[s] minimo (indice
           minimo)              // macchina meno carica
9         mach[j] ← s  // assegnare il lavoro j alla
           macchina s
10        L[s] ← L[s] + t[j]
11    end
12    return mach, C_max = max_h L[h]
13 end
```

Una regola online

- *LIST* è il corrispondente, nello scheduling, di una regola *online*: non ha bisogno di conoscere i lavori futuri, e si usa, per esempio, quando i lavori arrivano uno alla volta.
- Con una scansione lineare delle macchine richiede tempo $O(nm)$, e tempo $O(n \log m)$ con un heap dei carichi.
- Nell'istanza di riferimento, $t = (5, 5, 4, 4, 3, 3, 3)$, i lavori sono elencati per tempo di esecuzione non crescente, che è un ordine favorevole.
- Per vedere l'effetto dell'ordine, l'esempio esamina gli stessi lavori nella lista inversa.

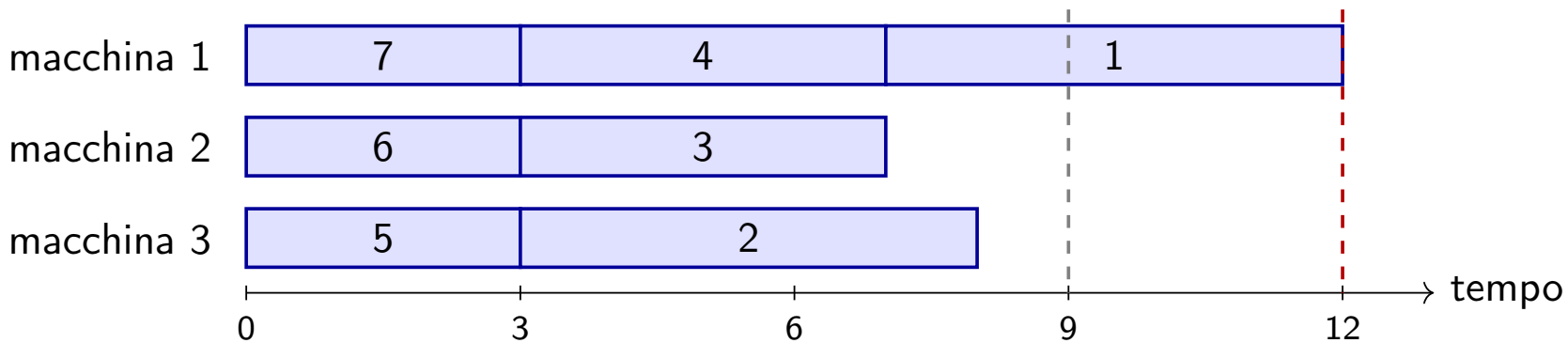
Esempio: *LIST* sull'istanza di riferimento, prima i lavori più brevi (1/2)

Consideriamo la lista $\rho = (7, 6, 5, 4, 3, 2, 1)$, cioè i lavori per tempo di esecuzione non decrescente. I carichi prima di ogni assegnamento e la macchina scelta sono:

lavoro j	t_j	carichi (L_1, L_2, L_3) prima	macchina
7	3	$(0, 0, 0)$	1
6	3	$(3, 0, 0)$	2
5	3	$(3, 3, 0)$	3
4	4	$(3, 3, 3)$	1
3	4	$(7, 3, 3)$	2
2	5	$(7, 7, 3)$	3
1	5	$(7, 7, 8)$	1

Esempio: *LIST* sull'istanza di riferimento, prima i lavori più brevi (2/2)

I carichi finali sono $(12, 7, 8)$, con $J_1 = \{7, 4, 1\}$, $J_2 = \{6, 3\}$, $J_3 = \{5, 2\}$, e $LIST(I) = 12$:



L'ultimo lavoro della lista è uno dei più lunghi: quando arriva, la macchina meno carica ha già carico 7, e il makespan salta a 12, tre unità sopra il lower bound 9 (linea grigia tratteggiata).

La garanzia di *LIST*

Quanto può essere lontano dall'ottimo *LIST*, qualunque sia l'ordine della lista?

Proposizione 1

Per ogni istanza di $P||C_{\max}$*, che indichiamo con* I*, e ogni lista* ρ*,*

$$LIST(I) \leq \left(2 - \frac{1}{m}\right) z(I).$$

Dimostrazione

Dimostrazione omessa. Il risultato è dovuto a

- R.L. Graham, *Bounds for certain multiprocessing anomalies*, Bell System Technical Journal 45 (1966) 1563–1581.

□

LIST è un algoritmo di approssimazione

LIST è un algoritmo di $\left(2 - \frac{1}{m}\right)$-approssimazione per $P \,||\, C_{\max}$, qualunque sia l'ordine della lista.

- Nell'esempio, $LIST(I) = 12 \leq \frac{5}{3} \cdot 9 = 15$.

Longest processing time first

- Il *Longest Processing Time first Algorithm* (indicato con LPT) è un'euristica costruttiva per $P \,||\, C_{\max}$.
- Ordina i lavori per tempo di esecuzione non crescente e applica $LIST$ a questa lista.
- Ogni lavoro, dal più lungo al più breve, si assegna a una macchina meno carica.

Algorithm: Longest Processing Time first

Input : istanza di $P \,||\, C_{\max}$ con n lavori, tempi $t[1..n]$, m macchine

Output: assegnamento lavori-macchine $mach[1..n]$ e makespan $C_{\max}$

```
1 function LPT(n, m, t)
2 |   ρ := i lavori ordinati per t[j] non crescente (pareggi
  |    per indice)        // prima i lavori più lunghi
3 |   return ListScheduling(n, m, t, ρ)
4 end
```

Perché prima i lavori più lunghi

- Il makespan è deciso dal lavoro che finisce per ultimo.
- Se i lavori in fondo alla lista sono brevi, gli ultimi assegnamenti possono aggiungere solo poco alla macchina meno carica, e fanno da "sabbia" che livella i carichi.
- Il parallelo con FFD è esatto: prima gli elementi grandi, poi quelli piccoli a riempire i vuoti.
- *LPT* richiede tempo $O(n \log n + n \log m)$.

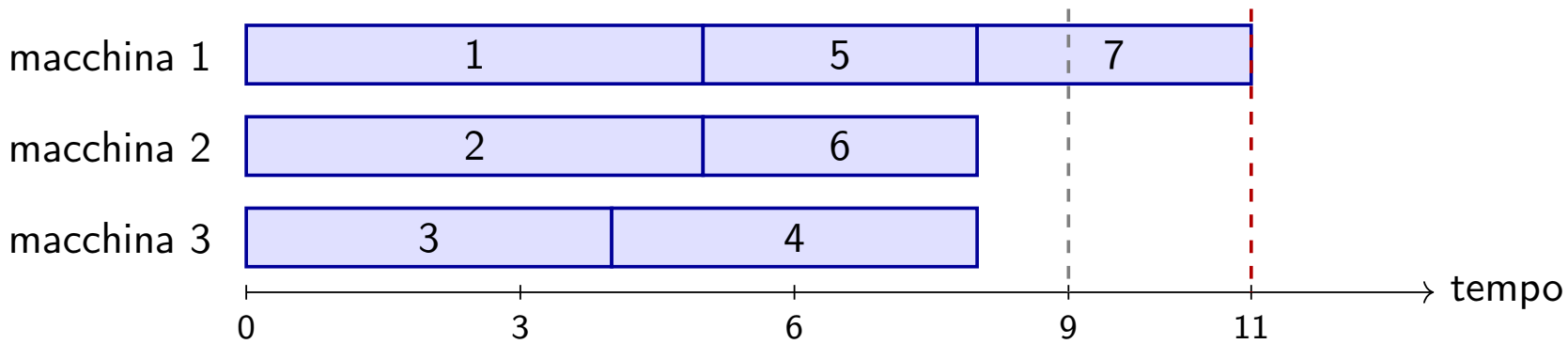
Esempio: *LPT* sull'istanza di riferimento (1/3)

I lavori sono già ordinati, $\rho = (1, 2, 3, 4, 5, 6, 7)$:

- i lavori 1, 2, 3 vanno sulle tre macchine vuote, con carichi $(5, 5, 4)$;
- il lavoro 4 va sulla macchina 3, la meno carica: $(5, 5, 8)$;
- il lavoro 5 va sulla macchina 1: $(8, 5, 8)$;
- il lavoro 6 sulla macchina 2: $(8, 8, 8)$;
- il lavoro 7 trova tre macchine con carico 8 e va sulla macchina 1: $(11, 8, 8)$.

Esempio: *LPT* sull'istanza di riferimento (2/3)

Quindi $J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$,
$mach = (1, 2, 3, 3, 1, 2, 1)$, e $LPT(I) = 11$:



Esempio: *LPT* sull'istanza di riferimento (3/3)

- Il lower bound elementare del capitolo «Algoritmi euristici: definizioni» dà

$$z(I) \geq \max\left\{\max_j t_j, \left\lceil \frac{\sum_j t_j}{m} \right\rceil\right\} = \max\{5, 9\} = 9,$$

  e $z(I) = 9$.

- L'ordinamento migliora il makespan da 12 a 11, ma *LPT* non è ottimo: dopo sei lavori i carichi sono perfettamente bilanciati, $(8, 8, 8)$, e il settimo lavoro li sbilancia di 3.

La garanzia di *LPT*

Con prima i lavori più lunghi, la garanzia diventa:

Proposizione 2

Per ogni istanza di $P||C_{\max}$*, che indichiamo con* I*,*

$$LPT(I) \leq \left(\frac{4}{3} - \frac{1}{3m}\right) z(I).$$

Dimostrazione

Dimostrazione omessa. Il risultato è dovuto a

- R.L. Graham, *Bounds on multiprocessing timing anomalies*, SIAM Journal on Applied Mathematics 17 (1969) 416–429.

□

LPT è un algoritmo di approssimazione

LPT è un algoritmo di $\left(\frac{4}{3} - \frac{1}{3m}\right)$-approssimazione per $P \,||\, C_{\max}$:

ordinare la lista abbassa la garanzia di *LIST*
da $2 - \frac{1}{m}$ a $\frac{4}{3} - \frac{1}{3m}$.

- Per $m = 3$ il bound è $\frac{4}{3} - \frac{1}{9} = \frac{11}{9}$, e sull'istanza di riferimento

$$\frac{LPT(I)}{z(I)} = \frac{11}{9}.$$

Il bound di *LPT* è tight

- L'istanza di riferimento raggiunge esattamente il bound di Graham.
- Appartiene alla famiglia di istanze, data da Graham, con $2m+1$ lavori di tempi di esecuzione

  $$2m-1,\ 2m-1,\ 2m-2,\ 2m-2,\ \ldots,\ m+1,\ m+1,\ m,\ m,\ m,$$

  su cui *LPT* dà $4m-1$ e l'ottimo è $3m$.
- Per $m=3$ questi sono i tempi $(5,5,4,4,3,3,3)$.
- Il bound della proposizione precedente non si può quindi migliorare.

Riepilogo

List scheduling:

- ogni lavoro della lista va su una macchina meno carica; una regola online, tempo $O(nm)$
- $LIST(I) \leq \left(2 - \frac{1}{m}\right) z(I)$ per ogni lista; sulla lista inversa $LIST(I) = 12$

Longest processing time first:

- $LIST$ sui lavori ordinati per tempo di esecuzione non crescente
- $LPT(I) \leq \left(\frac{4}{3} - \frac{1}{3m}\right) z(I)$; sull'istanza di riferimento $LPT(I) = 11$
- il bound è tight: $\frac{11}{9}$ per $m = 3$, raggiunto dall'istanza di riferimento

Riepilogo della prima parte (1/2)

Euristiche costruttive e paradigma greedy:

- una soluzione ammissibile, costruita un elemento alla volta a partire dalla soluzione parziale vuota
- un algoritmo greedy aggiunge l'elemento che sembra migliore adesso, e non rivede mai una scelta
- l'ordine indotto dal criterio di selezione decide tutto

Il problema dello zaino 0/1:

- tre ordinamenti: profitto, peso, efficienza; ciascuno può essere arbitrariamente cattivo
- combinare le soluzioni protegge: Ext-Greedy garantisce $\frac{z(I)}{GR^e(I)} \leq 2$

Riepilogo della prima parte (2/2)

Il problema del bin packing: prima ordinare, poi riempire:

- FFD e BFD: prima gli oggetti grandi; $3 = z(I)$ contenitori sull'istanza di riferimento, contro i 4 di FF
- $FFD(I) \leq \frac{11}{9} z(I) + \frac{6}{9}$, ma su alcune istanze l'ordinamento può peggiorare

Scheduling su macchine parallele: prima i lavori più lunghi:

- $LIST$: ogni lavoro su una macchina meno carica, $LIST(I) \leq \left(2 - \frac{1}{m}\right) z(I)$
- LPT: prima i lavori più lunghi, $LPT(I) \leq \left(\frac{4}{3} - \frac{1}{3m}\right) z(I)$, un bound tight

Nella seconda parte: il TSP, il VCP, la costruzione randomizzata, il regret e il look-ahead.