

Ricerca locale (2/2)

Indice (1/2)

$P \parallel C_{\max}$: move e swap

La ricerca locale move

La ricerca locale swap

Paesaggi piatti e obiettivi ausiliari

Plateau

L'obiettivo ausiliario

La ricerca locale move per il BPP

La ricerca locale swap per il BPP

Proprietà dell'obiettivo ausiliario

Esempi

Esperimenti su due istanze medie

Indice (2/2)

- KP: ammissibilità e penalità

 - Restare ammissibili

 - Penalizzare il peso in eccesso

- Variable neighbourhood descent

- Ricerca locale multi-start

 - I limiti del multi-start

 - Che cosa dà la ricerca locale, e che cosa non dà

- Implementazione in C++

- Errori tipici

- In sintesi

Dove eravamo

- ▶ Una ricerca locale parte da una soluzione ammissibile e si sposta su un vicino migliore finché ne esiste uno: si ferma in un ottimo locale rispetto al suo intorno N .
- ▶ Gli intorni: flip e swap per il KP, move e swap per il BPP e $P \parallel C_{\max}$, 2-opt e Or-opt per il TSP.
- ▶ First improvement e best improvement possono fermarsi in ottimi locali diversi.
- ▶ La valutazione incrementale calcola la variazione Δ di una mossa in $O(1)$ per 2-opt e Or-opt.
- ▶ **Ora:** $P \parallel C_{\max}$, plateau e obiettivi ausiliari, penalità per il KP, variable neighbourhood descent e multi-start.

$P \parallel C_{\max}$: move e swap

Introduzione: $P \parallel C_{\max}$: move e swap

- ▶ In questa sezione scriviamo le ricerche locali move e swap per $P \parallel C_{\max}$
- ▶ Entrambe le mosse trasferiscono tempo di lavorazione fra due macchine: la variazione del makespan si calcola dai carichi
- ▶ Sull'istanza di riferimento la soluzione di *LPT* è un ottimo locale per move, ma non per swap

Trasferire tempo di lavorazione

- ▶ Le mosse di $P \parallel C_{\max}$ trasferiscono tempo di lavorazione fra due macchine:
 - ▶ una move del lavoro j dalla macchina a alla macchina b trasferisce $\delta = t_j$ da a a b ;
 - ▶ uno swap dei lavori j (su a) e k (su b) trasferisce $\delta = t_j - t_k$, che può essere negativo.
- ▶ In entrambi i casi i nuovi carichi sono

$$L_a - \delta \quad \text{e} \quad L_b + \delta,$$

e $F(L, a, b, \delta)$ indica la variazione risultante dell'obiettivo.

La ricerca locale move

- ▶ La ricerca locale *move* per $P \parallel C_{\max}$ esamina le riassegnazioni di un lavoro a un'altra macchina.
- ▶ Ne applica una migliorante scelta con la regola first improvement o best improvement, e aggiorna i due carichi coinvolti.
- ▶ Si ferma quando nessuna riassegnazione diminuisce l'obiettivo.

Algorithm: Passo move per $P \parallel C_{\max}$ (1/3)

Input : n lavori, m macchine, tempi $t[1..n]$, assegnazione $mach[1..n]$, carichi $L[1..m]$, regola

Output : *true* se è stata applicata una mossa migliorante,
false altrimenti

```
1 function PCmaxMoveStep( $n, m, t, mach, L, rule$ )
2    $\bar{\Delta} := 0; (\bar{j}, \bar{h}) := (0, 0)$       // miglior mossa trovata
   finora
3   for  $j := 1$  to  $n$  do
4      $a := mach[j]$                       // macchina del lavoro  $j$ 
5     for  $h := 1$  to  $m$  do
6       if  $h = a$  then
7         continue
8     end
```

Algorithm: Passo move per $P \parallel C_{\max}$ (2/3)

function *PCmaxMoveStep*($n, m, t, mach, L, rule$) (continua)

```
  for  $j := 1$  to  $n$  (continua) do
    for  $h := 1$  to  $m$  (continua) do
      9       $\Delta := F(L, a, h, t[j])$                 // variazione
          dell'obiettivo
      10      if  $\Delta < \bar{\Delta}$  then
      11           $\bar{\Delta} \leftarrow \Delta; (\bar{j}, \bar{h}) \leftarrow (j, h)$     // mossa
          migliorante
      12          if  $rule = first$  then
      13              | uscire da entrambi i cicli
      14          end
      15      end
      16  end
      17  end
```

Algorithm: Passo move per $P \parallel C_{\max}$ (3/3)

function *PCmaxMoveStep*($n, m, t, mach, L, rule$) (continua)

```
18   if  $\bar{j} = 0$  then return false           // nessuna mossa
      migliorante: ottimo locale
19    $L[mach[\bar{j}]] \leftarrow L[mach[\bar{j}]] - t[\bar{j}]; L[\bar{h}] \leftarrow L[\bar{h}] + t[\bar{j}]$ 
      // aggiornare i carichi
20    $mach[\bar{j}] \leftarrow \bar{h}$                    // applicare la mossa
21   return true
22 end
```

La ricerca locale swap

- ▶ La ricerca locale *swap* per $P \parallel C_{\max}$ esamina gli scambi di due lavori su macchine diverse.
- ▶ Ne applica uno migliorante scelto con la regola first improvement o best improvement, e aggiorna i due carichi coinvolti.
- ▶ Si ferma quando nessuno scambio diminuisce l'obiettivo.

Algorithm: Passo swap per $P \parallel C_{\max}$ (1/3)

Input : n lavori, m macchine, tempi $t[1..n]$, assegnazione $mach[1..n]$, carichi $L[1..m]$, regola

Output : *true* se è stato applicato uno swap migliorante, *false* altrimenti

```
1 function PCmaxSwapStep( $n, m, t, mach, L, rule$ )
2    $\bar{\Delta} := 0$ ;  $(\bar{j}, \bar{k}) := (0, 0)$       // miglior swap trovato
   finora
3   for  $j := 1$  to  $n$  do
4     for  $k := j + 1$  to  $n$  do
5        $a := mach[j]$ ;  $b := mach[k]$ 
6       if  $a = b$  or  $t[j] = t[k]$  then continue      // i
           carichi non cambierebbero
```

Algorithm: Passo swap per $P \parallel C_{\max}$ (2/3)

function *PCmaxSwapStep*($n, m, t, mach, L, rule$) (continua)

```
  for  $j := 1$  to  $n$  (continua) do
    for  $k := j + 1$  to  $n$  (continua) do
7       $\Delta := F(L, a, b, t[j] - t[k])$            // variazione
        dell'obiettivo
8      if  $\Delta < \bar{\Delta}$  then
9         $\bar{\Delta} \leftarrow \Delta; (\bar{j}, \bar{k}) \leftarrow (j, k)$            // swap
        migliorante
10     if  $rule = first$  then
11       | uscire da entrambi i cicli
12     end
13   end
14 end
15 end
```

Algorithm: Passo swap per $P \parallel C_{\max}$ (3/3)

function *PCmaxSwapStep*($n, m, t, mach, L, rule$) (continua)

```
16  |  if  $\bar{j} = 0$  then return false           // nessuno swap
    |  migliorante: ottimo locale
17  |   $a := mach[\bar{j}]; b := mach[\bar{k}]; \delta := t[\bar{j}] - t[\bar{k}]$ 
18  |   $L[a] \leftarrow L[a] - \delta; L[b] \leftarrow L[b] + \delta$       // aggiornare i
    |  carichi
19  |   $mach[\bar{j}] \leftarrow b; mach[\bar{k}] \leftarrow a$       // applicare lo swap
20  |  return true
21 end
```

La variazione del makespan

Per il makespan, la variazione dell'obiettivo è

$$F(L, a, b, \delta) = \max \left\{ L_a - \delta, L_b + \delta, \max_{q \neq a, b} L_q \right\} - \max_q L_q,$$

calcolata in tempo $O(m)$ dai carichi $L_1, \dots, L_m$ della soluzione corrente.

Mosse miglioranti e macchine critiche

- ▶ Una move o uno swap è migliorante se e solo se $F(L, a, b, \delta) < 0$.
- ▶ Poiché cambiano solo L_a e L_b , il makespan può diminuire solo se a è l'**unica** macchina critica.
- ▶ Se un'altra macchina ha carico $C_{\max}$, nessuna mossa fra due macchine può abbassare il makespan.

Costo di un'iterazione, e move contro swap

- ▶ Un'iterazione costa tempo $O(nm^2)$ per move e $O(n^2m)$ per swap, a causa della valutazione in $O(m)$ di F ; tenere i carichi ordinati lo riduce, ma qui basta la versione semplice.
- ▶ Una move cambia il numero di lavori su due macchine, uno swap mantiene il numero di lavori su ogni macchina.
- ▶ Se una macchina ha troppi lavori, solo una move può correggerlo: i due intorni si completano a vicenda.

Esempio: la soluzione di LPT (1/6)

- ▶ La soluzione di LPT sull'istanza di riferimento di $P \parallel C_{\max}$ ha

$$J_1 = \{1, 5, 7\}, \quad J_2 = \{2, 6\}, \quad J_3 = \{3, 4\},$$

carichi $(11, 8, 8)$ e $C_{\max} = 11$.

- ▶ La macchina 1 è l'unica macchina critica.
- ▶ Mostriamo che è un ottimo locale per move, ma non per swap.

Esempio: la soluzione di *LPT* (2/6)

Move. Le 14 mosse danno i carichi

mossa	carichi	$C_{\max}$	mossa	carichi	$C_{\max}$
lavoro 1: $M_1 \rightarrow M_2$	(6, 13, 8)	13	lavoro 4: $M_3 \rightarrow M_2$	(11, 12, 4)	12
lavoro 1: $M_1 \rightarrow M_3$	(6, 8, 13)	13	lavoro 5: $M_1 \rightarrow M_2$	(8, 11, 8)	11
lavoro 2: $M_2 \rightarrow M_1$	(16, 3, 8)	16	lavoro 5: $M_1 \rightarrow M_3$	(8, 8, 11)	11
lavoro 2: $M_2 \rightarrow M_3$	(11, 3, 13)	13	lavoro 6: $M_2 \rightarrow M_1$	(14, 5, 8)	14
lavoro 3: $M_3 \rightarrow M_1$	(15, 8, 4)	15	lavoro 6: $M_2 \rightarrow M_3$	(11, 5, 11)	11
lavoro 3: $M_3 \rightarrow M_2$	(11, 12, 4)	12	lavoro 7: $M_1 \rightarrow M_2$	(8, 11, 8)	11
lavoro 4: $M_3 \rightarrow M_1$	(15, 8, 4)	15	lavoro 7: $M_1 \rightarrow M_3$	(8, 8, 11)	11

Esempio: la soluzione di *LPT* (3/6)

- ▶ Nessuna mossa diminuisce il makespan:
 - ▶ spostare il lavoro 1 via dalla macchina 1 porta la macchina ricevente a $8 + 5 = 13$;
 - ▶ spostare il lavoro 5 o 7 abbassa la macchina 1 a 8 ma alza la macchina ricevente a $8 + 3 = 11$.
- ▶ La soluzione di *LPT* è un **ottimo locale per move**, con $C_{\max} = 11$.

Esempio: la soluzione di LPT (4/6)

Swap.

- ▶ Il primo swap esaminato scambia il lavoro 1 (su M_1 , $t_1 = 5$) e il lavoro 3 (su M_3 , $t_3 = 4$): $\delta = 1$, i carichi diventano (10, 8, 9) e $C_{\max} = 10$.
- ▶ È migliorante, e first improvement lo accetta subito.
- ▶ Best improvement esamina tutti i 13 swap e ne trova altri due con $C_{\max} = 10$ (lavori 1 e 4, lavori 1 e 6), quindi accetta lo stesso swap, il primo trovato.
- ▶ Ora $J_1 = \{3, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{1, 4\}$.

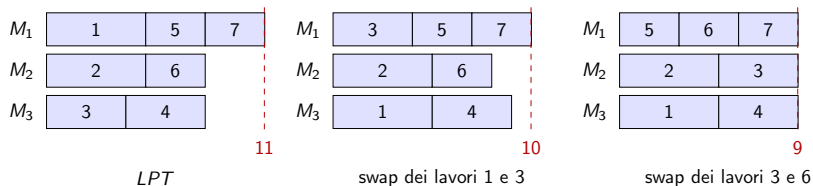
Esempio: la soluzione di LPT (5/6)

- ▶ Nella seconda iterazione, lo swap del lavoro 3 (su M_1 , $t_3 = 4$) e del lavoro 6 (su M_2 , $t_6 = 3$) ha $\delta = 1$ e dà i carichi $(9, 9, 9)$.
- ▶ Si trova dopo 9 valutazioni, ed è l'unico swap con $C_{\max} < 10$, quindi entrambe le regole lo accettano.
- ▶ La nuova soluzione

$$J_1 = \{5, 6, 7\}, \quad J_2 = \{2, 3\}, \quad J_3 = \{1, 4\}$$

ha $C_{\max} = 9 = \lceil 27/3 \rceil$: è **ottima**.

Esempio: la soluzione di *LPT* (6/6)



- Da *LPT* ($C_{\max} = 11$) due swap raggiungono l'ottimo $C_{\max} = 9$.

L'ottimo locale dipende dall'intorno

- ▶ In questo esempio l'intorno move si ferma a 11, e l'intorno swap raggiunge l'ottimo 9.
- ▶ Come osservato per il KP nel capitolo «Algoritmi euristici: definizioni», **l'ottimo locale dipende dall'intorno.**

Un intorno più grande non sempre aiuta

- Consideriamo l'istanza con $m = 4$ macchine e 12 lavori con tempi

$$t = (11, 10, 9, 8, 7, 7, 6, 5, 4, 4, 3, 2),$$

il cui makespan ottimo è $z(I) = 19 = \lceil 76/4 \rceil$.

- *LPT* dà

$$J_1 = \{1, 8, 10\}, \quad J_2 = \{2, 7, 11\},$$

$$J_3 = \{3, 6, 12\}, \quad J_4 = \{4, 5, 9\},$$

con carichi $(20, 19, 18, 19)$.

- Questa soluzione è un ottimo locale sia per move sia per swap.

Move e swap per $P \parallel C_{\max}$:

- ▶ Entrambe trasferiscono una quantità δ fra due macchine;
 $F(L, a, b, \delta)$ si calcola in $O(m)$
- ▶ Il makespan può diminuire solo se a è l'unica macchina critica
- ▶ Una move cambia il numero di lavori sulle macchine, uno swap no

Sull'istanza di riferimento:

- ▶ LPT (11) è un ottimo locale per move; due swap raggiungono l'ottimo 9
- ▶ L'ottimo locale dipende dall'intorno

Paesaggi piatti e obiettivi ausiliari

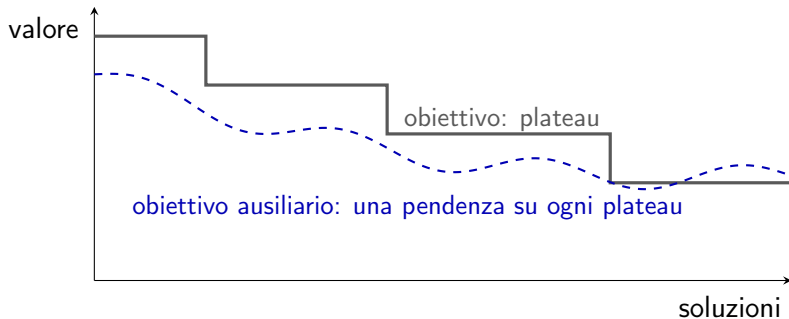
Introduzione: Paesaggi piatti e obiettivi ausiliari

- ▶ Gli obiettivi del BPP e di $P \parallel C_{\max}$ assumono pochi valori: la maggior parte dei vicini ha lo stesso valore
- ▶ In questa sezione vediamo perché una ricerca locale si ferma su questi plateau, e come un obiettivo ausiliario le dà una direzione
- ▶ Scriviamo le ricerche locali move e swap per il BPP, e le proviamo su istanze piccole e medie

Plateau

- ▶ Gli obiettivi del BPP e di $P \parallel C_{\max}$ sono molto “grossolani”: il numero di contenitori assume pochi valori, e il makespan dipende solo dalle macchine critiche.
- ▶ Molte soluzioni hanno lo stesso valore, e la maggior parte delle mosse non lo cambia: il paesaggio è fatto di ampi **plateau**.
- ▶ Su un plateau una ricerca locale che accetta solo mosse strettamente miglioranti si ferma, anche se una sequenza di mosse di valore zero porterebbe a una soluzione migliore.
- ▶ La ricerca è **cieca**: l'obiettivo non dice quali vicini sono più vicini a un miglioramento.

I plateau in una figura



Quanto piatto?

- ▶ Sull'istanza di $P \parallel C_{\max}$ con $m = 4$ e 12 lavori della sezione precedente, generiamo 100 assegnazioni casuali (seme 1) ed esaminiamo i loro $100 \cdot 12 \cdot 3 = 3600$ vicini move: 1290 di essi (35.8%) hanno lo stesso makespan della soluzione corrente.
- ▶ Per il BPP la situazione è estrema: sull'istanza con $c = 10$ e pesi

$$w = (7, 6, 6, 5, 5, 4, 4, 4, 3, 3, 2, 1),$$

per cui $z(I) = 5 = \lceil 50/10 \rceil$, le 100 soluzioni calcolate da *FF* su ordini casuali degli oggetti (seme 1) hanno 2374 vicini move e swap, e **tutti** hanno lo stesso numero di contenitori della soluzione corrente.

L'obiettivo ausiliario

- ▶ Il rimedio è guidare la ricerca con un **obiettivo ausiliario**, che distingue le soluzioni con lo stesso valore e premia quelle “più vicine” a un miglioramento.
- ▶ Per entrambi i problemi è una funzione dei carichi:

$$\text{BPP: } \max_b \sum L_b^2 = \sum_b \left(\sum_{j \in P_b} w_j \right)^2,$$

$$P \parallel C_{\max} : \min \sum_{h=1}^m L_h^2.$$

Carichi bilanciati e sbilanciati

- ▶ Entrambi i problemi ridistribuiscono un totale *fissato*, $\sum_j w_j$ o $\sum_j t_j$, fra i contenitori o le macchine.
- ▶ A totale fissato, la somma dei quadrati è piccola quando i carichi sono *bilanciati* e grande quando sono *sbilanciati*.
- ▶ Per $P || C_{\max}$ i carichi bilanciati sono esattamente ciò che vogliamo; per il BPP vogliamo il contrario: pochi contenitori pieni e gli altri vuoti.

Proposizione 1

Se la quantità δ è trasferita dal carico L_a al carico L_b ($a \neq b$), la somma dei carichi al quadrato varia di

$$(L_a - \delta)^2 + (L_b + \delta)^2 - L_a^2 - L_b^2 = 2\delta(L_b - L_a + \delta).$$

Dimostrazione

Sviluppando i quadrati,

$$(L_a - \delta)^2 - L_a^2 = -2\delta L_a + \delta^2,$$

$$(L_b + \delta)^2 - L_b^2 = 2\delta L_b + \delta^2;$$

la somma è

$$-2\delta L_a + \delta^2 + 2\delta L_b + \delta^2 = 2\delta(L_b - L_a + \delta).$$

Gli altri carichi non cambiano.



L'obiettivo ausiliario costa poco da valutare

- ▶ La variazione si calcola in tempo $O(1)$: l'obiettivo ausiliario è anche meno costoso da valutare del makespan.
- ▶ Per una move, $\delta > 0$ è il peso (o il tempo) dell'elemento spostato, e la somma dei quadrati **aumenta** se e solo se

$$L_b + \delta > L_a :$$

l'elemento va in un carico che, dopo la mossa, è più grande del carico che ha lasciato.

La ricerca locale *move* per il BPP

- ▶ La ricerca locale *move* per il BPP esamina gli spostamenti di un oggetto in un altro contenitore non vuoto in cui entra.
- ▶ Ne applica uno migliorante scelto con la regola first improvement o best improvement, e si ferma quando nessuna mossa migliora.
- ▶ L'obiettivo è o il numero di contenitori, $f = nb$, o l'obiettivo ausiliario

$$f = SQ = \sum_b L_b^2;$$

il **guadagno** G di una mossa è il miglioramento di f .

Algorithm: Passo move per il BPP (1/4)

Input : n oggetti, pesi $w[1..n]$, capacità c , contenitori
 $bin[1..n]$, carichi L , $f \in \{nb, SQ\}$, regola

Output : *true* se è stata applicata una mossa migliorante,
false altrimenti

```
1 function BPPMoveStep( $n, w, c, bin, L, f, rule$ )
2    $\bar{G} := 0$ ;  $(\bar{j}, \bar{b}) := (0, 0)$     // miglior mossa trovata
   finora
3   for  $j := 1$  to  $n$  do
4      $a := bin[j]$                 // contenitore dell'oggetto  $j$ 
5     for ogni contenitore  $b$ , in ordine crescente do
6       if  $b = a$  or  $L[b] = 0$  or  $L[b] + w[j] > c$  then
         continue // stesso, vuoto o troppo pieno
```

Algorithm: Passo move per il BPP (2/4)

function *BPPMoveStep*($n, w, c, bin, L, f, rule$) (continua)

```
    for  $j := 1$  to  $n$  (continua) do
        for ogni contenitore  $b$ , in ordine crescente (continua)
            do
                if  $f = SQ$  then
                    7       $G := 2 w[j] (L[b] - L[a] + w[j])$ 
                    8      // Proposizione 1
                else
                    9       $G := 1$  if  $L[a] = w[j]$  else 0
                    10     // il
                       contenitore  $a$  si svuota
                end
11
```

Algorithm: Passo move per il BPP (3/4)

function *BPPMoveStep*($n, w, c, bin, L, f, rule$) (continua)

```
12   for  $j := 1$  to  $n$  (continua) do
13       for ogni contenitore  $b$ , in ordine crescente (continua)
14           do
15               if  $G > \bar{G}$  then
16                    $\bar{G} \leftarrow G; (\bar{j}, \bar{b}) \leftarrow (j, b)$            // mossa
17                   if  $rule = first$  then
18                       uscire da entrambi i cicli
19                   end
16           end
17       end
18   end
```

Algorithm: Passo move per il BPP (4/4)

function *BPPMoveStep*($n, w, c, bin, L, f, rule$) (continua)

```
20   if  $\bar{j} = 0$  then return false           // nessuna mossa
      migliorante: ottimo locale
21    $L[bin[\bar{j}]] \leftarrow L[bin[\bar{j}]] - w[\bar{j}]; L[\bar{b}] \leftarrow L[\bar{b}] + w[\bar{j}]$ 
      // aggiornare i carichi
22    $bin[\bar{j}] \leftarrow \bar{b}$                      // applicare la mossa
23   return true
24 end
```

La ricerca locale *swap* per il BPP

- ▶ La ricerca locale *swap* per il BPP esamina gli scambi di due oggetti di peso diverso in contenitori diversi che mantengono ammissibili entrambi i contenitori.
- ▶ Ne applica uno migliorante scelto con la regola first improvement o best improvement, e si ferma quando nessuno scambio migliora.
- ▶ Uno swap non cambia mai il numero di contenitori, quindi si usa solo con l'obiettivo ausiliario *SQ*.

Algorithm: Passo swap per il BPP (1/3)

Input : n oggetti, pesi $w[1..n]$, capacità c , contenitori
 $bin[1..n]$, carichi L , regola

Output : *true* se è stato applicato uno swap migliorante, *false*
altrimenti

```
1 function BPPSwapStep( $n, w, c, bin, L, rule$ )
2    $\bar{G} := 0; (\bar{j}, \bar{k}) := (0, 0)$       // miglior swap trovato
   finora
3   for  $j := 1$  to  $n$  do
4     for  $k := j + 1$  to  $n$  do
5        $a := bin[j]; b := bin[k]$ 
6       if  $a = b$  or  $w[j] = w[k]$  then continue      // i
       carichi non cambierebbero
7        $\delta := w[j] - w[k]$   // peso trasferito da  $a$  a
        $b$ 
```

Algorithm: Passo swap per il BPP (2/3)

function *BPPSwapStep*($n, w, c, bin, L, rule$) (continua)

```
1  for  $j := 1$  to  $n$  (continua) do
2      for  $k := j + 1$  to  $n$  (continua) do
3          if  $L[b] + \delta > c$  or  $L[a] - \delta > c$  then continue
4              // un contenitore sarebbe troppo pieno
5           $G := 2\delta(L[b] - L[a] + \delta)$     // Proposizione 1
6          if  $G > \bar{G}$  then
7               $\bar{G} \leftarrow G; (\bar{j}, \bar{k}) \leftarrow (j, k)$            // swap
8              migliorante
9              if  $rule = first$  then
10                 | uscire da entrambi i cicli
11             end
12         end
13     end
14 end
```

Algorithm: Passo swap per il BPP (3/3)

function *BPPSwapStep*($n, w, c, bin, L, rule$) (continua)

```
18   if  $\bar{j} = 0$  then return false           // nessuno swap
    migliorante: ottimo locale
19    $a := bin[\bar{j}]; b := bin[\bar{k}]; \delta := w[\bar{j}] - w[\bar{k}]$ 
20    $L[a] \leftarrow L[a] - \delta; L[b] \leftarrow L[b] + \delta$  // aggiornare i
    carichi
21    $bin[\bar{j}] \leftarrow b; bin[\bar{k}] \leftarrow a$        // applicare lo swap
22   return true
23 end
```

Che relazione c'è fra gli obiettivi ausiliari
e gli obiettivi originali?

Due proposizioni, una per ogni problema.

Proposizione 2

Per il BPP, con gli intorni move e swap:

- 1. una move che svuota un contenitore aumenta $\sum_b L_b^2$;*
- 2. uno swap non cambia il numero di contenitori.*

Quindi ogni move o swap che diminuisce il numero di contenitori aumenta anche $\sum_b L_b^2$.

Dimostrazione

- ▶ Se la move dell'oggetto j dal contenitore a al contenitore b svuota il contenitore a , allora $L_a = w_j$, e il contenitore b non è vuoto, quindi $L_b > 0$.
- ▶ Per la Proposizione 1 con $\delta = w_j$ la variazione è

$$2w_j(L_b - w_j + w_j) = 2w_jL_b > 0.$$

- ▶ Uno swap scambia un oggetto del contenitore a con un oggetto del contenitore b , quindi entrambi i contenitori restano non vuoti e non si crea nessun contenitore.



Proposizione 3

Per $P \parallel C_{\max}$, con gli intorni move e swap:

- 1. una move o uno swap che diminuisce $\sum_h L_h^2$ non aumenta il makespan;*
- 2. una move o uno swap che diminuisce il makespan diminuisce $\sum_h L_h^2$.*

Dimostrazione (1/2)

- ▶ Sia δ trasferito dalla macchina a alla macchina b ; scambiando se necessario i nomi di a e b , $\delta > 0$. Cambiano solo L_a e L_b .
- ▶ (1) Per la Proposizione 1, $\sum_h L_h^2$ diminuisce se e solo se

$$L_b + \delta < L_a;$$

allora entrambi i nuovi carichi $L_a - \delta$ e $L_b + \delta$ sono minori di $L_a \leq C_{\max}$, e il makespan non aumenta.

Dimostrazione (2/2)

- ▶ (2) Se il makespan diminuisce, la macchina b , il cui carico aumenta, non è critica dopo la mossa, quindi

$$L_b + \delta < C_{\max}.$$

- ▶ La macchina a doveva essere l'unica macchina critica, $L_a = C_{\max}$.
- ▶ Quindi $L_b + \delta < L_a$, e $\sum_h L_h^2$ diminuisce.



Che cosa dicono le due proposizioni

- ▶ Per l'obiettivo ausiliario, ogni mossa che migliora l'obiettivo originale è migliorante, e lo sono anche alcune mosse che l'obiettivo originale vede come neutre.
- ▶ Per $P \parallel C_{\max}$, nessuna mossa che peggiora il makespan è migliorante.
- ▶ Per il BPP una move che aumenta $\sum_b L_b^2$ non aumenta mai il numero di contenitori, perché non si crea nessun contenitore.

Una guida, non l'obiettivo

- ▶ L'obiettivo ausiliario è una guida, non l'obiettivo: una soluzione con più contenitori può avere un $\sum_b L_b^2$ più grande.
- ▶ Con $c = 7$:
 - ▶ i carichi (5, 5) usano 2 contenitori e danno $25 + 25 = 50$;
 - ▶ i carichi (7, 2, 1) usano 3 contenitori e danno $49 + 4 + 1 = 54$.
- ▶ La qualità della soluzione restituita si misura sempre con l'obiettivo originale.

Esempio: il numero di contenitori è cieco (1/5)

- ▶ La soluzione di FF sull'istanza di riferimento del BPP ($c = 7$) ha gli schemi di carico

$$P_1 = \{1, 4\}, \quad P_2 = \{2, 5\}, \quad P_3 = \{3\}, \quad P_4 = \{6\},$$

con carichi $(7, 6, 5, 3)$ e $nb = 4$.

- ▶ Il suo obiettivo ausiliario è

$$SQ = 49 + 36 + 25 + 9 = 119.$$

- ▶ Mostriamo che il numero di contenitori è cieco, e l'obiettivo ausiliario no.

Esempio: il numero di contenitori è cieco (2/5)

Le mosse ammissibili della soluzione di *FF*:

mossa	carichi	nb	guadagno $2w_j(L_b - L_a + w_j)$	SQ
oggetto 1 ($w_1 = 4$): contenitore 1 $\rightarrow$ 4	(3, 6, 5, 7)	4	$2 \cdot 4 \cdot (3 - 7 + 4) = 0$	119
oggetto 2 ($w_2 = 4$): contenitore 2 $\rightarrow$ 4	(7, 2, 5, 7)	4	$2 \cdot 4 \cdot (3 - 6 + 4) = 8$	127
oggetto 4 ($w_4 = 3$): contenitore 1 $\rightarrow$ 4	(4, 6, 5, 6)	4	$2 \cdot 3 \cdot (3 - 7 + 3) = -6$	113
oggetto 5 ($w_5 = 2$): contenitore 2 $\rightarrow$ 3	(7, 4, 7, 3)	4	$2 \cdot 2 \cdot (5 - 6 + 2) = 4$	123
oggetto 5 ($w_5 = 2$): contenitore 2 $\rightarrow$ 4	(7, 4, 5, 5)	4	$2 \cdot 2 \cdot (3 - 6 + 2) = -4$	115

Esempio: il numero di contenitori è cieco (3/5)

Con $f = nb$:

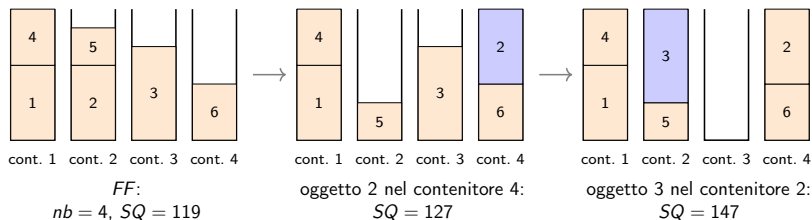
- ▶ Nessuna mossa svuota un contenitore: gli oggetti 3 e 6, soli nei loro contenitori, non entrano da nessun'altra parte.
- ▶ La soluzione di FF è un ottimo locale con 4 contenitori, e lo è anche per l'intorno swap, per la Proposizione 2.

Esempio: il numero di contenitori è cieco (4/5)

Con $f = SQ$:

- ▶ Entrambe le regole accettano la mossa dell'oggetto 2 nel contenitore 4 (guadagno 8, $SQ = 127$), che lascia l'oggetto 5 da solo nel contenitore 2.
- ▶ Nell'iterazione successiva, due mosse hanno il guadagno massimo, 20: l'oggetto 3 nel contenitore 2, con guadagno $2 \cdot 5 \cdot (2 - 5 + 5) = 20$, e l'oggetto 5 nel contenitore 3, con guadagno $2 \cdot 2 \cdot (5 - 2 + 2) = 20$.
- ▶ L'oggetto 3 è esaminato per primo, quindi entrambe le regole lo scelgono. Svuota il contenitore 3: i carichi diventano $(7, 7, 7)$, $SQ = 147$, e gli schemi di carico $\{1, 4\}$, $\{3, 5\}$, $\{2, 6\}$ usano 3 contenitori, che è **ottimo**.
- ▶ La prima mossa non ha cambiato il numero di contenitori: ha preparato la seconda.

Esempio: il numero di contenitori è cieco (5/5)



- Due mosse con $f = SQ$: la prima non cambia nb , la seconda svuota il contenitore 3.

Esempio: partendo da NF (1/2)

- ▶ La soluzione di NF sull'istanza di riferimento del BPP ha gli schemi di carico $\{1\}$, $\{2\}$, $\{3\}$, $\{4, 5\}$, $\{6\}$ (contenitori $1, \dots, 5$), carichi $(4, 4, 5, 5, 3)$, $nb = 5$ e $SQ = 91$.
- ▶ Con $f = nb$, la prima mossa esaminata, l'oggetto 1 nel contenitore 5 ($3 + 4 = 7$), svuota il contenitore 1: la soluzione ha 4 contenitori, $\{2\}$, $\{3\}$, $\{4, 5\}$, $\{1, 6\}$.
- ▶ È un ottimo locale: gli oggetti 2 e 3 non entrano in nessun altro contenitore, e i contenitori 4 e 5 contengono due oggetti ciascuno. Entrambe le regole si fermano con 4 contenitori.

Esempio: partendo da NF (2/2)

Con $f = SQ$, best improvement esegue tre mosse:

mossa	schemi di carico	guadagno	SQ
oggetto 1 nel contenitore 5	$\{2\}, \{3\}, \{4, 5\}, \{1, 6\}$	$2 \cdot 4 \cdot (3 - 4 + 4) = 24$	115
oggetto 4 nel contenitore 2	$\{2, 4\}, \{3\}, \{5\}, \{1, 6\}$	$2 \cdot 3 \cdot (4 - 5 + 3) = 12$	127
oggetto 3 nel contenitore 4	$\{2, 4\}, \{3, 5\}, \{1, 6\}$	$2 \cdot 5 \cdot (2 - 5 + 5) = 20$	147

e raggiunge una soluzione ottima con 3 contenitori, attraverso una mossa (la seconda) che non cambia il numero di contenitori. First improvement esegue le stesse tre mosse.

Esperimenti su due istanze medie (1/4)

- ▶ Su una sola istanza piccola può succedere di tutto; gli esperimenti seguenti ripetono le ricerche locali da 100 soluzioni di partenza casuali (seme 1), con best improvement.
- ▶ $P \parallel C_{\max}$ con $m = 4$ e $t = (11, 10, 9, 8, 7, 7, 6, 5, 4, 4, 3, 2)$, $z(I) = 19$. Le 100 assegnazioni casuali hanno un makespan medio di 33.32. La tabella riporta il makespan medio degli ottimi locali, e il numero di esecuzioni che terminano in una soluzione ottima:

	$f = C_{\max}$		$f = \sum_h L_h^2$	
	media	ottime	media	ottime
intorno				
move	20.94	3	20.39	4
swap	24.31	7	23.09	20

Esperimenti su due istanze medie (2/4)

- ▶ L'obiettivo ausiliario migliora entrambi gli intorni.
- ▶ Il solo swap è scarso perché non può cambiare il numero di lavori sulle macchine, che in un'assegnazione casuale è spesso lontano dall'equilibrio.

Esperimenti su due istanze medie (3/4)

- ▶ BPP con $c = 10$ e $w = (7, 6, 6, 5, 5, 4, 4, 4, 3, 3, 2, 1)$, $z(I) = 5$.
- ▶ Le 100 soluzioni di *FF* su ordini casuali degli oggetti usano 5 contenitori in 14 casi, 6 contenitori in 85 casi e 7 contenitori in 1 caso.
- ▶ La tabella riporta il numero di esecuzioni che terminano con 5 contenitori:

intorno	$f = nb$	$f = \sum_b L_b^2$
nessuno (le soluzioni di <i>FF</i>)	14	14
move	14	76
swap	—	14

Esperimenti su due istanze medie (4/4)

- ▶ Con $f = nb$ la ricerca locale move non migliora nessuna delle 100 soluzioni: sono tutte su un plateau.
- ▶ Con l'obiettivo ausiliario, altre 62 esecuzioni raggiungono l'ottimo.
- ▶ Il solo swap non può cambiare il numero di contenitori, ma può preparare le mosse che lo cambiano.

Riepilogo (1/2)

Plateau:

- ▶ Il numero di contenitori e il makespan assumono pochi valori: la maggior parte delle mosse non li cambia
- ▶ Su un plateau la ricerca è cieca e si ferma

L'obiettivo ausiliario:

- ▶ BPP: $\max \sum_b L_b^2$; $P || C_{\max}$: $\min \sum_h L_h^2$
- ▶ Trasferire δ da L_a a L_b lo fa variare di $2\delta(L_b - L_a + \delta)$, in $O(1)$
- ▶ Ogni mossa che migliora l'obiettivo originale è migliorante per esso, e lo sono anche alcune mosse neutre

Riepilogo (2/2)

II BPP:

- ▶ Da FF : 4 contenitori con $f = nb$, l'ottimo 3 con $f = SQ$, attraverso una mossa che prepara la successiva
- ▶ Da NF : 4 contenitori con $f = nb$, l'ottimo 3 con $f = SQ$

Gli esperimenti:

- ▶ $P \parallel C_{\max}$: l'obiettivo ausiliario migliora sia move sia swap
- ▶ BPP: con move, 14 esecuzioni ottime con $f = nb$, 76 con $f = \sum_b L_b^2$

KP: ammissibilità e penalità

Introduzione: KP: ammissibilità e penalità

- ▶ Un vettore binario può superare la capacità dello zaino: non è una soluzione ammissibile
- ▶ In questa sezione confrontiamo due modi di trattare questi vettori: restare ammissibili, oppure penalizzare il peso in eccesso
- ▶ Una penalità lascia passare la ricerca attraverso vettori non ammissibili, e le fa raggiungere soluzioni che una ricerca ammissibile non raggiunge

Due modi di trattare i vettori non ammissibili

- ▶ La rappresentazione del KP con un vettore binario x descrive anche oggetti non ammissibili, il cui peso supera la capacità.
- ▶ Ci sono due modi di trattarli:
 1. **restare ammissibili**: l'intorno contiene solo i vicini ammissibili, come fatto finora;
 2. **penalizzare**: i vettori non ammissibili sono ammessi, e la ricerca massimizza una funzione che sottrae dal profitto una penalità proporzionale al peso in eccesso.

Restare ammissibili può rendere la ricerca miope

- ▶ Nel capitolo «Algoritmi euristici: definizioni», $x = (1, 0, 1, 0)$ è un ottimo locale per flip.
- ▶ Per raggiungere la soluzione ottima $(1, 1, 0, 0)$ con i flip occorre aggiungere l'oggetto 2 e togliere l'oggetto 3, e il vettore intermedio è o $(1, 1, 1, 0)$, non ammissibile, o $(1, 0, 0, 0)$, di profitto $10 < 16$.
- ▶ L'intorno swap risolve questo caso eseguendo i due flip insieme.
- ▶ Le penalità lo risolvono lasciando che la ricerca passi attraverso il vettore non ammissibile.

L'obiettivo penalizzato

Con le penalità, lo spazio di ricerca è l'intero insieme $\{0, 1\}^n$, e la funzione da massimizzare è

$$g_{\alpha}(x) = \sum_{j \in \mathcal{J}} p_j x_j - \alpha \cdot \max \left\{ 0, \sum_{j \in \mathcal{J}} w_j x_j - c \right\},$$

dove $\alpha > 0$ è la **penalità** per ogni unità di peso in eccesso.

Per un x ammissibile, $g_{\alpha}(x)$ è il profitto di x .

Restare ammissibili: la ricerca locale flip+swap

- ▶ La ricerca locale *flip+swap* per il KP esamina i flip e poi gli swap della soluzione corrente, e scarta quelli che superano la capacità.
- ▶ Ne applica uno migliorante scelto con la regola first improvement o best improvement.
- ▶ Si ferma quando nessun vicino ammissibile ha un profitto maggiore.

Algorithm: Passo flip+swap per il KP (1/4)

Input : n oggetti, profitti $p[1..n]$, pesi $w[1..n]$, capacità c ,
soluzione ammissibile $x[1..n]$, regola

Output : *true* se a x è stata applicata una mossa migliorante,
false altrimenti

```
1 function KPFlipSwapStep( $n, p, w, c, x, rule$ )
2    $W := \sum_{j=1}^n w[j] x[j]$            // peso della soluzione
   corrente
3    $\bar{\Delta} := 0; (\bar{j}, \bar{k}) := (0, 0)$  // miglior mossa trovata
   finora ( $\bar{k} = 0$ : flip)
4   for  $j := 1$  to  $n$  do           // flip dell'oggetto  $j$ 
5      $\Delta p := (1 - 2x[j]) p[j]; \Delta w := (1 - 2x[j]) w[j]$ 
     // togliere, o aggiungere, l'oggetto  $j$ 
```

Algorithm: Passo flip+swap per il KP (2/4)

function *KPFlipSwapStep*($n, p, w, c, x, rule$) (continua)

```

6   |   for  $j := 1$  to  $n$  (continua) do
7   |       if  $W + \Delta w \leq c$  and  $\Delta p > \bar{\Delta}$  then
8   |            $\bar{\Delta} \leftarrow \Delta p$ ;  $(\bar{j}, \bar{k}) \leftarrow (j, 0)$ 
9   |           if  $rule = first$  then
10  |               uscire da tutti i cicli
11  |           end
12  |   end
```

Algorithm: Passo flip+swap per il KP (3/4)

function *KPFlipSwapStep*($n, p, w, c, x, rule$) (continua)

```
13  for  $j := 1$  to  $n$  con  $x[j] = 1$  do    // swap:   $j$  fuori,  $k$   
    dentro  
14      for  $k := 1$  to  $n$  con  $x[k] = 0$  do  
15           $\Delta p := p[k] - p[j]; \Delta w := w[k] - w[j]$   
16          if  $W + \Delta w \leq c$  and  $\Delta p > \bar{\Delta}$  then  
17               $\bar{\Delta} \leftarrow \Delta p; (\bar{j}, \bar{k}) \leftarrow (j, k)$   
18              if  $rule = first$  then  
19                  uscire da tutti i cicli  
20              end  
21          end  
22      end  
23  end
```

Algorithm: Passo flip+swap per il KP (4/4)

function *KPFlipSwapStep*($n, p, w, c, x, rule$) (continua)

```
24  | if  $\bar{j} = 0$  then return false           // nessuna mossa  
    |   migliorante: ottimo locale  
25  |  $x[\bar{j}] \leftarrow 1 - x[\bar{j}]$  // flip, oppure togliere l'oggetto  
    |   che esce  
26  | if  $\bar{k} \neq 0$  then  
27  |   |  $x[\bar{k}] \leftarrow 1$    // aggiungere l'oggetto che entra  
28  | end  
29  | return true  
30 end
```

Come funziona il passo flip+swap

- ▶ Con la regola first improvement, se si trova un flip migliorante gli swap non vengono esaminati.
- ▶ Solo i flip che *aggiungono* un oggetto possono migliorare il profitto; i flip che tolgono un oggetto restano nell'intorno per uniformità.
- ▶ Un'iterazione costa tempo $O(n^2)$, a causa degli swap.

Penalizzare il peso in eccesso: la ricerca locale *penalty flip*

- ▶ La ricerca locale *penalty flip* per il KP esamina i flip del vettore corrente, ammissibile o no, e li valuta con g_α .
- ▶ Ne applica uno migliorante scelto con la regola first improvement o best improvement.
- ▶ Si ferma quando nessun flip aumenta g_α .

Algorithm: Passo penalty flip per il KP (1/2)

Input : n oggetti, profitti $p[1..n]$, pesi $w[1..n]$, capacità c ,
penalità α , vettore $x[1..n]$, regola

Output : *true* se a x è stato applicato un flip migliorante,
false altrimenti

```
1 function KPPenaltyFlipStep( $n, p, w, c, \alpha, x, rule$ )
2    $P := \sum_{j=1}^n p[j] x[j]$ ;  $W := \sum_{j=1}^n w[j] x[j]$  // profitto e
   peso di  $x$ 
3    $g := P - \alpha \max\{0, W - c\}$  // valore di  $x$ 
4    $\bar{\Delta} := 0$ ;  $\bar{j} := 0$  // miglior flip trovato finora
5   for  $j := 1$  to  $n$  do
6      $P' := P + (1 - 2x[j]) p[j]$ ;  $W' := W + (1 - 2x[j]) w[j]$ 
       // flip dell'oggetto  $j$ 
7      $\Delta := P' - \alpha \max\{0, W' - c\} - g$  // variazione di
        $g_\alpha$ 
```

Algorithm: Passo penalty flip per il KP (2/2)

function *KPPenaltyFlipStep*($n, p, w, c, \alpha, x, rule$) (continua)

```
    for  $j := 1$  to  $n$  (continua) do
8      |   if  $\Delta > \bar{\Delta}$  then
9      |        $\bar{\Delta} \leftarrow \Delta; \bar{j} \leftarrow j$            // flip migliorante
10     |       if  $rule = first$  then
11     |           uscire dal ciclo
12     |       end
13     |   end
14 end
15 if  $\bar{j} = 0$  then return false           // nessun flip
    |   migliorante: ottimo locale
16  $x[\bar{j}] \leftarrow 1 - x[\bar{j}]$            // applicare il flip
17 return true
18 end
```

La migliore soluzione ammissibile visitata

- ▶ L'ottimo locale può essere non ammissibile.
- ▶ La ricerca deve quindi ricordare la **migliore soluzione ammissibile** visitata, che è la soluzione che restituisce.

Esempio: penalità sull'istanza di riferimento del KP (1/4)

Partiamo dalla soluzione di GR , $x = (1, 0, 1, 0)$, di peso 8 e profitto 16, che è un ottimo locale per l'intorno flip ristretto alle soluzioni ammissibili. I valori g_α dei suoi quattro flip sono

flip	vettore	peso	eccesso	$\alpha = 1$	$\alpha = 2$	$\alpha = \frac{9}{4}$	$\alpha = 3$
1	(0, 0, 1, 0)	3	0	6	6	6	6
2	(1, 1, 1, 0)	12	3	20	17	$\frac{65}{4}$	14
3	(1, 0, 0, 0)	5	0	10	10	10	10
4	(1, 0, 1, 1)	11	2	18	16	$\frac{31}{2}$	14

Esempio: penalità sull'istanza di riferimento del KP (2/4)

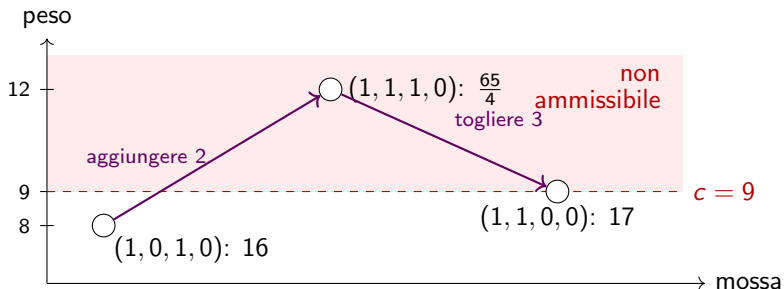
Le due regole eseguono le stesse mosse:

- ▶ $\alpha = 1$: la ricerca aggiunge l'oggetto 2 ($g_1 = 20$), poi l'oggetto 4 ($g_1 = 27 - 6 = 21$), e si ferma in $(1, 1, 1, 1)$, di peso 15: ogni suo flip toglie un oggetto e dà 16, 18, 18 o 20. La penalità è troppo piccola: l'ottimo locale non è ammissibile, e la migliore soluzione ammissibile visitata è quella di partenza, di profitto 16.
- ▶ $\alpha = 2$: la ricerca aggiunge l'oggetto 2 ($g_2 = 17 > 16$) e si ferma in $(1, 1, 1, 0)$, non ammissibile: togliere l'oggetto 3 dà $(1, 1, 0, 0)$ con $g_2 = 17$, non strettamente migliore.

Esempio: penalità sull'istanza di riferimento del KP (3/4)

- ▶ $\alpha = \frac{9}{4}$: la ricerca aggiunge l'oggetto 2 ($g = \frac{65}{4} = 16.25 > 16$) e poi toglie l'oggetto 3, raggiungendo $(1, 1, 0, 0)$, di profitto 17: la soluzione **ottima**, attraverso un vettore non ammissibile (si veda la figura nella prossima slide).
- ▶ $\alpha = 3$: nessun flip migliora, e la ricerca si ferma subito in $(1, 0, 1, 0)$, come la ricerca ammissibile.

Esempio: penalità sull'istanza di riferimento del KP (4/4)



- Con $\alpha = \frac{9}{4}$ la ricerca raggiunge l'ottimo attraverso il vettore non ammissibile $(1, 1, 1, 0)$.

Quanto deve essere grande la penalità?

Con una penalità abbastanza grande,
gli ottimi locali sono ammissibili.

Proposizione 4

Siano i dati interi e sia $\alpha > \max_{j \in \mathcal{J}} p_j$. Allora:

- 1. da un vettore ammissibile, nessun flip che produce un vettore non ammissibile aumenta g_α ;*
- 2. ogni vettore non ammissibile ha un flip che aumenta g_α .*

Dimostrazione

- ▶ (1) Se aggiungere l'oggetto k a un vettore ammissibile produce un eccesso $e \geq 1$, allora g_α varia di

$$p_k - \alpha e \leq p_k - \alpha < 0.$$

- ▶ (2) Sia x non ammissibile, con eccesso $e \geq 1$, e sia j un oggetto selezionato. Togliere j diminuisce l'eccesso di $\min\{w_j, e\} \geq 1$, quindi g_α varia di

$$-p_j + \alpha \min\{w_j, e\} \geq -p_j + \alpha > 0.$$



Penalità grande o piccola

- ▶ Con una penalità **grande** gli ottimi locali sono ammissibili, ma la ricerca non lascia mai la regione ammissibile.
- ▶ Con una penalità **piccola** attraversa la regione non ammissibile, ma può fermarsi lì.
- ▶ I valori utili dipendono dall'istanza (qui $\frac{9}{4}$ funziona, 1, 2 e 3 no).
- ▶ In pratica α si regola durante la ricerca: si aumenta finché la ricerca resta fra vettori non ammissibili, si diminuisce finché resta fra vettori ammissibili.

Restare ammissibili:

- ▶ Flip+swap: solo vicini ammissibili; lo swap esegue due flip insieme
- ▶ La ricerca può essere miope

Penalizzare:

- ▶ $g_\alpha(x)$ = profitto meno α per il peso in eccesso; si restituisce la migliore soluzione ammissibile visitata
- ▶ Sull'istanza di riferimento solo $\alpha = \frac{9}{4}$ raggiunge l'ottimo 17, attraverso $(1, 1, 1, 0)$
- ▶ Con $\alpha > \max_j p_j$ gli ottimi locali sono ammissibili; in pratica α si regola

Variable neighbourhood descent

Introduzione: Variable neighbourhood descent

- ▶ Un ottimo locale per un intorno spesso non è un ottimo locale per un altro
- ▶ In questa sezione combiniamo più intorni in un'unica ricerca, che si ferma solo quando nessuno di essi può migliorare
- ▶ La eseguiamo su $P || C_{\max}$ e sul TSP, e sulle due istanze medie

Più intorni

- ▶ Gli esempi delle sezioni precedenti mostrano che un ottimo locale per un intorno spesso non è un ottimo locale per un altro:
 - ▶ la soluzione di *LPT*, per move e swap;
 - ▶ l'ottimo locale 2-opt di lunghezza 41, per 2-opt e Or-opt.
- ▶ L'idea naturale è usare **più intorni** $N_1, \dots, N_K$, e fermarsi solo quando nessuno di essi contiene un vicino migliorante.

Variable neighbourhood descent

- ▶ La *variable neighbourhood descent* (indicata con *VND*) esplora gli intornoi $N_1, \dots, N_K$ in quest'ordine.
- ▶ Cerca una mossa migliorante nell'intorno corrente N_k :
 - ▶ se ne trova una, la applica e torna a N_1 ;
 - ▶ altrimenti passa a N_{k+1} .
- ▶ Si ferma quando la soluzione corrente è un ottimo locale per tutti i K intornoi.

Algorithm: Variable neighbourhood descent

Input : una soluzione ammissibile x , i passi
 $Step_1, \dots, Step_K$ di K intorni, una regola

Output : un ottimo locale x per tutti i K intorni

```
1 function VND( $x, Step_1, \dots, Step_K, rule$ )
2    $k := 1$                                      // intorno corrente
3   while  $k \leq K$  do
4     if  $Step_k(x, rule)$  then
5        $k \leftarrow 1$       // mossa migliorante applicata:
                          tornare al primo intorno
6     else
7        $k \leftarrow k + 1$       // ottimo locale per  $N_k$ :
                          intorno successivo
8   end
9 end
10 return  $x$ 
11 end
```

VND si ferma in un ottimo locale per tutti gli intorni

- ▶ VND si ferma dopo un numero finito di mosse, per lo stesso argomento usato per la terminazione di LS: ogni mossa migliora strettamente l'obiettivo.
- ▶ Quando si ferma, $Step_K$ è fallito e, dall'ultima mossa migliorante, sono falliti sulla stessa soluzione tutti i passi $Step_1, \dots, Step_{K-1}$.
- ▶ La soluzione restituita è un ottimo locale per ogni N_k , cioè per l'unione $N_1 \cup \dots \cup N_K$.

L'ordine degli intorni

- ▶ Gli intorni sono ordinati dal **più piccolo e meno costoso** al più grande: gli intorni costosi si esplorano solo quando quelli economici sono esauriti.
- ▶ *VND* è la parte deterministica della variable neighbourhood search di Mladenović e Hansen (*Computers & Operations Research* 24, 1997).

Esempio: VND su $P \parallel C_{\max}$ con move e swap

Sull'istanza di riferimento, dalla soluzione di LPT , con $N_1 = \text{move}$ e $N_2 = \text{swap}$, e il makespan come obiettivo:

k	risultato di $Step_k$	carichi	poi
1	nessuna mossa migliorante	(11, 8, 8)	$k = 2$
2	swap dei lavori 1 e 3	(10, 8, 9)	$k = 1$
1	nessuna mossa migliorante	(10, 8, 9)	$k = 2$
2	swap dei lavori 3 e 6	(9, 9, 9)	$k = 1$
1	nessuna mossa migliorante	(9, 9, 9)	$k = 2$
2	nessuno swap migliorante	(9, 9, 9)	stop

Nella soluzione con carichi (10, 8, 9) la mossa migliore dà makespan 11 (lavoro 5 o lavoro 7 dalla macchina 1 alla macchina 2), quindi move è di nuovo esaurito, e swap trova l'ottimo.

Esempio: VND sul TSP con 2-opt e Or-opt

- ▶ Sull'istanza con 10 città, dal giro di NN (lunghezza 49), con $N_1 = 2\text{-opt}$, $N_2 = \text{Or-opt}$ e la regola first improvement:
 - ▶ VND esegue le quattro mosse 2-opt della ricerca locale 2-opt ($49 \rightarrow 47 \rightarrow 46 \rightarrow 43 \rightarrow 41$);
 - ▶ poi la mossa Or-opt che inserisce la città 9 fra le città 3 e 4 ($41 \rightarrow 39$);
 - ▶ e si ferma: il giro ottimo è un ottimo locale per entrambi gli intornoi.
- ▶ Con la regola best improvement, il solo 2-opt raggiunge già l'ottimo.

VND sulle due istanze medie

Sulle due istanze medie delle sezioni precedenti, con 100 soluzioni di partenza casuali (seme 1) e best improvement, VND con $N_1 = \text{move}$ e $N_2 = \text{swap}$ dà:

istanza	obiettivo	esecuzioni che terminano in un ottimo	valore medio
$P \parallel C_{\max}, m = 4, n = 12$	$C_{\max}$	15	20.62
$P \parallel C_{\max}, m = 4, n = 12$	$\sum_h L_h^2$	68	19.32
BPP, $c = 10, n = 12$	$\sum_b L_b^2$	89	5.11

Due miglioramenti che si sommano

- ▶ Rispetto ai singoli intorni, VND con l'obiettivo ausiliario raggiunge l'ottimo
 - ▶ in 68 esecuzioni su 100 per $P \parallel C_{\max}$ (solo move: 4);
 - ▶ in 89 per il BPP (solo move: 76).
- ▶ Combinare gli intorni e usare l'obiettivo ausiliario sono miglioramenti indipendenti, e si sommano.

Variable neighbourhood descent:

- ▶ Esplorare $N_1, \dots, N_K$ in ordine; dopo una mossa migliorante tornare a N_1
- ▶ Si ferma in un ottimo locale per l'unione $N_1 \cup \dots \cup N_K$
- ▶ Ordinare gli intorno dal meno costoso al più costoso

Risultati:

- ▶ $P \parallel C_{\max}$: da *LPT* all'ottimo 9; TSP con 10 città: da 49 all'ottimo 39
- ▶ Con l'obiettivo ausiliario: 68 e 89 esecuzioni ottime su 100

Ricerca locale multi-start

Introduzione: Ricerca locale multi-start

- ▶ Una ricerca locale si ferma nella valle in cui parte
- ▶ In questa sezione la facciamo ripartire da molte soluzioni di partenza e teniamo il miglior ottimo locale
- ▶ Misuriamo sul TSP quanto spesso le ripartenze raggiungono l'ottimo, e vediamo i limiti di questa idea

Ripartire

- ▶ La ricerca locale si ferma nell'ottimo locale della valle in cui parte.
- ▶ Il modo più semplice di esplorare altre valli è **ripartire** da una soluzione diversa, e tenere il miglior ottimo locale.
- ▶ La *ricerca locale multi-start* (indicata con *MSLS*):
 - ▶ genera M soluzioni di partenza, a caso o con un'euristica costruttiva randomizzata;
 - ▶ esegue una ricerca locale da ciascuna di esse, in modo indipendente;
 - ▶ restituisce il miglior ottimo locale trovato.

Algorithm: 2-opt multi-start (1/2)

Input : n città, distanze d , numero di partenze M , seme s del generatore, regola

Output : il miglior ottimo locale 2-opt $\bar{\sigma}$ trovato, e la sua lunghezza $\bar{L}$

```
1 function MultiStartTwoOpt( $n, d, M, s, rule$ )
2    $\bar{L} := +\infty$  // lunghezza del miglior giro trovato finora
3   for  $r := 1$  to  $M$  do
4      $\sigma := \text{RandomTour}(n, s)$  // giro di partenza casuale; aggiorna il seme
5     while  $\text{TwoOptStep}(n, d, \sigma, rule)$  do
6       // ripetere finché  $\sigma$  non è un ottimo locale 2-opt
7     end
```

Algorithm: 2-opt multi-start (2/2)

function *MultiStartTwoOpt*($n, d, M, s, rule$) (continua)

```

8   |   for  $r := 1$  to  $M$  (continua) do
9   |       |   if  $L(\sigma) < \bar{L}$  then
10  |       |       |    $\bar{L} \leftarrow L(\sigma); \bar{\sigma} \leftarrow \sigma$     // nuovo miglior ottimo
11  |       |       |       locale
12  |       |   end
13 end
14  |   end
15  |   return  $(\bar{\sigma}, \bar{L})$ 
16 end
```

Esecuzioni indipendenti

- ▶ Le M ricerche locali sono **indipendenti**: possono essere eseguite in parallelo, e il risultato non dipende dal loro ordine.
- ▶ Il multi-start è la forma più semplice di **diversificazione**.

Esempio: *MSLS* sul TSP con 10 città (1/2)

Sull'istanza con 10 città, 100 giri casuali (seme 1), ciascuno migliorato da una ricerca locale, terminano nei seguenti ottimi locali:

intorno	regola	esecuzioni nell'ottimo 39	altri ottimi locali (esecuzioni)
2-opt	first	46	41 (54)
2-opt	best	87	41 (8), 43 (5)
Or-opt	first	72	43 (26), 44 (2)
Or-opt	best	77	43 (22), 44 (1)
VND (2-opt, Or-opt)	first	100	—
VND (2-opt, Or-opt)	best	100	—

Esempio: *MSLS* sul TSP con 10 città (2/2)

- ▶ Con 2-opt e first improvement, il primo giro casuale termina nell'ottimo locale di lunghezza 41 e il terzo nell'ottimo.
- ▶ Su questa istanza best improvement è molto migliore di first improvement.
- ▶ *VND* trova sempre l'ottimo, l'unico giro che è un ottimo locale per entrambi gli intorni.

Esempio: *MSLS* sul TSP con 30 città (1/4)

- ▶ Le 30 città hanno coordinate

$$(\lfloor 101u_{2k-1} \rfloor, \lfloor 101u_{2k} \rfloor), \quad k = 1, \dots, 30,$$

dove $u_1, u_2, \dots$ sono generati con seme 12345, e d_{ij} è la distanza euclidea arrotondata all'intero più vicino.

- ▶ Il giro ottimo, calcolato con un solutore MIP, ha lunghezza $z(I) = 478$.
- ▶ Il migliore dei 30 giri di *NN* (uno per ogni città di partenza) ha lunghezza 537.
- ▶ I 100 giri casuali (seme 1) hanno una lunghezza media di 1768.73 (fra 1364 e 2080).

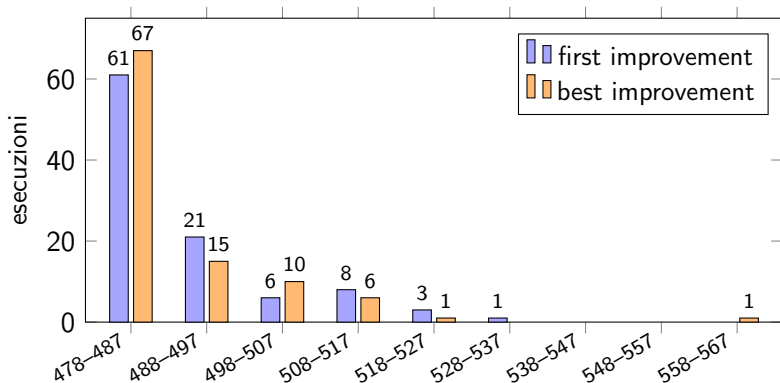
Esempio: *MSLS* sul TSP con 30 città (2/4)

Dopo la ricerca locale 2-opt:

	first improvement	best improvement
ottimi locali distinti	59	54
esecuzioni che terminano nell'ottimo	14	12
lunghezza media (errore relativo)	488.45 (2.2%)	487.24 (1.9%)
lunghezza peggiore (errore relativo)	537 (12.3%)	567 (18.6%)
prima esecuzione che trova l'ottimo	2	9

Esempio: *MSLS* sul TSP con 30 città (3/4)

Gli ottimi locali si distribuiscono come segue (numero di esecuzioni per classe di lunghezza):



Esempio: *MSLS* sul TSP con 30 città (4/4)

- ▶ La ricerca locale trasforma giri di lunghezza circa 1769 in giri entro il 2% dall'ottimo in media.
- ▶ Ma ogni esecuzione termina nella sua valle, e solo circa un'esecuzione su otto raggiunge l'ottimo.

I limiti del multi-start (1/2)

- ▶ **Nessuna memoria.** Ogni esecuzione riparte da zero e ignora ciò che hanno trovato le precedenti.
 - ▶ Se un'esecuzione raggiunge l'ottimo con probabilità q , indipendentemente dalle altre, M esecuzioni lo mancano tutte con probabilità $(1 - q)^M$.
 - ▶ Con $q = 0.12$, come nell'ultimo esempio con best improvement, 10 esecuzioni mancano l'ottimo con probabilità $0.88^{10} \approx 0.28$, e servono 37 esecuzioni per portare questa probabilità sotto l'1%.
- ▶ **La probabilità q si riduce.** Il numero di ottimi locali cresce molto rapidamente con n : con 10 città solo 3 giri sono ottimi locali 2-opt, con 30 città 100 esecuzioni ne trovano più di 50 distinti.
 - ▶ Sulle istanze grandi una partenza casuale non raggiunge quasi mai l'ottimo, e $(1 - q)^M$ è vicino a 1 per ogni M ragionevole.

I limiti del multi-start (2/2)

- ▶ **La maggior parte del tempo si spende a scendere.** Una partenza casuale è lontana dalle buone soluzioni, e la ricerca locale spende la maggior parte delle sue mosse nella discesa; una buona euristica costruttiva dà soluzioni di partenza migliori, ma poche e poco diverse fra loro.
- ▶ Una strategia migliore **userebbe gli ottimi locali già trovati:** ripartire *vicino* a un buon ottimo locale invece che da un punto qualsiasi, oppure continuare la ricerca da esso accettando alcune mosse peggioranti.
- ▶ Sono le idee delle metaeuristiche.

Che cosa dà la ricerca locale, e che cosa non dà

- ▶ La ricerca locale migliora una soluzione ammissibile fino a raggiungere un ottimo locale: un **bound primale** migliore, spesso molto migliore di quello di partenza.
- ▶ Non dà un bound duale, e il fatto che si sia fermata non significa che sia arrivata.
- ▶ Sull'istanza con 30 città sappiamo che 478 è ottimo solo perché lo ha dimostrato un solutore MIP; la sola ricerca locale non saprebbe distinguere dall'ottimo l'ottimo locale di lunghezza 479, trovato in molte esecuzioni.

Un ottimo locale non è un bound migliore

Un ottimo locale non è un bound migliore:
è una soluzione ammissibile, cioè un lato del gap.

La sua qualità si può valutare solo rispetto
a un bound duale calcolato in un altro modo.

Multi-start:

- ▶ M ricerche locali indipendenti da soluzioni di partenza casuali o randomizzate; si tiene la migliore
- ▶ La forma più semplice di diversificazione

Sul TSP:

- ▶ 10 città: *VND* raggiunge l'ottimo in tutte le 100 esecuzioni
- ▶ 30 città: entro il 2% dall'ottimo in media, ma solo circa un'esecuzione su otto è ottima

I limiti:

- ▶ Nessuna memoria, una probabilità q che si riduce, tempo speso a scendere; nessun bound duale

Implementazione in C++



Introduzione: Implementazione in C++

- ▶ Gli algoritmi del capitolo sono implementati in C++, una funzione per file
- ▶ Mostriamo solo brevi estratti: la ricerca locale generica, la variable neighbourhood descent e come si chiamano
- ▶ Il codice completo è nella cartella del capitolo e nei suoi notebook

Il codice

- ▶ Il codice è nella cartella `LOCAL_SEARCH_CODE`: una funzione per file in `src/`, le dichiarazioni in `src/LocalSearch.h`.
- ▶ Il comando `make` compila il programma, e `make test` esegue tutti gli esempi delle note e tutte le istanze degli esercizi, e confronta i risultati con quelli calcolati da un motore indipendente.
- ▶ Nel codice gli oggetti, i lavori, le macchine, i contenitori e le città sono indicizzati da 0, e la città 0 resta in prima posizione nel giro. La logica dell'algoritmo è invariata.
- ▶ Le costanti `FIRST`, `BEST` scelgono la regola, `CMAX`, `SQ`, `NB` l'obiettivo; `Vec` è `std::vector<int>` e `Mat` un vettore di `Vec`.

La ricerca locale generica

La **ricerca locale** generica riceve il passo come funzione senza argomenti, che agisce sulla soluzione corrente:

```
1 int LocalSearch(const std::function<bool()>& Step)
2 {
3     int moves = 0; // number of moves performed
4     while (Step()) // an improving neighbour was found and accepted
5     {
6         moves = moves + 1;
7     }
8     return moves; // the current solution is a local optimum
9 }
```

La variable neighbourhood descent

Il cuore di VND, che riceve i passi $\text{Steps}[0], \dots, \text{Steps}[K-1]$ dei K intorni:

```
1 while (k < K)
2 {
3     if (Steps[k]()) // improving neighbour found in N_k, and accepted
4     {
5         moves = moves + 1;
6         k = 0; // back to the first neighbourhood
7     }
8     else
9     {
10        k = k + 1; // local optimum for N_k: try the next neighbourhood
11    }
12 }
```

Chiamare le ricerche locali

Per esempio, la ricerca locale 2-opt su un giro π , e *VND* con 2-opt e Or-opt, sono

```
1 LocalSearch([&]() { return TwoOptStep(n, d, pi, BEST); });  
2 VND({ [&]() { return TwoOptStep(n, d, pi, FIRST); },  
3      [&]() { return OrOptStep(n, d, pi, FIRST); } });
```

- Gli altri file contengono i passi del capitolo (TwoOptStep, OrOptStep, PCmaxMoveStep, PCmaxSwapStep, BPPMoveStep, BPPSwapStep, KPFlipSwapStep, KPPenaltyFlipStep), il 2-opt multi-start, il generatore di Park–Miller e i giri casuali.

Il codice:

- ▶ Una funzione per file in `LOCAL_SEARCH_CODE/src/`; `make test` controlla tutti gli esempi
- ▶ Indici da 0, stessa logica dell'algoritmo

Il progetto:

- ▶ Un passo esplora un intorno e applica una mossa migliorante, restituendo `true`, oppure restituisce `false`
- ▶ `LocalSearch` e `VND` chiamano soltanto i passi

Errori tipici

Introduzione: Errori tipici

- ▶ Raccogliamo gli errori che si fanno più spesso con la ricerca locale
- ▶ Riguardano l'intorno, l'obiettivo, l'ordine delle mosse, la mossa 2-opt e il significato di un ottimo locale
- ▶ Controllarli prima di eseguire a mano una ricerca locale

Errori tipici (1/3)

Osservazione 1: Dimenticare che un ottimo locale dipende dall'intorno

La soluzione di LPT è un ottimo locale per move, ma non per swap.

Osservazione 2: Usare il numero di contenitori come unico obiettivo

È cieco: la maggior parte dei vicini ha lo stesso valore, e la ricerca si ferma su un plateau; l'obiettivo ausiliario SQ li distingue.

Osservazione 3: Ignorare l'ordine di esame

Le mosse sono esaminate in un ordine fissato e i pareggi vanno alla prima esaminata (nell'esempio della soluzione di FF l'oggetto 3 viene prima dell'oggetto 5).

Osservazione 4: Calcolare il Δ sbagliato per una mossa 2-opt

Contano solo i due lati tolti e i due lati aggiunti; i due lati tolti non possono avere una città in comune.

Osservazione 5: Concludere che un ottimo locale è ottimo

Solo un bound duale può dirlo.

Intorno e obiettivo:

- ▶ Un ottimo locale dipende dall'intorno
- ▶ Il numero di contenitori è cieco: usare l'obiettivo ausiliario

Mosse:

- ▶ Ordine di esame fissato, pareggi alla prima mossa esaminata
- ▶ 2-opt: contano solo i due lati tolti e i due lati aggiunti

Ottimalità:

- ▶ Senza un bound duale un ottimo locale non è dimostrato ottimo

In sintesi

Introduzione: In sintesi

- ▶ Raccogliamo le idee chiave del capitolo sulla ricerca locale
- ▶ Il quadro riassume i risultati sulle istanze di riferimento

Riepilogo (1/2)

- ▶ Una ricerca locale si sposta su un vicino migliore finché ne esiste uno: restituisce un ottimo locale, un bound primale migliore.
- ▶ First improvement è più veloce a ogni passo, best improvement prende il guadagno massimo; entrambe si fermano in un ottimo locale dell'intorno.
- ▶ Valutazione incrementale, obiettivi ausiliari e penalità rendono la ricerca veloce e capace di attraversare plateau e regioni non ammissibili.
- ▶ La variable neighbourhood descent e il multi-start allargano ciò che la ricerca può raggiungere, ma le ripartenze non hanno memoria.

Riepilogo (2/2)

Quadro delle istanze di riferimento. Soluzione di partenza e ottimo locale:

problema	partenza	ricerca locale	$z(I)$
TSP	25 (<i>NN</i>)	18 (2-opt)	18
$P \parallel C_{\max}$	11 (<i>LPT</i>)	9 (swap, = $LB(I)$)	9
BPP	4 (<i>FF</i>)	3 (move, <i>SQ</i>)	3

I bound duali vengono dal capitolo «Algoritmi euristici: definizioni»: la sola ricerca locale non certifica il suo risultato.