Metaeuristiche (2/2)

Indice

- Variable neighbourhood search
  - Esempio
- GRASP
  - GRASP per il TSP, e l'effetto di α
- Large neighbourhood search
  - Large neighbourhood search per il BPP: riparazione euristica ed esatta
  - Adaptive large neighbourhood search
- Metodi basati su popolazioni: algoritmi genetici e memetici
- Confronto e conclusioni
  - I metodi sulle istanze medie
  - Intensificazione e diversificazione
  - Parametri e calibrazione
  - Che cosa questi metodi non usano
- Errori tipici
- Implementazione in C++
- In sintesi

Dove eravamo (1/2)

- Una **metaeuristica** è una strategia generale che guida una ricerca locale oltre il primo ottimo locale, per esplorare altre regioni dell'insieme delle soluzioni ammissibili.
- I metodi sono illustrati sul KP, sul BPP, sul TSP e su $P \,||\, C_{\max}$, con le ricerche locali TwoOptLS, MoveDescent e KPLocalSearch del capitolo «Ricerca locale».
- I numeri casuali sono prodotti dal minimal standard generator di Park e Miller: con $seed = 1$ i primi sono 0.0000, 0.1315, 0.7556, 0.4587, 0.5328, 0.2190, 0.0470, 0.6789.
- Le istanze di riferimento hanno valori ottimi 17 (KP), 3 (BPP), 18 (TSP) e 9 ($P \,||\, C_{\max}$); le istanze medie TSP20, KP20, PCM15 e BPP24 hanno valori ottimi 291, 521, 69 e 9.

Dove eravamo (2/2)

- Il **simulated annealing** accetta un vicino casuale peggiore con probabilità $e^{-\delta/T}$, e abbassa gradualmente la temperatura T.
- La **tabu search** si sposta sul miglior vicino non tabu, anche se è peggiore, e vieta per *tenure* iterazioni le mosse che annullerebbero quelle recenti.
- L'**iterated local search** perturba l'ottimo locale corrente con un kick, scende di nuovo, e accetta il nuovo ottimo locale se è migliore.

Questa seconda parte presenta VNS, GRASP, large neighbourhood search, algoritmi genetici e memetici,
e confronta tutti i metodi sulle istanze medie.

Variable neighbourhood search

Introduzione: Variable neighbourhood search

- La variable neighbourhood search esce da un ottimo locale *cambiando l'intorno*: quando la ricerca è bloccata, ne campiona uno più grande
- Vediamo il suo pseudocodice per $P \,||\, C_{\max}$, il ruolo dello shaking, e la differenza con la variable neighbourhood descent del capitolo «Ricerca locale»
- Poi la eseguiamo a mano sull'istanza di riferimento e con dieci semi su PCM15

L'idea della variable neighbourhood search

- La *variable neighbourhood search* (indicata con *VNS*) usa una successione di intorni $N_1, N_2, \ldots, N_{k_{\max}}$ di dimensione crescente.
- Dall'ottimo locale corrente estrae una soluzione casuale in N_k (*shaking*), le applica una ricerca locale, e
  - si sposta sul nuovo ottimo locale se è migliore, ripartendo da $k = 1$;
  - altrimenti prova l'intorno successivo, più grande, N_{k+1}.

Vediamo ora lo pseudocodice di *VNS* per $P \,||\, C_{\max}$, dove N_k contiene gli assegnamenti ottenuti con k mosse e la ricerca locale è MoveDescent.

Algorithm: Variable neighbourhood search per $P \,||\, C_{\max}$ (1/3)

Input : tempi $t[1..n]$, m macchine, assegnamento di partenza $mach[1..n]$, $k_{\max}$, $maxIter$

Output : un assegnamento $mach$ e il suo makespan C

1 **function** *VNS*$(t, m, mach, k_{\max}, maxIter)$

2 $\quad$ $\mathrm{LB} := \max\{\max_j t[j],\ \lceil \sum_j t[j]/m \rceil\}$

3 $\quad$ $mach := \textsc{MoveDescent}(mach)$; $L :=$ carichi di $mach$; $C := \max_h L[h]$

4 $\quad$ $it := 0$; $k := 1$

Algorithm: Variable neighbourhood search per $P \,||\, C_{\max}$ (2/3)

```
function VNS(t, m, mach, k_max, maxIter) (continua)
5     while it < maxIter and C > LB do
6         it := it + 1
7         nw := Shake(mach, k)    // soluzione casuale in N_k
8         nw := MoveDescent(nw); Lnew := carichi di nw
              // ricerca locale
9         if Lnew è lessicograficamente migliore di L then
10            mach := nw; L := Lnew; k := 1; C := max_h L[h]
                  // spostamento
11        else
12            if k < k_max then k := k + 1 else k := 1
13        end
14    end
15    return mach, C
16 end
```

Algorithm: Variable neighbourhood search per $P \,||\, C_{\max}$ (3/3)

```
17 function Shake(mach, k)
18 |   for s := 1 to k do
19 |   |   j := 1 + ⌊RAND() · n⌋;  g := 1 + ⌊RAND() · (m − 1)⌋
20 |   |   mach[j] := la g-esima macchina diversa da mach[j]
21 |   end
22 |   return mach
23 end
```

Tre osservazioni

- La *VNS* è stata proposta da Mladenović e Hansen (1997). Si basa su tre osservazioni:
  - un ottimo locale per un intorno non è necessariamente un ottimo locale per un altro;
  - un ottimo globale è un ottimo locale per ogni intorno;
  - e gli ottimi locali di intorni diversi sono spesso vicini fra loro.

- Lo **shaking** diversifica, e la dimensione k controlla quanto: un k piccolo esplora le soluzioni vicine all'ottimo locale corrente, un k più grande va più lontano solo quando quelle più vicine hanno fallito.

- La **ricerca locale** intensifica. Tornare a $k = 1$ dopo ogni miglioramento concentra la ricerca vicino alla nuova soluzione, migliore.

VND e VNS

- Anche la variable neighbourhood descent (*VND*) del capitolo «Ricerca locale» usa più intorni, ma in modo *deterministico*: li esplora completamente, uno dopo l'altro, e si ferma in una soluzione che è un ottimo locale per tutti.
- La *VNS* esplora gli intorni grandi solo per *campioni casuali*, ed è questo che rende utilizzabili intorni molto grandi; la ricerca locale dentro la *VNS* può essere essa stessa una *VND*.
- Qui N_k contiene gli assegnamenti ottenuti spostando k lavori (o meno, se lo stesso lavoro è estratto due volte).
- Poiché uno swap di due lavori è la combinazione di due spostamenti, N_2 contiene l'intorno swap: lo shaking con $k = 2$ può riprodurre gli swap che l'intorno move non vede.

Esempio: *VNS* sull'istanza di riferimento di $P \,||\, C_{\max}$ (1/3)

- Partiamo dalla soluzione LPT, con carichi $(11, 8, 8)$, e usiamo $k_{\max} = 3$ e $seed = 1$ (numeri casuali 0.0000, 0.1315, 0.7556, 0.4587, 0.5328, 0.2190, 0.0470, 0.6789).
- MoveDescent non cambia la soluzione LPT, che è un ottimo locale per l'intorno move (come visto nell'esempio della tabu search).
- Ogni spostamento casuale usa due numeri: $j = 1 + \lfloor 7u \rfloor$ e $g = 1 + \lfloor 2u \rfloor$.

Esempio: *VNS* sull'istanza di riferimento di $P \,||\, C_{\max}$ (2/3)

- **Iterazione 1**, $k = 1$: il lavoro $j = 1 + \lfloor 7 \cdot 0.0000 \rfloor = 1$ passa da M_1 alla prima altra macchina ($g = 1 + \lfloor 2 \cdot 0.1315 \rfloor = 1$), M_2: carichi $(6, 13, 8)$.
  MoveDescent sposta il lavoro 6 da M_2 a M_1: carichi $(9, 10, 8)$, $C_{\max} = 10$, migliore di 11. La nuova soluzione $J_1 = \{5, 6, 7\}$, $J_2 = \{1, 2\}$, $J_3 = \{3, 4\}$ viene accettata, e $k = 1$.

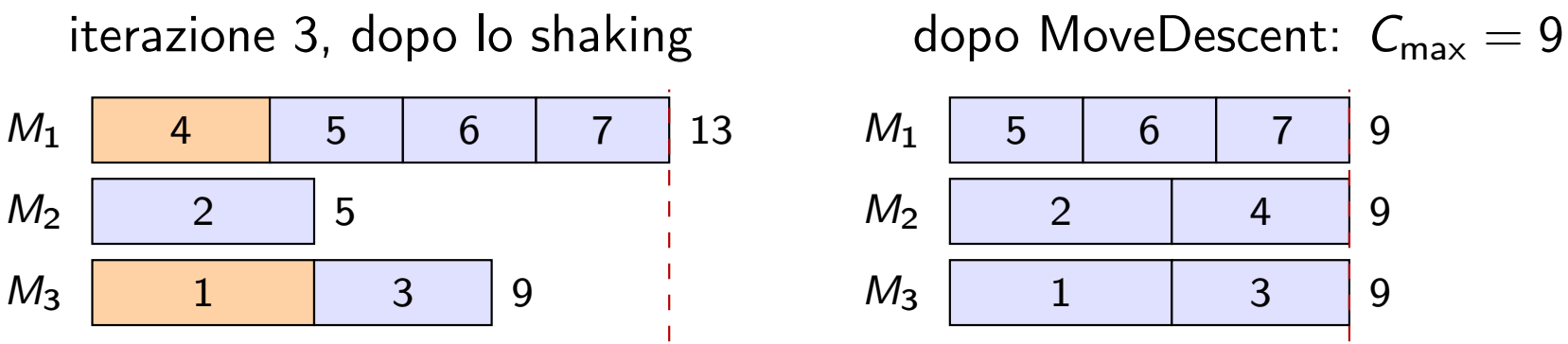
- **Iterazione 2**, $k = 1$: il lavoro $j = 1 + \lfloor 7 \cdot 0.7556 \rfloor = 6$ passa da M_1 a M_2 ($g = 1 + \lfloor 2 \cdot 0.4587 \rfloor = 1$): carichi $(6, 13, 8)$.
  MoveDescent riporta il lavoro 6 su M_1 e restituisce la soluzione corrente, che non è migliore: $k = 2$.

Esempio: *VNS* sull'istanza di riferimento di $P \,||\, C_{\max}$ (3/3)

- **Iterazione 3**, $k = 2$: il lavoro $j = 1 + \lfloor 7 \cdot 0.5328 \rfloor = 4$ passa da M_3 alla prima altra macchina M_1 ($g = 1 + \lfloor 2 \cdot 0.2190 \rfloor = 1$); poi il lavoro $j = 1 + \lfloor 7 \cdot 0.0470 \rfloor = 1$ passa da M_2 alla seconda altra macchina M_3 ($g = 1 + \lfloor 2 \cdot 0.6789 \rfloor = 2$): carichi $(13, 5, 9)$. MoveDescent sposta il lavoro 4 da M_1 a M_2: carichi $(9, 9, 9)$. La soluzione viene accettata, e poiché $C_{\max} = 9 = \mathrm{LB}$ la ricerca si ferma: $VNS(I) = 9 = z(I)$.

- Rispetto alla soluzione corrente dell'iterazione 3, i due spostamenti casuali e la discesa hanno scambiato i lavori 1 e 4: uno swap, che l'intorno move di MoveDescent non può eseguire.

L'iterazione 3 della *VNS*



- I due lavori spostati dallo shaking sono in arancione: il lavoro 4 su M_1 e il lavoro 1 su M_3.
- MoveDescent sposta poi il lavoro 4 su M_2, e le tre macchine hanno carico 9.

La *VNS* su PCM15 con dieci semi

- Su PCM15, dalla soluzione LPT (73) e con al più 300 iterazioni, la *VNS* dà i seguenti makespan per $seed = 1, \ldots, 10$, e il numero di iterazioni eseguite:

seed	1	2	3	4	5	6	7	8	9	10
$k_{\max} = 1$: $C_{\max}$	69	70	70	70	69	69	69	69	70	69
iterazioni	27	300	300	300	48	59	20	43	300	13
$k_{\max} = 2$: $C_{\max}$	70	69	69	69	69	70	69	70	70	69
iterazioni	300	41	126	285	33	300	31	300	300	26
$k_{\max} = 3$: $C_{\max}$	70	69	69	69	69	69	69	70	70	69
iterazioni	300	32	38	29	26	11	25	300	300	25

Che cosa mostrano i semi

- Con $k_{\max} = 3$, la *VNS* raggiunge l'ottimo 69 in 7 esecuzioni su 10, di solito in poche decine di iterazioni; la migliore tabu search su PCM15 richiedeva una durata tabu calibrata con cura.
- Quando la *VNS* fallisce, la ricerca è bloccata a 70: lo shaking esplora al più tre mosse di distanza, e l'ottimo è più lontano.
- Si possono aggiungere un $k_{\max}$ più grande, o delle ripartenze, al prezzo di più iterazioni.

Riepilogo

Variable neighbourhood search:

- shaking in N_k, ricerca locale, spostamento se migliore e ritorno a $k = 1$, altrimenti $k + 1$
- lo shaking diversifica, la ricerca locale intensifica
- a differenza della *VND*, campiona a caso gli intorni grandi

Su $P \,||\, C_{\max}$:

- N_2 contiene gli swap che l'intorno move non vede
- istanza di riferimento: ottimo in tre iterazioni; PCM15 con $k_{\max} = 3$: ottimo in 7 esecuzioni su 10

GRASP

Introduzione: GRASP

- La GRASP esce da un ottimo locale *ripartendo* da una nuova soluzione, costruita da un algoritmo greedy randomizzato
- Vediamo il suo pseudocodice per il KP, la restricted candidate list e il ruolo del parametro α
- Poi eseguiamo a mano due iterazioni sul KP, e misuriamo l'effetto di α su TSP20 e KP20

L'idea della GRASP

- La *greedy randomized adaptive search procedure* (indicata con *GRASP*) è un metodo multi-start.
- A ogni iterazione costruisce una soluzione con un algoritmo greedy *randomizzato*, che sceglie l'elemento successivo a caso fra i candidati migliori (la *restricted candidate list*, RCL) invece di prendere sempre il migliore,
- e migliora la soluzione con una ricerca locale.
- Si restituisce il miglior ottimo locale trovato.

Vediamo ora lo pseudocodice di *GRASP* per il KP, con l'algoritmo greedy randomizzato per efficienza e KPLocalSearch.

Algorithm: GRASP per il KP (1/2)

Input : istanza del KP $p[1..n]$, $w[1..n]$, c; parametro $\alpha \in [0,1]$; numero di iterazioni *maxIter*

Output : migliore soluzione trovata

```
1 function GRASP(p, w, c, α, maxIter)
2     zbest := −1
3     for it := 1 to maxIter do
4         x := RandomizedGreedy(p, w, c, α)   // costruzione
5         x := KPLocalSearch(x)               // ricerca locale
6         if Σ_j p[j]x[j] > zbest then best := x;  zbest := Σ_j p[j]x[j]
7     end
8     return best
9 end
```

Algorithm: GRASP per il KP (2/2)

```
10 function RandomizedGreedy(p, w, c, α)
11     x := (0, ..., 0);  rc := c        // capacità residua
12     while qualche oggetto con x[j] = 0 ha w[j] ≤ rc do
13         e_max, e_min := massimo e minimo di p[j]/w[j] su
            questi oggetti candidati
14         RCL := candidati con
            p[j]/w[j] ≥ e_max − α (e_max − e_min), per
            p[j]/w[j] non crescente
15         j := l'elemento di RCL in posizione
            1 + ⌊RAND() · |RCL|⌋
16         x[j] := 1;  rc := rc − w[j]
17     end
18     return x
19 end
```

La restricted candidate list

- La *GRASP* è stata introdotta da Feo e Resende (1989) per il problema del set covering e presentata come metaeuristica generale in Feo e Resende (1995). La costruzione randomizzata con una restricted candidate list è quella del capitolo «Euristiche costruttive».
- La RCL è definita da una soglia sul punteggio greedy, controllata da α.

KP (massimizzare l'efficienza p_j/w_j degli oggetti che entrano):

$$RCL = \left\{ j \text{ candidato} : \frac{p_j}{w_j} \geq e_{\max} - \alpha\,(e_{\max} - e_{\min}) \right\}.$$

TSP (minimizzare la distanza dall'ultima città ℓ del giro parziale):

$$RCL = \left\{ j \text{ non visitata} : d_{\ell j} \leq d_{\min} + \alpha\,(d_{\max} - d_{\min}) \right\}.$$

Il parametro α

- Con $\alpha = 0$ la RCL contiene solo i candidati migliori, e la costruzione è l'algoritmo greedy: ogni iterazione costruisce la stessa soluzione (a meno di pareggi), e la *GRASP* è una sola ricerca locale.
- Con $\alpha = 1$ la RCL contiene tutti i candidati, e la costruzione è casuale.
- I valori intermedi danno soluzioni **diverse** (diversificazione) e **buone** (una buona partenza per la ricerca locale, che poi intensifica).
- Le iterazioni della *GRASP* sono **indipendenti**: niente viene appreso dall'una all'altra, e le iterazioni si possono eseguire in parallelo.
- Alcune estensioni aggiungono memoria: la GRASP *reattiva* adatta α ai risultati, e il *path relinking* esplora le soluzioni fra due ottimi locali buoni.

Esempio: *GRASP* sull'istanza di riferimento del KP (1/3)

- Con $\alpha = \frac{1}{2}$ e *seed* $= 2$ (numeri casuali 0.0000, 0.2631, 0.5112, 0.9173) eseguiamo due iterazioni.
- Le efficienze sono $\frac{10}{5} = 2$, $\frac{7}{4} = 1.75$, $\frac{6}{3} = 2$, $\frac{4}{3} \approx 1.33$.

Iterazione 1.

- Con $rc = 9$ tutti gli oggetti entrano: $e_{\max} = 2$, $e_{\min} = \frac{4}{3}$, la soglia è $2 - \frac{1}{2} \cdot \frac{2}{3} = \frac{5}{3} \approx 1.67$, e $RCL = (1, 3, 2)$.
- Con $u = 0.0000$ si sceglie il primo elemento: l'oggetto 1, $rc = 4$.
- I candidati sono gli oggetti 2, 3, 4, la soglia è ancora $\frac{5}{3}$, $RCL = (3, 2)$, e $u = 0.2631$ sceglie la posizione $1 + \lfloor 2 \cdot 0.2631 \rfloor = 1$: l'oggetto 3, $rc = 1$, e nessun oggetto entra.
- La costruzione dà $\{1, 3\}$, di profitto 16; KPLocalSearch fa uscire l'oggetto 3 ed entrare l'oggetto 2: $\{1, 2\}$, profitto 17.

Esempio: *GRASP* sull'istanza di riferimento del KP (2/3)

Iterazione 2.

- $RCL = (1, 3, 2)$ e $u = 0.5112$ sceglie la posizione $1 + \lfloor 3 \cdot 0.5112 \rfloor = 2$: l'oggetto 3, $rc = 6$.
- I candidati sono gli oggetti 1, 2, 4, la soglia è $\frac{5}{3}$, $RCL = (1, 2)$, e $u = 0.9173$ sceglie la posizione 2: l'oggetto 2, $rc = 2$.
- La costruzione dà $\{2, 3\}$, di profitto 13, una soluzione *diversa* e peggiore di quella greedy.
- La ricerca locale fa uscire l'oggetto 3 ed entrare l'oggetto 1: $\{1, 2\}$, profitto 17.

Esempio: *GRASP* sull'istanza di riferimento del KP (3/3)

$$GRASP(I) = 17 = z(I).$$

- Su questa istanza piccola ogni partenza porta all'ottimo.
- Su istanze più grandi partenze diverse portano a ottimi locali diversi.

GRASP per il TSP

- Per il TSP la costruzione randomizzata è l'euristica nearest neighbour con una RCL: dall'ultima città ℓ del giro parziale, la città successiva si sceglie a caso fra le città non visitate la cui distanza da ℓ è al più $d_{\text{min}} + \alpha(d_{\text{max}} - d_{\text{min}})$, e la ricerca locale è TwoOptLS.

- La tabella della slide successiva riporta i risultati della *GRASP* con 20 iterazioni su TSP20 e su KP20, per 10 esecuzioni ($seed = 1, \ldots, 10$):
  - valore minimo, medio e massimo della soluzione migliore,
  - e numero di esecuzioni che raggiungono l'ottimo.

L'effetto di α

α	TSP20 ($z(I) = 291$) min	media	max	ottime	KP20 ($z(I) = 521$) min	media	max	ottime
0	319	319.0	319	0	514	514.0	514	0
0.1	291	291.0	291	10	521	521.0	521	10
0.25	291	291.0	291	10	521	521.0	521	10
0.5	291	291.0	291	10	521	521.0	521	10
1	291	291.0	291	10	519	520.8	521	9

- Con $\alpha = 0$ tutte le iterazioni sono uguali: il risultato è l'ottimo locale raggiunto dalla soluzione greedy (319 per TSP20, 514 per KP20), in ogni esecuzione.
- Basta un po' di randomizzazione per raggiungere l'ottimo in ogni esecuzione su queste istanze; con $\alpha = 1$ le costruzioni sono casuali, e su KP20 un'esecuzione manca l'ottimo.

Riepilogo

GRASP:

- multi-start: costruzione greedy randomizzata, poi ricerca locale; si restituisce il miglior ottimo locale
- la RCL contiene i candidati entro una soglia fissata da α
- $\alpha = 0$: greedy, una sola ricerca locale; $\alpha = 1$: costruzione casuale

Risultati:

- le iterazioni sono indipendenti e si possono eseguire in parallelo
- con $0 < \alpha < 1$, la *GRASP* raggiunge l'ottimo di TSP20 e KP20 in ogni esecuzione

Large neighbourhood search

Introduzione: Large neighbourhood search

- La large neighbourhood search esce da un ottimo locale *distruggendo* una parte grande della soluzione e *riparandola*
- Vediamo il suo pseudocodice per il TSP e per il BPP, con una riparazione euristica o esatta
- Poi vediamo la sua versione adattiva, che impara quale metodo di distruzione funziona meglio

L'idea della large neighbourhood search

- La *large neighbourhood search* (indicata con *LNS*) migliora una soluzione *distruggendone* una parte e *riparandola*:
  - un metodo di distruzione toglie alcuni elementi (oggetti, città, lavori) dalla soluzione corrente,
  - un metodo di riparazione li reinserisce,
  - e la nuova soluzione sostituisce quella corrente se è soddisfatto un criterio di accettazione.

- L'intorno di una soluzione è l'insieme di tutte le soluzioni che distruzione e riparazione possono produrre, ed è *grande*.

Vediamo ora lo pseudocodice di *LNS* per il TSP, che toglie q città casuali e le reinserisce con l'inserimento più economico (cheapest insertion).

Algorithm: Large neighbourhood search per il TSP (1/2)

Input : distanze d, giro di partenza σ, numero q di città da togliere, numero di iterazioni *maxIter*

Output : il giro corrente alla fine

```
1  function LNS(d, σ, q, maxIter)
2      cur := σ;  lcur := L(cur)
3      for it := 1 to maxIter do
4          nw := cur
5          removed := RandomRemoval(nw, q)
               // distruzione
6          CheapestInsertionRepair(nw, removed)
               // riparazione
7          if L(nw) < lcur then cur := nw;  lcur := L(nw)
               // accettazione
8      end
9      return cur
10 end
```

Algorithm: Large neighbourhood search per il TSP (2/2)

```
11 function RandomRemoval(σ, q)
12     for s := 1 to q do
13         k := 2 + ⌊RAND() · (|σ| − 1)⌋   // la città 1 non
               viene mai tolta
14         aggiungere σ[k] in coda a removed, e cancellarla da σ
15     end
16     return removed
17 end
18 function CheapestInsertionRepair(σ, removed)
19     foreach città v di removed, nell'ordine di rimozione do
20         inserire v fra le città consecutive a, b di σ di minimo
               d_av + d_vb − d_ab
21     end
22 end
```

Un intorno grande, esplorato implicitamente

- La *LNS* è stata introdotta da Shaw (1998) per problemi di vehicle routing, in cui la riparazione era fatta con la constraint programming.
- La **dimensione** dell'intorno cresce molto rapidamente con il numero q di elementi tolti: per il TSP ci sono $\binom{n-1}{q}$ modi di scegliere le città tolte, e molti modi di reinserirle.
- Un intorno del genere non si può elencare e valutare mossa per mossa, come in una ricerca locale: viene esplorato **implicitamente**, dal metodo di riparazione.

Il metodo di riparazione

- Il metodo di riparazione è un **sotto-algoritmo**: un'euristica costruttiva (cheapest insertion, FFD), o un algoritmo *esatto* per il sottoproblema lasciato dal metodo di distruzione.

- Più forte è la riparazione, migliore è il miglior vicino trovato; più grande è q, più grande è l'intorno, e più costosa la sua esplorazione.

Distruzione e riparazione

- Il metodo di distruzione **diversifica**: di solito è casuale, e q controlla quanto lontano va la ricerca.
- Il metodo di riparazione **intensifica**: ricostruisce la parte distrutta nel modo migliore possibile.
- Togliere elementi *correlati* (città vicine fra loro, oggetti degli stessi contenitori) rende la riparazione più efficace che togliere elementi non correlati, che possono solo essere rimessi dov'erano.

Esempio: *LNS* sull'istanza di riferimento del TSP (1/3)

- Dal giro nearest neighbour $(1, 3, 2, 4, 5)$ di lunghezza 25, con $q = 2$ e $seed = 1$.
- Il primo numero casuale 0.0000 toglie la città in posizione $2 + \lfloor 4 \cdot 0.0000 \rfloor = 2$, la città 3; il giro parziale è $(1, 2, 4, 5)$.
- 0.1315 toglie la città in posizione $2 + \lfloor 3 \cdot 0.1315 \rfloor = 2$, la città 2.
- Il giro parziale $(1, 4, 5)$ ha lunghezza $2 + 7 + 9 = 18$.

Esempio: *LNS* sull'istanza di riferimento del TSP (2/3)

Cheapest insertion della città 3:

$$
\begin{aligned}
(1,4): &\ d_{13} + d_{34} - d_{14} = 2 + 4 - 2 = 4, \\
(4,5): &\ 4 + 7 - 7 = 4, \\
(5,1): &\ 7 + 2 - 9 = 0,
\end{aligned}
$$

quindi la città 3 va fra 5 e 1: $(1,4,5,3)$.

Esempio: *LNS* sull'istanza di riferimento del TSP (3/3)

Cheapest insertion della città 2:

$$(1,4): \ 5+3-2=6, \qquad (4,5): \ 3+4-7=0,$$
$$(5,3): \ 4+4-7=1, \qquad (3,1): \ 4+5-2=7,$$

quindi la città 2 va fra 4 e 5: $(1,4,2,5,3)$, di lunghezza 18, accettato.

- È il giro ottimo, percorso nel verso opposto: un solo passo di distruzione e riparazione ha cambiato quattro lati del giro.

Large neighbourhood search per il BPP

- Per il BPP il metodo di distruzione naturale **svuota contenitori**: il contenitore meno carico, che è quello di cui vorremmo liberarci, e $q - 1$ altri contenitori scelti a caso.
- Il metodo di riparazione reimpacchetta gli oggetti tolti in contenitori nuovi, mentre gli altri contenitori restano come sono.
- Il nuovo impacchettamento è accettato se usa meno contenitori, o lo stesso numero di contenitori e un valore dell'obiettivo ausiliario $\sum_b (\sum_{j \in P_b} w_j)^2$ non minore: sui plateau dell'obiettivo piatto, la ricerca si muove verso contenitori più pieni.

Large neighbourhood search per il BPP: la riparazione

- La riparazione risolve un **sottoproblema di BPP**, sui soli oggetti tolti, e può farlo in due modi:
  - con FFD,
  - oppure in modo **esatto**, con un branch-and-bound che prova ogni contenitore per ogni oggetto e conserva l'impacchettamento con il minor numero di contenitori.

- Il sottoproblema ha solo gli oggetti di q contenitori, quindi una riparazione esatta è sostenibile.

Algorithm: Large neighbourhood search per il BPP (1/2)

Input : pesi $w[1..n]$, capacità c, impacchettamento $bin[1..n]$ con nb contenitori, q, *exact*, *maxIter*

Output : un impacchettamento bin e il suo numero di contenitori nb

```
1 function LNS-BPP(w, c, bin, nb, q, exact, maxIter)
2 |   LB := ⌈∑_j w[j]/c⌉;  it := 0
3 |   while it < maxIter and nb > LB do
4 |   |   it := it + 1
5 |   |   empty := il contenitore meno carico e q − 1 altri
  |   |    contenitori a caso                    // distruzione
6 |   |   items := gli oggetti posti nei contenitori di empty
7 |   |   newbin := gli altri contenitori, rinumerati
  |   |    1, . . . , nb − q
```

Algorithm: Large neighbourhood search per il BPP (2/2)

function *LNS-BPP*$(w, c, bin, nb, q, exact, maxIter)$ (continua)
  while $it < maxIter$ ***and*** $nb >$ LB (continua) **do**
8 **if** *exact* **then** impacchettare *items* in modo ottimo in k contenitori nuovi **else** impacchettare *items* con FFD in k contenitori nuovi `// riparazione`
9 $nb' := nb - q + k$
10 **if** $nb' < nb$ ***or*** $(nb' = nb$ ***and*** $\sum_b (\text{carico di } b \text{ in } newbin)^2 \geq \sum_b (\text{carico di } b \text{ in } bin)^2)$ **then**
11 $bin := newbin$; $nb := nb'$ `// accettazione`
12 **end**
13 **end**
14 **return** *bin, nb*
15 **end**

Esempio: *LNS* sull'istanza di riferimento del BPP (1/3)

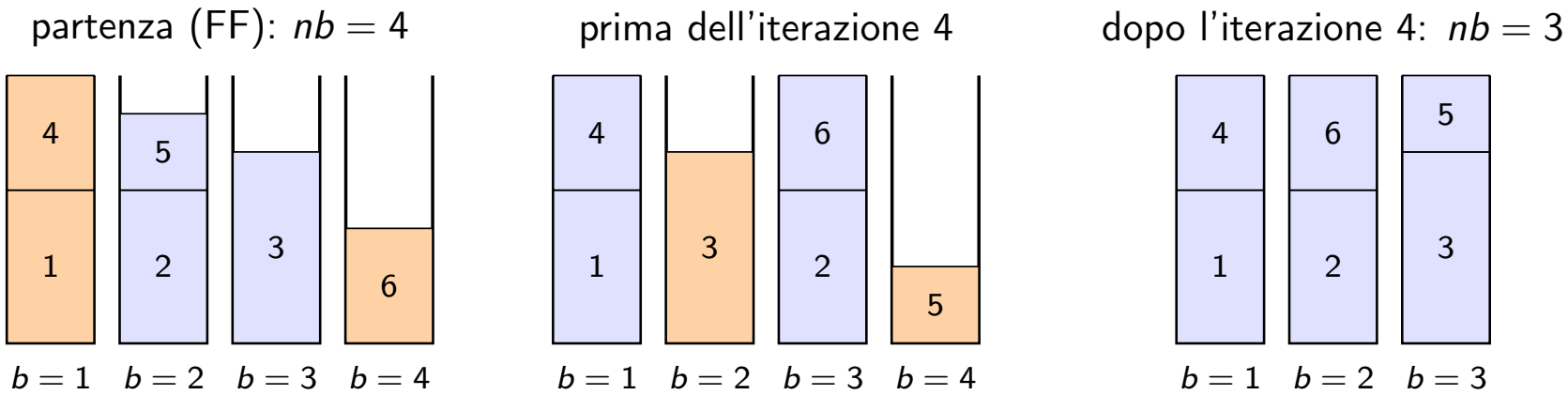
- Partiamo dall'impacchettamento calcolato da First Fit, $P_1 = \{1, 4\}$, $P_2 = \{2, 5\}$, $P_3 = \{3\}$, $P_4 = \{6\}$, con carichi $(7, 6, 5, 3)$.
- Usiamo $q = 2$, riparazione con FFD e $seed = 3$ (numeri casuali 0.0000, 0.3946, 0.2668, 0.3760).
- A ogni iterazione si svuota il contenitore meno carico, insieme al contenitore in posizione $1 + \lfloor 3u \rfloor$ fra gli altri tre.

Esempio: *LNS* sull'istanza di riferimento del BPP (2/3)

it	carichi	svuotati	oggetti	FFD	nb'	$\sum_b(\cdot)^2$
1	$(7,6,5,3)$	P_4, P_1	$\{1,4,6\}$	$\{1,4\},\{6\}$	4	$119 \to 119$
2	$(6,5,7,3)$	P_4, P_2	$\{3,6\}$	$\{3\},\{6\}$	4	$119 \to 119$
3	$(6,7,5,3)$	P_4, P_1	$\{2,5,6\}$	$\{2,6\},\{5\}$	4	$119 \to 127$
4	$(7,5,7,2)$	P_4, P_2	$\{3,5\}$	$\{3,5\}$	3	$127 \to 147$

- Le iterazioni 1 e 2 si limitano a rimescolare gli oggetti e mantengono gli stessi carichi (l'obiettivo ausiliario resta 119, e il nuovo impacchettamento è accettato perché non è peggiore).
- L'iterazione 3 costruisce il contenitore pieno $\{2,6\}$, portando l'obiettivo ausiliario a 127.
- L'iterazione 4 svuota i contenitori $\{5\}$ e $\{3\}$, e FFD mette insieme gli oggetti 3 e 5: 3 contenitori, l'ottimo, e la ricerca si ferma poiché $nb = \mathrm{LB}$.

Esempio: *LNS* sull'istanza di riferimento del BPP (3/3)



- Gli impacchettamenti alla partenza (in arancione i contenitori svuotati all'iterazione 1), prima dell'iterazione 4 (in arancione i contenitori che svuota) e dopo l'iterazione 4.

LNS su BPP24: riparazione euristica o esatta

- Su BPP24, partendo dall'impacchettamento FFD (10 contenitori), con 200 iterazioni e $seed = 1, \ldots, 10$, il risultato dipende dalla riparazione:

q	riparazione	contenitori alla fine (10 semi)	iterazioni per arrivare a 9
2	FFD	10, 10, 10, 10, 10, 10, 10, 10, 10, 10	mai
2	esatta	10, 10, 10, 10, 10, 10, 10, 10, 10, 10	mai
3	FFD	10, 10, 10, 10, 10, 10, 10, 10, 10, 10	mai
3	esatta	9, 9, 9, 9, 9, 9, 9, 9, 9, 9	4, 4, 12, 38, 3, 20, 5, 2, 7, 5
4	FFD	10, 10, 10, 10, 10, 10, 10, 10, 10, 10	mai
4	esatta	9, 9, 9, 9, 9, 9, 9, 9, 9, 9	2, 3, 9, 12, 4, 5, 1, 3, 10, 2

La forza della riparazione

- Con la riparazione FFD, la *LNS* non migliora **mai** l'impacchettamento FFD: gli oggetti tolti da un impacchettamento FFD vengono reimpacchettati da FFD nello stesso modo.
- Con la riparazione esatta e $q \geq 3$ raggiunge l'ottimo in ogni esecuzione, in poche iterazioni: trova tre contenitori i cui oggetti entrano in due, ed è ciò che FFD non vede.
- Con $q = 2$ anche la riparazione esatta fallisce, perché gli oggetti di due contenitori di un impacchettamento FFD non entrano mai in uno solo.

Un intorno grande si esplora risolvendo un **sottoproblema**: la parte della soluzione lasciata libera dal metodo di distruzione viene riottimizzata, mentre il resto è fissato.

Più potente è il metodo che risolve il sottoproblema, più potente è l'intorno.

Adaptive large neighbourhood search

- Quale metodo di distruzione funzioni meglio dipende dall'istanza, e dalla fase della ricerca.
- La LNS **adattiva** (ALNS), introdotta da Ropke e Pisinger (2006), usa più metodi di distruzione (e di riparazione), ne sceglie uno a caso a ogni iterazione, e **impara** quali funzionano, assegnando a ogni metodo o un peso w_o:

$$P(\text{scegliere } o) = \frac{w_o}{\sum_{o'} w_{o'}}, \qquad w_o \leftarrow (1-\rho)\, w_o + \rho\, \psi,$$

- $\rho \in [0, 1]$ è il *fattore di reazione*;
- il punteggio ψ premia l'esito dell'iterazione in cui o è stato usato: $\psi = 3$ se la nuova soluzione è la migliore trovata finora, 2 se è migliore di quella corrente, 1 se è altrettanto buona, 0 se è peggiore.

Algorithm: Adaptive large neighbourhood search per il TSP (1/2)

Input : distanze d, giro di partenza σ, q, *maxIter*, fattore di reazione ρ

Output : migliore giro trovato

```
1 function ALNS(d, σ, q, maxIter, ρ)
2     cur := σ;  best := σ;  w1 := 1;  w2 := 1
3     for it := 1 to maxIter do
4         if RAND() · (w1 + w2) < w1 then o := 1 else o := 2
            // roulette wheel
5         nw := cur
6         if o = 1 then removed := RANDOMREMOVAL(nw, q)
           else removed := WORSTREMOVAL(nw, q)
7         CHEAPESTINSERTIONREPAIR(nw, removed)
```

Algorithm: Adaptive large neighbourhood search per il TSP (2/2)

function *ALNS*$(d, \sigma, q, maxIter, \rho)$ (continua)

    for $it := 1$ **to** *maxIter* (continua) **do**

8        $\psi := 3$, 2, 1 o 0 se $L(nw) < L(best)$, $L(nw) < L(cur)$, $L(nw) = L(cur)$, $L(nw) > L(cur)$

9   **if** $L(nw) \leq L(cur)$ **then** $cur := nw$

10   **if** $L(cur) < L(best)$ **then** $best := cur$

11       $w_o := (1 - \rho)\, w_o + \rho\, \psi$ `// pesi adattivi`

12   **end**

13   **return** *best*

14 **end**

Che cosa impara la ALNS

- WorstRemoval toglie, q volte, la città la cui rimozione accorcia di più il giro: la città v, fra a e b, di massimo $d_{av} + d_{vb} - d_{ab}$. È un metodo di distruzione deterministico e greedy: toglie le città che sembrano mal posizionate.
- Su TSP20, dal giro nearest neighbour, con $q = 5$, 200 iterazioni e $\rho = 0.1$, la ALNS raggiunge l'ottimo in 7 esecuzioni su 10 (media 299.4).
- Alla fine il peso della rimozione casuale è fra 0.77 e 0.98, il peso della worst removal fra 0.03 e 0.08: il metodo ha imparato che, da un giro buono, togliere le città peggio posizionate e reinserirle con cheapest insertion ricostruisce quasi sempre un giro non migliore, e ha quasi smesso di usarla.
- In queste esecuzioni la *LNS* semplice con rimozione casuale ($q = 5$) fa leggermente meglio: l'adattività ripaga quando ci sono molti metodi, di qualità ignota, ed esecuzioni lunghe.

Riepilogo (1/2)

Large neighbourhood search:

- distruggere una parte della soluzione, ripararla, accettare se migliore (o non peggiore)
- l'intorno è grande ed è esplorato implicitamente dal metodo di riparazione
- la distruzione diversifica (q), la riparazione intensifica

Sulle istanze di riferimento:

- TSP: un passo di distruzione e riparazione con $q = 2$ raggiunge il giro ottimo
- BPP: svuotando il contenitore meno carico e reimpacchettando con FFD si arriva a 3 contenitori in quattro iterazioni

Riepilogo (2/2)

La riparazione:

- su BPP24 la riparazione FFD non migliora mai l'impacchettamento FFD
- la riparazione esatta con $q \geq 3$ raggiunge l'ottimo 9 in ogni esecuzione
- più potente è il metodo che risolve il sottoproblema, più potente è l'intorno

LNS adattiva:

- sceglie il metodo di distruzione con la roulette wheel, con pesi aggiornati dai punteggi ψ
- ripaga con molti metodi di qualità ignota ed esecuzioni lunghe

Metodi basati su popolazioni: algoritmi genetici e memetici

Introduzione: Metodi basati su popolazioni

- Gli algoritmi genetici e memetici mantengono una *popolazione* di soluzioni, e combinano le parti di soluzioni buone in soluzioni nuove
- Vediamo un algoritmo genetico steady-state per il KP, le sue componenti (codifica, selezione, crossover, mutazione, riparazione, sostituzione), e la variante memetica con una ricerca locale
- Poi eseguiamo a mano tre iterazioni sul KP, e confrontiamo le due varianti su KP20

L'idea degli algoritmi genetici e memetici

- Un *algoritmo genetico* (indicato con *GA*) mantiene una *popolazione* di soluzioni, codificate come stringhe (*cromosomi*). A ogni iterazione
  - *seleziona* due genitori, preferendo i più adatti,
  - li *ricombina* in un figlio (*crossover*),
  - cambia alcuni geni casuali del figlio (*mutazione*),
  - lo *ripara* se è inammissibile,
  - e lo inserisce nella popolazione al posto di un individuo peggiore.

- Un *algoritmo memetico* applica inoltre una ricerca locale a ogni figlio.

Vediamo ora lo pseudocodice di un *GA* steady-state per il KP con codifica binaria;
con $memetic = \mathrm{true}$ è un algoritmo memetico.

Algorithm: Algoritmo genetico e memetico per il KP (1/4)

Input : istanza del KP $p[1..n]$, $w[1..n]$, c; dimensione della popolazione P; probabilità di mutazione p_m; *maxIter*; *memetic*

Output : miglior individuo della popolazione finale

```
1 function GeneticKP(p, w, c, P, p_m, maxIter, memetic)
2     for s := 1 to P do // popolazione iniziale casuale
3         for j := 1 to n do pop[s][j] := 1 se RAND() < 0.5,
           altrimenti 0
4         REPAIRKP(pop[s]); if memetic then
           KPLOCALSEARCH(pop[s])
5         fit[s] := Σ_j p[j] pop[s][j]
6     end
```

Algorithm: Algoritmo genetico e memetico per il KP (2/4)

function *GeneticKP*$(p, w, c, P, p_m, maxIter, memetic)$

(continua)

```
7   |   for it := 1 to maxIter do
8   |   |   a := TOURNAMENT(); b := TOURNAMENT()
    |   |    // selezione
9   |   |   cut := 1 + ⌊RAND() · (n − 1)⌋   // crossover a un
    |   |    punto
10  |   |   child := (pop[a][1..cut], pop[b][cut+1..n])
11  |   |   for j := 1 to n do if RAND() < p_m then
    |   |    child[j] := 1 − child[j]
12  |   |                                        // mutazione
```

Algorithm: Algoritmo genetico e memetico per il KP (3/4)

function *GeneticKP*$(p, w, c, P, p_m, maxIter, memetic)$
(continua)

```
    for it := 1 to maxIter (continua) do
12      RepairKP(child); if memetic then
          KPLocalSearch(child)
13      worst := l'ultimo indice s di minimo fit[s]
14      if child ∉ pop and Σ_j p[j] child[j] > fit[worst] then
          // sostituzione
15        pop[worst] := child; fit[worst] := Σ_j p[j] child[j]
16      end
17  end
18  return l'individuo di massimo fit
19 end
```

Algorithm: Algoritmo genetico e memetico per il KP (4/4)

```
20 function Tournament()
21 |   a := 1 + ⌊RAND() · P⌋;  b := 1 + ⌊RAND() · P⌋
22 |   if fit[a] ≥ fit[b] then return a else return b
23 end
```

Codifica, selezione, crossover

- Gli algoritmi genetici sono stati introdotti da Holland (1975) come modello dell'evoluzione naturale, e resi popolari come metodi di ottimizzazione da Goldberg (1989). Gli algoritmi memetici, in cui ogni individuo è migliorato da una ricerca locale, sono stati proposti da Moscato (1989).
- **Codifica.** Per il KP il vettore binario x è un cromosoma naturale: il gene j dice se l'oggetto j è selezionato. La **fitness** di un individuo è il suo profitto.
- **Selezione.** Un torneo binario estrae due individui a caso e tiene il più adatto: gli individui buoni hanno più probabilità di diventare genitori, ma ogni individuo ha una possibilità.
- **Crossover.** Il crossover a un punto prende i geni $1, \ldots, cut$ dal primo genitore e i geni $cut + 1, \ldots, n$ dal secondo. L'idea è che le soluzioni buone condividano dei *blocchi costitutivi* buoni, e che ricombinare due genitori buoni possa mettere i blocchi buoni di entrambi in un solo figlio.

Mutazione, riparazione, sostituzione

- **Mutazione.** Ogni gene viene invertito con una probabilità piccola p_m, tipicamente $\frac{1}{n}$: riporta nella popolazione geni che la selezione può aver perso.
- **Riparazione.** Il figlio può essere inammissibile. RepairKP toglie l'oggetto selezionato di efficienza minima finché la capacità non è rispettata (pareggi: l'oggetto di indice massimo). Le alternative sono le penalità, come nel simulated annealing con penalità, o codifiche in cui ogni cromosoma è ammissibile.
- **Sostituzione.** Nella versione steady-state dello pseudocodice, il figlio sostituisce l'individuo peggiore se è migliore e non è già nella popolazione; i duplicati farebbero perdere alla popolazione la sua diversità.

Intensificazione e diversificazione in un *GA*

- Selezione e sostituzione **intensificano** (la popolazione si concentra sulle soluzioni buone); crossover e mutazione **diversificano**. Senza ricerca locale, un *GA* è bravo a trovare regioni promettenti ma lento a raggiungerne gli ottimi locali: l'algoritmo memetico aggiunge l'intensificazione mancante.

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

- Con $P = 4$, $p_m = \frac{1}{4}$ e $seed = 36$. I primi 16 numeri casuali costruiscono la popolazione iniziale (gene 1 se il numero è minore di 0.5):

$$
\begin{array}{lllll}
pop[1]: & 0.0003, & 0.7354, & 0.2018, & 0.5114 \quad \rightarrow (1,0,1,0) \\
pop[2]: & 0.1796, & 0.8825, & 0.6936, & 0.4391 \quad \rightarrow (1,0,0,1) \\
pop[3]: & 0.4547, & 0.6489, & 0.8061, & 0.6990 \quad \rightarrow (1,0,0,0) \\
pop[4]: & 0.9148, & 0.2446, & 0.9246, & 0.0692 \quad \rightarrow (0,1,0,1)
\end{array}
$$

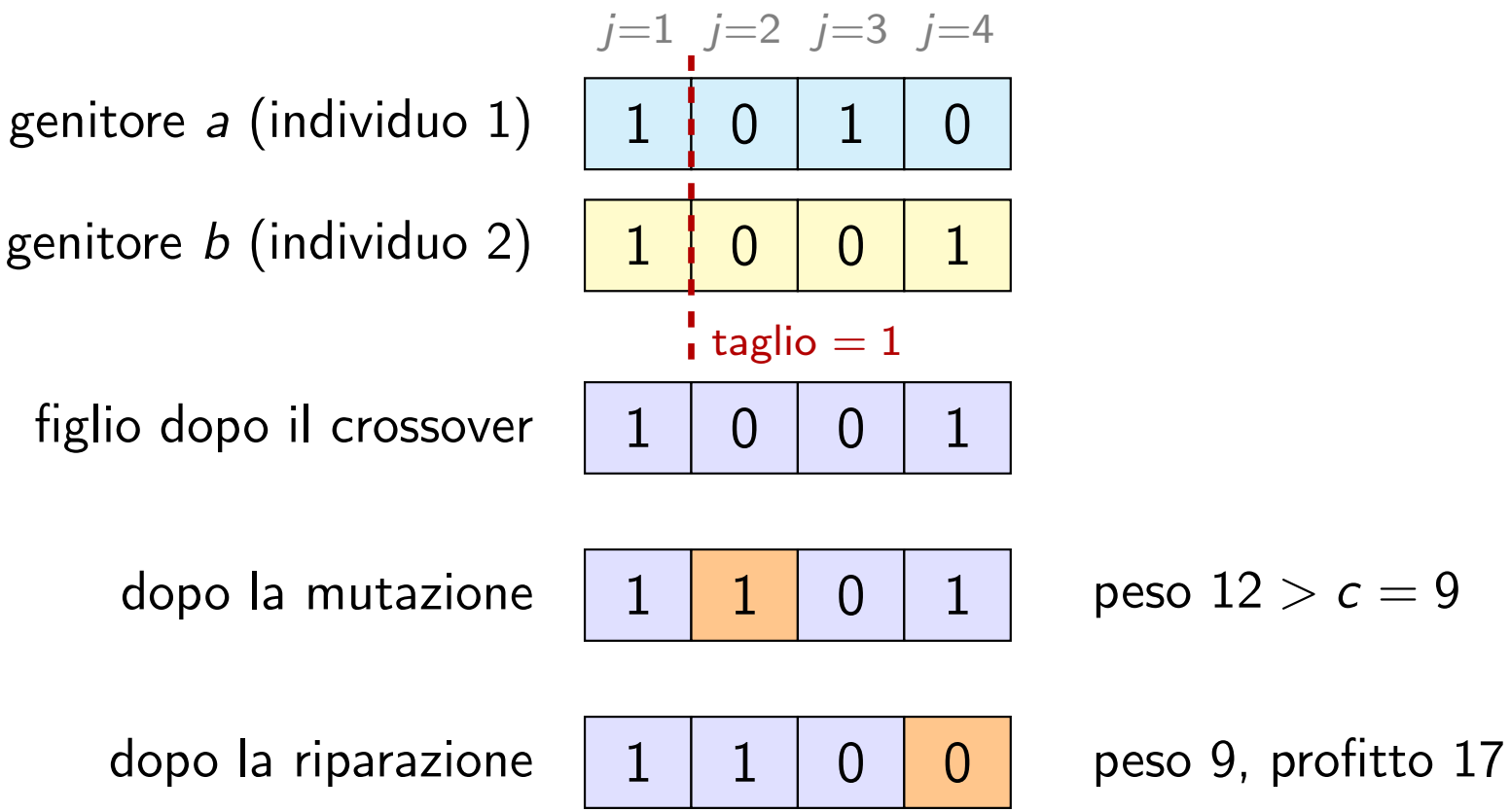
- Tutti e quattro gli individui sono ammissibili: $pop[1] = (1,0,1,0)$, $pop[2] = (1,0,0,1)$, $pop[3] = (1,0,0,0)$, $pop[4] = (0,1,0,1)$, con fitness $(16, 14, 10, 11)$.

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

Iterazione 1 (numeri casuali 0.1614, 0.2771, 0.8030, 0.4063, 0.0295, 0.7238, 0.2032, 0.4957, 0.4620).

- Il primo torneo estrae gli individui $1 + \lfloor 4 \cdot 0.1614 \rfloor = 1$ e $1 + \lfloor 4 \cdot 0.2771 \rfloor = 2$, e seleziona $a = 1$ (fitness $16 > 14$).
- Il secondo estrae 4 e 2 e seleziona $b = 2$ (fitness $14 > 11$).
- Il taglio è $1 + \lfloor 3 \cdot 0.0295 \rfloor = 1$: il figlio prende il gene 1 da $pop[1]$ e i geni 2, 3, 4 da $pop[2]$, $child = (1, 0, 0, 1)$.
- Solo il secondo numero di mutazione, 0.2032, è minore di $p_m = 0.25$: il gene 2 si inverte, $child = (1, 1, 0, 1)$, di peso $12 > 9$.
- La riparazione toglie l'oggetto 4, l'oggetto selezionato di efficienza minima: $child = (1, 1, 0, 0)$, di peso 9 e profitto 17. Sostituisce l'individuo peggiore, $pop[3]$ di fitness 10.

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



- Iterazione 1: crossover, mutazione del gene 2, e riparazione (oggetto 4 tolto).

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

- L'**iterazione 2** seleziona $a = 2$ e $b = 3$, taglia dopo il gene 1 e costruisce $(1, 1, 0, 0)$, senza mutazione: è già nella popolazione, e viene scartato.
- L'**iterazione 3** seleziona $a = 2$ e $b = 1$, taglia dopo il gene 3, costruisce $(1, 0, 0, 0)$, e la mutazione inverte i geni 1 e 4: $(0, 0, 0, 1)$, di profitto 4, peggiore di ogni individuo, scartato.

Il miglior individuo è $(1, 1, 0, 0)$: $GA(I) = 17 = z(I)$, trovato da crossover, mutazione e riparazione insieme.

GA e algoritmo memetico su KP20

- Su KP20, con $P = 10$, $p_m = 0.05$ e 300 iterazioni, per $seed = 1, \ldots, 10$:

	min	media	max	ottime
GA	508	516.9	521	4
memetico	521	521.0	521	10

	profitto migliore per seme
GA	521, 519, 521, 508, 514, 521, 510, 519, 521, 515
memetico	521, 521, 521, 521, 521, 521, 521, 521, 521, 521

- Il *GA* semplice raggiunge l'ottimo 521 in 4 esecuzioni su 10; l'algoritmo memetico in ogni esecuzione.
- La ricerca locale rende ogni individuo un ottimo locale, e il crossover ricombina allora ottimi locali invece di soluzioni arbitrarie.

Riepilogo

Algoritmo genetico:

- una popolazione di cromosomi; selezione con torneo, crossover a un punto, mutazione, riparazione, sostituzione dell'individuo peggiore
- selezione e sostituzione intensificano, crossover e mutazione diversificano

Algoritmo memetico:

- una ricerca locale su ogni figlio: ricombina ottimi locali
- su KP20 raggiunge l'ottimo in ogni esecuzione, il *GA* semplice in 4 esecuzioni su 10

Confronto e conclusioni

Introduzione: Confronto e conclusioni

- Raccogliamo i risultati di tutti gli esperimenti del capitolo sulle istanze medie
- Riassumiamo quale componente di ogni metodo intensifica e quale diversifica, e come si calibrano i parametri
- Infine vediamo che cosa questi metodi non usano, e che cosa non possono dare: un bound duale

I metodi sulle istanze medie (1/3)

- Per ogni metodo, il miglior valore trovato in 10 esecuzioni ($seed = 1, \ldots, 10$), come minimo, media e massimo, e il numero di esecuzioni che raggiungono l'ottimo. I metodi deterministici danno lo stesso valore in ogni esecuzione.

istanza	metodo	parametri	min	media	max	ottime
TSP20	NN	—	326	326	326	0
	NN + 2-opt	—	319	319	319	0
	SA	$T_0 = 20$, $\alpha = 0.95$	291	294.3	298	5
	SA	$T_0 = 20$, $\alpha = 0.99$	291	291.0	291	10
	TS	*tenure* $= 5$, 200 it.	291	291	291	10
	ILS	100 it.	291	291.0	291	10
	GRASP	$\alpha = 0.25$, 20 it.	291	291.0	291	10
	LNS	$q = 5$, 200 it.	291	293.8	319	9
	ALNS	$q = 5$, 200 it.	291	299.4	319	7

I metodi sulle istanze medie (2/3)

istanza	metodo	parametri	min	media	max	ottime
KP20	greedy	—	490	490	490	0
	greedy + ricerca locale	—	514	514	514	0
	GRASP	$\alpha = 0.25$, 20 it.	521	521.0	521	10
	GA	$P = 10$, 300 it.	508	516.9	521	4
	memetico	$P = 10$, 300 it.	521	521.0	521	10

I metodi sulle istanze medie (3/3)

istanza	metodo	parametri	min	media	max	ottime
PCM15	LPT	—	73	73	73	0
	TS	*tenure* $= 15$, 300 it.	69	69	69	10
	VNS	$k_{\max} = 3$, 300 it.	69	69.3	70	7
BPP24	FFD	—	10	10	10	0
	LNS, riparazione FFD	$q = 3$, 200 it.	10	10.0	10	0
	LNS, riparazione esatta	$q = 3$, 200 it.	9	9.0	9	10

Che cosa mostra il confronto

- Ogni metaeuristica migliora la ricerca locale su cui è costruita, e con parametri adatti diverse di esse raggiungono l'ottimo in ogni esecuzione.

- La tabella mostra però anche quanto il risultato dipenda
  - dai parametri (la durata tabu di *TS*, l'intensità q della distruzione, il raffreddamento di *SA*)
  - e dalle componenti (la riparazione di *LNS*, la ricerca locale dell'algoritmo memetico).

- Su quattro istanze piccole non stabilisce una classifica dei metodi: una classifica del genere richiede molte istanze e tempi di calcolo uguali, come discusso nel capitolo «Algoritmi euristici: definizioni».

Intensificazione e diversificazione (1/2)

- Ogni metaeuristica di questo capitolo combina una componente che **intensifica** (cerca intorno alle soluzioni buone già trovate) e una componente che **diversifica** (sposta la ricerca in regioni inesplorate), e ha parametri che fissano l'equilibrio fra le due.

metodo	intensificazione	diversificazione	equilibrio fissato da
SA	accettare solo mosse miglioranti a bassa temperatura	accettare mosse peggioranti ad alta temperatura	T_0, fattore di raffreddamento α, L
TS	spostarsi sul miglior vicino; ripartire da soluzioni d'élite	lista tabu; penalità sugli attributi frequenti	durata tabu, criterio di aspirazione

Intensificazione e diversificazione (2/2)

metodo	intensificazione	diversificazione	equilibrio fissato da
ILS	ricerca locale dopo ogni kick; accettare solo ottimi locali migliori	il kick	intensità del kick, criterio di accettazione
VNS	ricerca locale; ritorno a $k = 1$ dopo un miglioramento	shaking in N_k con k crescente	$k_{\max}$, gli intorni N_k
GRASP	ricerca locale su ogni soluzione costruita	costruzione randomizzata	α
LNS / ALNS	riparazione (euristica o esatta), accettazione	distruzione casuale	q, scelta dei metodi, pesi
GA / memetico	selezione, sostituzione, ricerca locale (memetico)	crossover, mutazione	dimensione della popolazione, p_m

Parametri e calibrazione

- Una metaeuristica è un algoritmo solo dopo che i suoi parametri sono stati fissati, e gli esperimenti di questo capitolo mostrano che i parametri possono fare la differenza fra trovare l'ottimo in ogni esecuzione e non trovarlo mai. Sceglierli si chiama **calibrazione** (tuning).

- I parametri vanno calibrati su istanze di **addestramento** e valutati su istanze *diverse*, altrimenti il quadro è ottimistico.

- Calibrare un parametro alla volta ignora le loro interazioni (per *SA*, il T_0 migliore dipende da α); strumenti come `irace` (López-Ibáñez et al., 2016) li calibrano tutti insieme.

- Un metodo che funziona bene per un ampio intervallo di parametri è preferibile a uno che funziona molto bene solo per un intervallo ristretto.

Che cosa questi metodi non usano (1/2)

- Nessuna delle metaeuristiche di questo capitolo usa un **rilassamento** del problema o un **solver MIP**. Manipolano solo soluzioni ammissibili (o penalizzate), e producono solo bound *primali*.

- Di conseguenza, non possono dire quanto siano buone le loro soluzioni. Sulle istanze medie abbiamo potuto misurare la qualità solo perché i valori ottimi erano stati calcolati a parte, con algoritmi esatti.

Che cosa questi metodi non usano (2/2)

- Le due eccezioni confermano la regola: per $P \,||\, C_{\max}$ e il BPP, *TS*, *VNS* e *LNS* si sono fermati appena hanno raggiunto un lower bound semplice, che è un bound *duale* che certifica l'ottimalità.
- Quando l'ottimo è sopra il bound, un certificato del genere non esiste, e la ricerca prosegue fino al limite di iterazioni.
- La riparazione esatta di *LNS* per il BPP, infine, ha mostrato che riottimizzare in modo esatto una parte della soluzione, con il resto fissato, dà intorni molto potenti.

Le metaeuristiche escono dagli ottimi locali, ma danno solo un lato del gap: il bound primale.

Riepilogo

Confronto:

- ogni metaeuristica migliora la sua ricerca locale; diverse raggiungono l'ottimo in ogni esecuzione
- i risultati dipendono fortemente dai parametri e dalle componenti
- una classifica richiede molte istanze e tempi di calcolo uguali

Calibrazione e limiti:

- calibrare su istanze di addestramento, valutare su istanze diverse
- niente rilassamenti, niente solver MIP: solo bound primali, nessun certificato di qualità

Errori tipici

Introduzione: Errori tipici

- Raccogliamo gli errori che si commettono più spesso quando si esegue e si valuta una metaeuristica
- Due riguardano gli esperimenti, due gli algoritmi, uno l'esecuzione a mano
- Conviene controllarli prima di riportare un risultato

Errori tipici (1/3)

Osservazione 1: Riportare una sola esecuzione

I risultati dipendono dal seme: riportare minimo, media e massimo su più esecuzioni.

Osservazione 2: Calibrare e valutare sulle stesse istanze

Il quadro è ottimistico; usare istanze di addestramento per i parametri e istanze diverse per la valutazione.

Errori tipici (2/3)

Osservazione 3: Perdere la soluzione migliore

Nel simulated annealing e nella tabu search la soluzione corrente può essere peggiore della migliore trovata: la migliore va memorizzata a parte.

Osservazione 4: Scegliere la durata tabu senza provarla

Con durata 0 la tabu search è una ricerca locale; una durata troppo piccola cicla, una troppo grande vieta quasi ogni mossa.

Errori tipici (3/3)

Osservazione 5: Contare male i numeri casuali

Ogni iterazione ne consuma un numero fisso (dieci nell'algoritmo genetico degli esercizi, con $n = 5$); un conteggio sbagliato cambia tutti i passi successivi.

Riepilogo

Esperimenti:

- riportare minimo, media e massimo su più semi
- calibrare e valutare su istanze diverse

Algoritmi ed esecuzione a mano:

- memorizzare la soluzione migliore separatamente da quella corrente
- provare la durata tabu della tabu search
- contare i numeri casuali consumati da ogni iterazione

Implementazione in C++

Introduzione: Implementazione in C++

- Ogni algoritmo del capitolo è implementato in C++, con la stessa logica degli pseudocodici
- Vediamo come è organizzato il codice, come si verificano gli esempi, e tre brevi estratti: il generatore di numeri casuali, il cuore del simulated annealing e il kick double bridge
- Il codice completo è nelle note del capitolo

L'organizzazione del codice

- Tutte le funzioni sono nella cartella `METAHEUR_CODE/src`, una funzione per file, e sono dichiarate nell'header `Metaheuristics.h`.
- Gli oggetti, le città e i lavori hanno indici $0, 1, \ldots, n-1$, le macchine $0, \ldots, m-1$ e i contenitori $0, \ldots, nb-1$: la logica algoritmica è quella degli pseudocodici, con indici a partire da 0.
- I vettori sono `Vec` (`vector<int>`) e le matrici `Matrix` (`vector<Vec>`).
- Il comando `make test` compila il codice ed esegue ogni esempio di questo capitolo e ogni istanza degli esercizi, confrontando il valore trovato con quello atteso; `make data` scrive i dati della figura dell'esperimento di *SA* su TSP20.

Il generatore di numeri casuali

- I numeri casuali sono prodotti da `Rand`, il minimal standard generator. La struttura `Random` contiene il seme e, facoltativamente, una lista di numeri assegnati, che vengono restituiti al suo posto: gli esercizi danno esplicitamente i loro numeri casuali.

```
1  double Rand(Random& r)
2  {
3      double u;
4      if (r.given.size() > 0)
5      {
6          u = r.given[r.used];
7      }
8      else
9      {
10         r.seed = (16807 * r.seed) % 2147483647;
11         u = (double) r.seed / 2147483647.0;
12     }
13     r.used = r.used + 1;
14     return u;
15 }
```

Il cuore del simulated annealing per il TSP

```
while (T > Tmin)
{
    for (int l = 0; l < L; ++l)  // L moves at temperature T
    {
        int k = (int) (Rand(r) * M);  // random neighbour
        int delta = TwoOptDelta(cur, d, mi[k], mj[k]);
        double v = Rand(r);
        if (delta <= 0 || v < exp(-delta / T))  // Metropolis acceptance rule
        {
            ApplyTwoOpt(cur, mi[k], mj[k]);
            fcur = fcur + delta;
            if (fcur < fbest) { best = cur; fbest = fcur; }
        }
    }
    T = alpha * T;  // cooling
}
```

Il kick double bridge

```
1  Vec DoubleBridge(const Vec& pi, Random& r)
2  {
3      int n = pi.size();
4      int p1 = 1 + (int) (Rand(r) * (n - 3));
5      int p2 = p1 + 1 + (int) (Rand(r) * (n - 2 - p1));
6      int p3 = p2 + 1 + (int) (Rand(r) * (n - 1 - p2));
7      Vec kick;
8      for (int k = 0; k < p1; ++k) kick.push_back(pi[k]);  // A
9      for (int k = p2; k < p3; ++k) kick.push_back(pi[k]);  // C
10     for (int k = p1; k < p2; ++k) kick.push_back(pi[k]);  // B
11     for (int k = p3; k < n; ++k) kick.push_back(pi[k]);  // D
12     return kick;
13 }
```

I file del capitolo (1/2)

che cosa	file in `METAHEUR_CODE/src`
numeri casuali	`Rand`
ricerca locale 2-opt per il TSP	`TourLength`, `TwoOptDelta`, `ApplyTwoOpt`, `TwoOptLS`, `NearestNeighbour`
discesa move per $P \,\|\|\, C_{\max}$	`Loads`, `LexLess`, `MoveDescent`, `LPT`
ricerca locale per il KP	`Profit`, `Weight`, `KPLocalSearch`
simulated annealing	`SimulatedAnnealingTSP`, `SimulatedAnnealingKP`
tabu search	`TabuSearchPCmax`, `TabuSearchTSP`

I file del capitolo (2/2)

che cosa	file in `METAHEUR_CODE/src`
iterated local search	`DoubleBridge`, `IteratedLocalSearchTSP`
variable neighbourhood search	`Shake`, `VNSPCmax`
GRASP	`RandomizedGreedyKP`, `GraspKP`, `RandomizedNN`, `GraspTSP`
LNS e ALNS per il TSP	`RandomRemoval`, `WorstRemoval`, `CheapestInsertionRepair`, `LnsTSP`, `AlnsTSP`
LNS per il BPP	`FirstFit`, `DestroyBins`, `RepairFFD`, `RepairExact`, `LnsBPP`
algoritmo genetico e memetico	`RepairKP`, `Tournament`, `GeneticKP`

Riepilogo

Il codice:

- una funzione per file in `METAHEUR_CODE/src`, dichiarata in `Metaheuristics.h`
- la logica degli pseudocodici, con indici a partire da 0

Verifiche:

- `make test` esegue ogni esempio e ogni esercizio e lo confronta con il valore atteso
- `Rand` può restituire numeri assegnati, come negli esercizi

Bibliografia (1/3)

- V. Černý, Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm, *Journal of Optimization Theory and Applications* 45 (1985) 41–51.
- T.A. Feo, M.G.C. Resende, A probabilistic heuristic for a computationally difficult set covering problem, *Operations Research Letters* 8 (1989) 67–71.
- T.A. Feo, M.G.C. Resende, Greedy randomized adaptive search procedures, *Journal of Global Optimization* 6 (1995) 109–133.
- F. Glover, Future paths for integer programming and links to artificial intelligence, *Computers & Operations Research* 13 (1986) 533–549.
- F. Glover, M. Laguna, *Tabu Search*, Kluwer, 1997.
- D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.
- B. Hajek, Cooling schedules for optimal annealing, *Mathematics of Operations Research* 13 (1988) 311–329.

Bibliografia (2/3)

- J.H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.
- S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, *Science* 220 (1983) 671–680.
- M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, M. Birattari, T. Stützle, The irace package: iterated racing for automatic algorithm configuration, *Operations Research Perspectives* 3 (2016) 43–58.
- H.R. Lourenço, O.C. Martin, T. Stützle, Iterated local search, in F. Glover, G.A. Kochenberger (eds.), *Handbook of Metaheuristics*, Kluwer, 2003, 321–353.
- O. Martin, S.W. Otto, E.W. Felten, Large-step Markov chains for the traveling salesman problem, *Complex Systems* 5 (1991) 299–326.
- N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of state calculations by fast computing machines, *The Journal of Chemical Physics* 21 (1953) 1087–1092.

Bibliografia (3/3)

- N. Mladenović, P. Hansen, Variable neighborhood search, *Computers & Operations Research* 24 (1997) 1097–1100.
- P. Moscato, On evolution, search, optimization, genetic algorithms and martial arts: towards memetic algorithms, Technical Report C3P 826, Caltech, 1989.
- S.K. Park, K.W. Miller, Random number generators: good ones are hard to find, *Communications of the ACM* 31 (1988) 1192–1201.
- S. Ropke, D. Pisinger, An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows, *Transportation Science* 40 (2006) 455–472.
- P. Shaw, Using constraint programming and local search methods to solve vehicle routing problems, in *Principles and Practice of Constraint Programming – CP98*, Lecture Notes in Computer Science 1520, Springer, 1998, 417–431.
- É. Taillard, Robust taboo search for the quadratic assignment problem, *Parallel Computing* 17 (1991) 443–455.

In sintesi

Introduzione: In sintesi

- Raccogliamo le idee chiave di tutto il capitolo sulle metaeuristiche, entrambe le parti
- Il quadro riassuntivo confronta, sulle istanze medie, le euristiche costruttive, le ricerche locali, la migliore metaeuristica e l'ottimo

Riepilogo (1/2)

- Una metaeuristica esce dagli ottimi locali accettando soluzioni peggiori (SA, TS), perturbando (ILS, VNS), randomizzando la costruzione (GRASP), distruggendo e riparando (LNS) o ricombinando una popolazione (GA).
- Ogni metodo bilancia intensificazione e diversificazione con pochi parametri, e i risultati ne dipendono fortemente.
- Non usano rilassamenti: producono solo bound primali, e non possono dire quanto siano buone le loro soluzioni.

Riepilogo (2/2)

Quadro riassuntivo delle istanze medie. Miglior valore dei metodi a confronto con l'ottimo (l'ottimo calcolato da un algoritmo esatto):

istanza	costruttiva	ricerca locale	migliore metaeuristica	ottimo
TSP20 (min)	326 (*NN*)	319 (2-opt)	291	291
KP20 (max)	490 (greedy)	514	521	521
PCM15 (min)	73 (*LPT*)	—	69	69
BPP24 (min)	10 (FFD)	—	9	9