

SAPIENZA
UNIVERSITÀ DI ROMA

Algoritmi euristici

Dispensa

Algoritmi euristici per i problemi di ottimizzazione combinatoria: bound primali e duali e valutazione delle euristiche, euristiche costruttive, ricerca locale, metaeuristiche ed euristiche basate sulla programmazione lineare intera, applicate ai problemi di zaino, di packing, di routing, di scheduling e di colorazione.

Fabio Furini

fabiofurini.github.io

Dipartimento di Ingegneria informatica, automatica e gestionale — Sapienza Università di Roma

Versione online e materiale scaricabile:
fabiofurini.github.io/algoritmi-euristici/
github.com/fabiofurini/algoritmi-euristici

Anno accademico 2026–2027

Indice

1 Algoritmi euristici: definizioni **3**

1.1 Algoritmi euristici . . . 4
1.2 I problemi e le ipotesi . . . 4
1.3 Bound primali e duali . . . 6
1.4 Classificazione degli algoritmi euristici . . . 8
1.5 Rappresentazione delle soluzioni e spazio di ricerca . . . 9
1.6 Intorni e ottimi locali . . . 9
1.7 Intensificazione e diversificazione . . . 11
1.8 Come valutare un algoritmo euristico . . . 12
1.9 Errori tipici . . . 12
1.10 In sintesi . . . 13

2 Euristiche costruttive **14**

2.1 Euristiche costruttive . . . 15
2.2 I problemi e le ipotesi . . . 15
2.3 Il paradigma greedy . . . 16
2.4 Il problema dello zaino 0/1: quale ordinamento? . . . 18
2.5 Il problema del bin packing: prima ordinare, poi riempire . . . 20
2.6 Scheduling su macchine parallele: prima i lavori più lunghi . . . 25
2.7 Il problema del commesso viaggiatore: accodare o inserire? . . . 28
2.8 Colorazione dei vertici: greedy e DSATUR . . . 35
2.9 Costruzione randomizzata . . . 39
2.10 Look-ahead e regret . . . 41
2.11 Che cosa danno le euristiche costruttive, e che cosa non danno . . . 44
2.12 Errori tipici . . . 45
2.13 In sintesi . . . 46
2.14 Implementazione in `C++` . . . 46

3 Ricerca locale **58**

3.1 Ricerca locale . . . 58
3.2 I problemi, le ipotesi e le soluzioni di partenza . . . 59
3.3 Ricerca locale: lo schema . . . 60
3.4 Intorni per i quattro problemi . . . 62

3.5 First improvement e best improvement . . . 65
3.6 Valutare una mossa in modo incrementale . . . 68
3.7 TSP: 2-opt e Or-opt . . . 69
3.8 $P \,||\, C_{\max}$: move e swap . . . 75
3.9 Paesaggi piatti e obiettivi ausiliari . . . 78
3.10 KP: ammissibilità e penalità . . . 85
3.11 Variable neighbourhood descent . . . 89
3.12 Ricerca locale multi-start . . . 91
3.13 Errori tipici . . . 94
3.14 In sintesi . . . 94
3.15 Implementazione in `C++` . . . 95

4 Metaeuristiche 105
4.1 Metaeuristiche . . . 106
4.2 I problemi e le ipotesi . . . 106
4.3 Come uscire da un ottimo locale . . . 110
4.4 Simulated annealing . . . 112
4.5 Tabu search . . . 117
4.6 Iterated local search . . . 121
4.7 Variable neighbourhood search . . . 124
4.8 GRASP . . . 126
4.9 Large neighbourhood search . . . 129
4.10 Metodi basati su popolazioni: algoritmi genetici e memetici . . . 134
4.11 Confronto e conclusioni . . . 136
4.12 Errori tipici . . . 138
4.13 In sintesi . . . 139
4.14 Implementazione in `C++` . . . 139
Bibliografia . . . 159

5 Euristiche MIP per i problemi classici 161
5.1 Euristiche basate sulla PLI . . . 162
5.2 I problemi e i loro modelli MIP . . . 162
5.3 Arrotondamento . . . 165
5.4 Diving . . . 166
5.5 La feasibility pump . . . 168
5.6 RINS, local branching e proximity search . . . 170
5.7 Relax-and-fix e fix-and-optimize sul lot sizing . . . 173
5.8 Le euristiche dentro il solver . . . 175
5.9 Che cosa danno le euristiche basate sulla PLI . . . 176
5.10 Errori tipici . . . 177
5.11 In sintesi . . . 177
5.12 Implementazione in `Python` . . . 178

Capitolo 1

Algoritmi euristici: definizioni

Prerequisiti

- Dal modulo *Problemi di ottimizzazione combinatoria*, il problema dello zaino 0/1, il problema del bin packing, il problema del commesso viaggiatore e $P\,||\,C_{\max}$: istanze, soluzioni ammissibili e valori ottimi dei quattro problemi usati come esempi.
- Dal modulo *Programmazione lineare intera*, Capitolo 2 «Bound primali e duali»: il rilassamento lineare come fonte di bound duali. Qui ne diamo un breve richiamo per il problema dello zaino 0/1.

Obiettivi

Alla fine di questo capitolo saremo in grado di:

- definire un algoritmo euristico e calcolarne il gap rispetto a un bound duale;
- distinguere i bound primali dai bound duali sulle quattro istanze di riferimento;
- classificare gli algoritmi euristici e descrivere la rappresentazione delle soluzioni e lo spazio di ricerca;
- definire intorni e ottimi locali, e spiegare intensificazione e diversificazione;
- valutare un algoritmo euristico in base a qualità, tempo di esecuzione ed esecuzioni ripetute.

1.1 Algoritmi euristici

Definizione 1.1: Algoritmo euristico

Dato un **problema di ottimizzazione** $\mathscr{P}$, sia $z(I)$ il **valore ottimo** di $\mathscr{P}$ su un'**istanza** I. Un **algoritmo euristico** per $\mathscr{P}$ è un algoritmo che, per ogni istanza I di $\mathscr{P}$, calcola una soluzione ammissibile di I, senza garantire che sia ottima. Indichiamo con $\mathscr{A}(I)$ il valore della soluzione calcolata dall'algoritmo euristico $\mathscr{A}$ sull'istanza I.

- Poiché la soluzione calcolata da $\mathscr{A}$ è ammissibile, per ogni istanza I

$$\begin{cases} \mathscr{A}(I) \geq z(I) & \text{se } \mathscr{P} \text{ è un problema di minimo,} \\ \mathscr{A}(I) \leq z(I) & \text{se } \mathscr{P} \text{ è un problema di massimo.} \end{cases}$$

- I quattro problemi usati come esempi — il problema dello zaino 0/1, il problema del bin packing, il problema del commesso viaggiatore e il problema di scheduling $P\,||\,C_{\max}$ — sono tutti **NP-hard**: un algoritmo esatto può richiedere un tempo di esecuzione che cresce esponenzialmente con la dimensione dell'istanza.

- Gli **algoritmi di approssimazione** sono algoritmi euristici con una garanzia *dimostrata* nel caso peggiore sul rapporto fra $\mathscr{A}(I)$ e $z(I)$. La maggior parte degli algoritmi euristici usati in pratica non ha questa garanzia: la loro qualità si valuta sulle istanze da risolvere, come descritto nella Sezione 1.8.

Un algoritmo euristico scambia la **qualità** della soluzione con il **tempo di esecuzione**: lo scopo di un'euristica è calcolare, in poco tempo, una soluzione il cui valore sia vicino a $z(I)$.

1.2 I problemi e le ipotesi

- Gli algoritmi euristici di queste note sono illustrati su quattro problemi classici di ottimizzazione combinatoria: il problema dello zaino 0/1 (KP), il problema del bin packing (BPP), il problema del commesso viaggiatore (TSP) e $P\,||\,C_{\max}$. Le loro definizioni e ipotesi sono nelle rispettive pagine; la tabella richiama la notazione.

problema	dati	soluzione	obiettivo
KP	oggetti $\mathscr{I}$, profitti p_j, pesi w_j, capacità c	sottoinsieme di oggetti	max profitto totale
BPP	oggetti $\mathscr{I}$, pesi w_j, capacità c	schemi di riempimento	min numero di contenitori
TSP	città $\mathscr{V}$, distanze d_{ij}	giro σ	min lunghezza $L(\sigma)$
$P\,||\,C_{\max}$	lavori $\mathscr{J}$, tempi t_j, macchine m	insiemi $J_1, \ldots, J_m$	min makespan

- Assumiamo che ogni istanza soddisfi le ipotesi del suo problema: $w_j \leq c$ per tutti gli oggetti e $\sum_j w_j > c$ per il KP e il BPP; $n \geq 3$ e distanze simmetriche per il TSP; $n > m$ per $P \,||\, C_{\max}$.

- Le quattro istanze seguenti si usano in tutte le note sugli algoritmi euristici. I loro valori ottimi sono noti, quindi la qualità di ogni soluzione euristica si può misurare esattamente.

Esempio 1.1: Istanza di riferimento del KP

Uno zaino di capacità $c = 9$ e 4 oggetti con profitti e pesi

	$j=1$	$j=2$	$j=3$	$j=4$
p_j	10	7	6	4
w_j	5	4	3	3

La soluzione ottima è $\tilde{S} = \{1, 2\}$, di peso 9 e profitto $z(I) = 17$.

Esempio 1.2: Istanza di riferimento del BPP

Contenitori di capacità $c = 7$ e 6 oggetti con pesi

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
w_j	4	4	5	3	2	3

Il peso totale è $21 = 3c$, quindi servono almeno 3 contenitori, e gli schemi di riempimento $\{1, 4\}$, $\{2, 6\}$, $\{3, 5\}$ usano esattamente 3 contenitori, tutti pieni: $z(I) = 3$.

Esempio 1.3: Istanza di riferimento del TSP

5 città con le distanze simmetriche

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

L'istanza è metrica. Fra i suoi 12 giri distinti, l'unico ottimo è $\sigma = (1, 3, 5, 2, 4)$, di lunghezza $z(I) = 18$.

Esempio 1.4: Istanza di riferimento di $P||C_{\max}$

$m = 3$ macchine identiche e 7 lavori con tempi di lavorazione

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
t_j	5	5	4	4	3	3	3

Il tempo di lavorazione totale è 27, quindi il makespan è almeno $\lceil 27/3 \rceil = 9$, e gli insiemi $J_1 = \{1, 3\}$, $J_2 = \{2, 4\}$, $J_3 = \{5, 6, 7\}$ lo raggiungono: $z(I) = 9$.

1.3 Bound primali e duali

- Il valore di *qualunque* soluzione ammissibile limita il valore ottimo da un lato, e si chiama **bound primale**. Un algoritmo euristico è un metodo per calcolare bound primali.

- Un bound sul valore ottimo dall'*altro* lato, ottenuto senza calcolare una soluzione ammissibile, si chiama **bound duale**. I bound duali vengono da rilassamenti del problema, come il rilassamento lineare, o da semplici proprietà del problema.

- Per un problema di minimo un bound primale è un **upper bound** UB e un bound duale è un **lower bound** LB; per un problema di massimo i ruoli si scambiano. In entrambi i casi il valore ottimo è compreso fra i due bound:

$$\begin{cases} \mathrm{LB}(I) \le z(I) \le \mathscr{A}(I) & \text{se } \mathscr{P} \text{ è un problema di minimo,} \\ \mathscr{A}(I) \le z(I) \le \mathrm{UB}(I) & \text{se } \mathscr{P} \text{ è un problema di massimo.} \end{cases}$$

- Il **gap** di un valore euristico $\mathscr{A}(I)$ è il gap relativo $\frac{\mathrm{UB}-\mathrm{LB}}{\mathrm{LB}}$ definito nel modulo *Problemi di ottimizzazione combinatoria*, «Bound primali, bound duali e gap»: la differenza dei due bound divisa per il **minore**. Con il valore euristico al posto del bound primale diventa

$$gap(I) = \begin{cases} \dfrac{\mathscr{A}(I) - \mathrm{LB}(I)}{\mathrm{LB}(I)} & \text{se } \mathscr{P} \text{ è un problema di minimo,} \\ \dfrac{\mathrm{UB}(I) - \mathscr{A}(I)}{\mathscr{A}(I)} & \text{se } \mathscr{P} \text{ è un problema di massimo.} \end{cases}$$

- L'**errore relativo** di $\mathscr{A}$ su I è $\frac{\mathscr{A}(I)-z(I)}{z(I)}$ per un problema di minimo e $\frac{z(I)-\mathscr{A}(I)}{z(I)}$ per un problema di massimo. Non si può calcolare quando $z(I)$ è ignoto, che è la situazione tipica; il gap, invece, usa solo quantità calcolate.

Proposizione 1.2

Per ogni istanza I, l'errore relativo di $\mathscr{A}$ su I è al più $gap(I)$.

Dimostrazione. Per un problema di minimo, $\mathrm{LB}(I) \le z(I)$, quindi $\frac{\mathscr{A}(I)}{z(I)} \le \frac{\mathscr{A}(I)}{\mathrm{LB}(I)}$ e

$$\frac{\mathscr{A}(I) - z(I)}{z(I)} = \frac{\mathscr{A}(I)}{z(I)} - 1 \;\le\; \frac{\mathscr{A}(I)}{\mathrm{LB}(I)} - 1 = gap(I).$$

Per un problema di massimo, $\mathscr{A}(I) \le z(I) \le \mathrm{UB}(I)$, quindi

$$\frac{z(I) - \mathscr{A}(I)}{z(I)} \;\le\; \frac{z(I) - \mathscr{A}(I)}{\mathscr{A}(I)} \;\le\; \frac{\mathrm{UB}(I) - \mathscr{A}(I)}{\mathscr{A}(I)} = gap(I).$$

□

- Il gap è quindi un **certificato**: se $gap(I) \le 5\%$, la soluzione dista al più il 5% dall'ottimo, anche se l'ottimo è ignoto. Se $gap(I) = 0$, la soluzione è ottima.

Esempio 1.5: Bound e gap sulle istanze di riferimento

KP. La soluzione $\{1, 3\}$ ha peso $8 \le 9$ e profitto $\mathscr{A}(I) = 16$. Il rilassamento lineare, con gli oggetti ordinati per efficienza non crescente $(1, 3, 2, 4)$, inserisce gli oggetti 1 e 3 e una frazione $\frac{1}{4}$ dell'oggetto 2, quindi $z(I) \le 10 + 6 + 7 \cdot \frac{1}{4} = 17.75$ e, poiché i profitti sono interi, $\mathrm{UB}(I) = 17$. Quindi $gap(I) = \frac{17-16}{16} = 6.25\%$, mentre l'errore relativo è $\frac{17-16}{17} \approx 5.9\%$.

BPP. Gli schemi di riempimento $\{1\}$, $\{2\}$, $\{3\}$, $\{4, 5\}$, $\{6\}$ sono ammissibili e usano $\mathscr{A}(I) = 5$ contenitori, mentre $\mathrm{LB}(I) = \lceil \frac{21}{7} \rceil = 3$. Quindi $gap(I) = \frac{5-3}{3} \approx 66.7\%$.

TSP. Il giro $(1, 3, 2, 4, 5)$ ha lunghezza $\mathscr{A}(I) = 2 + 4 + 3 + 7 + 9 = 25$. Un bound duale è dato dalla Proposizione 1.3 più avanti: $\mathrm{LB}(I) = 17$. Quindi $gap(I) = \frac{25-17}{17} \approx 47.1\%$, mentre l'errore relativo è $\frac{25-18}{18} \approx 38.9\%$.

$P \,||\, C_{\max}$**.** La soluzione con $J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$ ha makespan $\mathscr{A}(I) = 11$, mentre $\mathrm{LB}(I) = \max\{5, \lceil 27/3 \rceil\} = 9$. Quindi $gap(I) = \frac{11-9}{9} \approx 22.2\%$.

Sul BPP e su $P \,||\, C_{\max}$ il bound duale è uguale a $z(I)$, quindi il gap è uguale all'errore relativo. Anche sul KP il bound duale è uguale a $z(I)$, ma il gap divide per il valore minore $\mathscr{A}(I)$ ed è leggermente più grande. Sul TSP il bound duale è più debole, e il gap sovrastima l'errore relativo.

Proposizione 1.3

Per ogni istanza I del TSP, siano a_i e b_i le due distanze minime dalla città i alle altre città. Allora

$$z(I) \;\ge\; \left\lceil \frac{1}{2} \sum_{i \in \mathcal{V}} (a_i + b_i) \right\rceil.$$

Dimostrazione. In ogni giro ogni città è estremo di esattamente due spigoli, di lunghezza almeno a_i e b_i. Sommando le lunghezze dei due spigoli incidenti in ogni città si conta

due volte ogni spigolo del giro, quindi

$$2\,L(\sigma) \;=\; \sum_{i\in V} \big(\text{lunghezze dei due spigoli incidenti in } i\big) \;\geq\; \sum_{i\in V}(a_i + b_i)$$

per ogni giro σ, e in particolare per uno ottimo. Poiché le distanze sono intere, $z(I)$ è un intero, e il bound si può arrotondare per eccesso. □

- Sull'istanza di riferimento le due distanze minime sono $(2,2)$ per la città 1, $(3,4)$ per la città 2, $(2,4)$ per la città 3, $(2,3)$ per la città 4 e $(4,7)$ per la città 5, quindi $\mathrm{LB}(I) = \left\lceil \frac{4+7+6+5+11}{2} \right\rceil = \lceil 16.5 \rceil = 17$.

Un algoritmo euristico dà solo un lato del gap. Senza un bound duale sull'altro lato, non si può dire nulla sulla qualità della soluzione, per quanto buona possa sembrare.

1.4 Classificazione degli algoritmi euristici

- Gli algoritmi euristici si classificano in base a come producono e usano le soluzioni, nelle quattro classi seguenti.

classe	idea
euristiche costruttive	costruiscono una soluzione passo per passo, estendendo una soluzione parziale con un elemento alla volta; una scelta, una volta fatta, non viene mai rivista
ricerca locale	parte da una soluzione ammissibile e la sostituisce ripetutamente con una soluzione migliore ottenuta con una piccola modifica, finché nessuna piccola modifica la migliora
metaeuristiche	strategie generali che guidano una ricerca locale oltre il punto in cui si ferma, esplorando altre regioni delle soluzioni
euristiche basate sulla PLI	usano una formulazione di programmazione lineare intera mista, il suo rilassamento e un solver MIP per guidare la ricerca o per esplorare grandi insiemi di soluzioni

- Le classi non sono alternative: una ricerca locale ha bisogno di una soluzione di partenza, di solito calcolata da un'euristica costruttiva; una metaeuristica contiene una o più ricerche locali; un'euristica basata sulla PLI può partire dalla soluzione di una qualunque delle altre, e migliorarla.

1.5 Rappresentazione delle soluzioni e spazio di ricerca

- Un algoritmo euristico manipola soluzioni, che vanno memorizzate in una struttura dati: la **rappresentazione** delle soluzioni. L'insieme di tutti gli oggetti che la rappresentazione può descrivere è lo **spazio di ricerca**.

- La rappresentazione è una scelta di progetto, e determina quali modifiche di una soluzione sono facili da eseguire. Per i quattro problemi usiamo le rappresentazioni seguenti.

problema	rappresentazione	dimensione dello spazio di ricerca	ammissibile?
KP	vettore binario $x \in \{0,1\}^n$	2^n	non sempre: la capacità può essere superata
BPP	vettore bin, con bin_j il contenitore dell'oggetto j	al più n^n	non sempre: un contenitore può essere troppo pieno
TSP	permutazione σ delle città	$\frac{(n-1)!}{2}$ giri distinti	sempre
$P \,\|\|\, C_{\max}$	vettore $mach$, con $mach_j$ la macchina del lavoro j	m^n	sempre

- Quando la rappresentazione può descrivere oggetti inammissibili, un'euristica deve limitarsi a quelli ammissibili oppure accettare quelli inammissibili e penalizzarli.

- Gli spazi di ricerca crescono molto rapidamente: un TSP con $n = 20$ città ha $\frac{19!}{2} \approx 6 \cdot 10^{16}$ giri distinti. Un'euristica esplora solo una minima parte dello spazio di ricerca, e la domanda è *quale* parte.

1.6 Intorni e ottimi locali

- Sia $\mathscr{F}(I)$ l'insieme delle soluzioni ammissibili dell'istanza I, e sia $f(S)$ il valore della soluzione $S \in \mathscr{F}(I)$, così che $z(I)$ sia il minimo (o il massimo) di f su $\mathscr{F}(I)$.

Definizione 1.4: Intorno

Un **intorno** è una funzione N che associa a ogni soluzione ammissibile $S \in \mathscr{F}(I)$ un insieme $N(S) \subseteq \mathscr{F}(I)$ di soluzioni ammissibili, i **vicini** di S. Una **mossa** è la trasformazione di S in uno dei suoi vicini.

- Un intorno si definisce di solito con una famiglia di piccole modifiche della rappresentazione: cambiare una componente del vettore x, spostare un lavoro su un'altra macchina, sostituire due spigoli di un giro.

Definizione 1.5: Ottimo locale

Una soluzione ammissibile $S \in \mathscr{F}(I)$ è un **ottimo locale** rispetto all'intorno N se nessun vicino di S è migliore di S:

$$\begin{cases} f(S) \leq f(S') & \text{per ogni } S' \in N(S) \quad \text{se } \mathscr{P} \text{ è un problema di minimo,} \\ f(S) \geq f(S') & \text{per ogni } S' \in N(S) \quad \text{se } \mathscr{P} \text{ è un problema di massimo.} \end{cases}$$

Una soluzione ottima si chiama anche **ottimo globale**.

- Un ottimo globale è un ottimo locale per ogni intorno, ma un ottimo locale non è necessariamente un ottimo globale. Inoltre, **essere un ottimo locale dipende dall'intorno**: la stessa soluzione può essere un ottimo locale per un intorno e non per un altro.

Esempio 1.6: Due intorni per il KP

Sull'istanza di riferimento del KP ($c = 9$), consideriamo la soluzione $x = (1, 0, 1, 0)$, cioè gli oggetti $\{1, 3\}$, di peso 8 e profitto 16.

L'intorno **flip** cambia una componente di x, cioè aggiunge o toglie un oggetto:

vicino	peso	profitto	ammissibile
$(0, 0, 1, 0)$	3	6	sì
$(1, 1, 1, 0)$	12	—	no
$(1, 0, 0, 0)$	5	10	sì
$(1, 0, 1, 1)$	11	—	no

I vicini ammissibili hanno profitto 6 e 10, minore di 16: x è un **ottimo locale** rispetto all'intorno flip.

L'intorno **swap** toglie un oggetto scelto e aggiunge un oggetto non scelto:

vicino	peso	profitto	ammissibile
$(0, 1, 1, 0)$: esce 1, entra 2	7	13	sì
$(0, 0, 1, 1)$: esce 1, entra 4	6	10	sì
$(1, 1, 0, 0)$: esce 3, entra 2	9	17	sì
$(1, 0, 0, 1)$: esce 3, entra 4	8	14	sì

Il vicino $(1, 1, 0, 0)$ ha profitto $17 > 16$: x **non** è un ottimo locale rispetto all'intorno swap, e la mossa porta all'ottimo globale.

Definizione 1.6: Intorno esatto

Un intorno N è **esatto** se ogni ottimo locale rispetto a N è un ottimo globale.

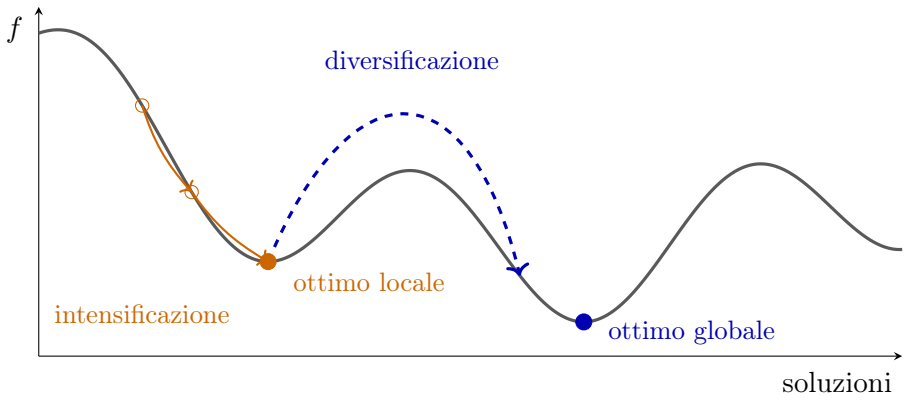
- Gli intorni esatti sono rari, ed esistono per problemi risolvibili in tempo polinomiale: l'intorno del metodo del simplesso, che si sposta fra vertici adiacenti del poliedro, è esatto

per la programmazione lineare; l'intorno che scambia uno spigolo entrante e uno uscente è esatto per il problema dell'albero ricoprente di costo minimo.

- Per i problemi NP-hard gli intorni piccoli non sono esatti, e gli intorni esatti, come quello che contiene tutte le soluzioni ammissibili, sono troppo grandi per essere esplorati. Il progetto di un intorno è un compromesso fra la sua **dimensione**, che determina il tempo necessario per esplorarlo, e la **qualità** dei suoi ottimi locali.

1.7 Intensificazione e diversificazione

- I valori delle soluzioni, visti come funzione sullo spazio di ricerca, formano il **paesaggio** esplorato da un'euristica. Per un problema di minimo gli ottimi locali sono i fondi delle valli, e l'ottimo globale è il fondo della valle più profonda.



- L'**intensificazione** è la ricerca di soluzioni migliori nella regione dove sono già state trovate buone soluzioni: si scende verso il fondo della valle corrente. La ricerca locale è pura intensificazione, e si ferma al primo ottimo locale che raggiunge.

- La **diversificazione** è lo spostamento verso regioni dello spazio di ricerca non ancora esplorate: si lascia la valle corrente, eventualmente passando per soluzioni peggiori, per raggiungerne un'altra.

- Troppa intensificazione intrappola la ricerca nella prima valle; troppa diversificazione la fa vagare senza mai esplorare una valle fino in fondo. Ogni metaeuristica è un modo di bilanciare le due.

1.8 Come valutare un algoritmo euristico

- Un algoritmo euristico si valuta in base alla **qualità** delle sue soluzioni e al suo **tempo di esecuzione**, che sono obiettivi in conflitto.

- La **qualità** si misura con il gap rispetto al miglior bound duale disponibile, o con l'errore relativo quando il valore ottimo è noto, come sulle istanze di riferimento di queste note.

- L'**analisi del caso peggiore**, come per gli algoritmi di approssimazione, dà garanzie valide per tutte le istanze, ma spesso lontane dal comportamento tipico. L'**analisi sperimentale** misura i gap e i tempi di esecuzione su un insieme di istanze di benchmark, che deve essere abbastanza grande e vario da evitare conclusioni valide solo per poche istanze.

- Una singola istanza non dimostra nulla: sull'istanza di riferimento del TSP la stessa regola, applicata da città di partenza diverse, dà giri di lunghezza fra 18 e 25. I confronti fra euristiche devono usare le stesse istanze e riportare la distribuzione dei risultati, non solo il migliore.

- Per le euristiche che fanno scelte casuali, i risultati dipendono dai numeri casuali: ogni istanza va risolta più volte, con diversi **semi** del generatore di numeri casuali, riportando la media e la dispersione dei risultati.

Un algoritmo euristico è buono se, sulle istanze a cui è destinato, trova con regolarità soluzioni con un gap piccolo nel tempo disponibile.

1.9 Errori tipici

Attenzione

- **Giudicare un'euristica da una sola esecuzione o da una sola istanza.** La stessa regola, da città di partenza diverse, dà giri di lunghezza fra 18 e 25 sull'istanza di riferimento del TSP.
- **Confondere il gap con l'errore relativo.** Il gap si misura rispetto a un bound duale, quindi non è mai minore dell'errore relativo; in un problema di minimo coincidono solo quando il bound duale è uguale a $z(I)$.
- **Usare il denominatore sbagliato.** Il gap divide sempre per il bound minore: per un problema di massimo è $\frac{\mathrm{UB}(I)-\mathscr{A}(I)}{\mathscr{A}(I)}$, per un problema di minimo $\frac{\mathscr{A}(I)-\mathrm{LB}(I)}{\mathrm{LB}(I)}$.
- **Dimenticare di arrotondare il bound duale.** Con dati interi il bound 17.75 del KP si arrotonda per difetto a 17, e il bound 16.5 del TSP si arrotonda per eccesso a 17.

- **Fidarsi di una soluzione che sembra buona.** Senza un bound duale sull'altro lato, non si può dire nulla sulla qualità di una soluzione.

1.10 In sintesi

In sintesi

- Un algoritmo euristico restituisce una soluzione ammissibile senza garantirne l'ottimalità: il suo valore è un bound *primale*.
- Il gap confronta il valore euristico con il miglior bound duale, ed è un upper bound sull'errore relativo.
- Gli algoritmi euristici sono costruttivi, di ricerca locale, metaeuristiche o basati sulla PLI; un intorno definisce gli ottimi locali, e la ricerca bilancia intensificazione e diversificazione.
- La qualità si valuta su molte istanze e, per i metodi casuali, su molti semi.

Quadro riassuntivo delle istanze di riferimento. Soluzioni euristiche di questo capitolo e loro bound duali:

problema	bound primale	bound duale	gap	$z(I)$
KP (max)	16	$\mathrm{UB}(I) = 17$	6.25%	17
BPP (min)	5	$\mathrm{LB}(I) = 3$	66.7%	3
TSP (min)	25	$\mathrm{LB}(I) = 17$	47.1%	18
$P \,\|\|\, C_{\max}$ (min)	11	$\mathrm{LB}(I) = 9$	22.2%	9

Capitolo 2

Euristiche costruttive

Prerequisiti

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

Obiettivi

Alla fine di questo capitolo saremo in grado di:

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

2.1 Euristiche costruttive

Definizione 2.1: Euristica costruttiva

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

- Gli elementi dipendono dal problema: gli oggetti selezionati per lo zaino, gli assegnamenti degli oggetti ai contenitori e dei lavori alle macchine, gli archi di un giro.
- La qualità del risultato dipende da come si sceglie, a ogni passo, l'elemento da aggiungere (Sezione 2.3).

2.2 I problemi e le ipotesi

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

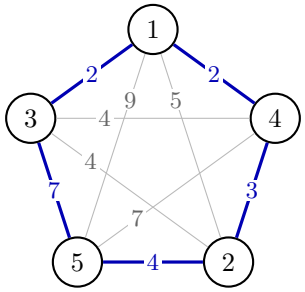
- Come nelle altre note, $\mathscr{A}(I)$ indica il valore della soluzione calcolata dall'euristica $\mathscr{A}$ sull'istanza I, e $z(I)$ il valore ottimo. Per $P\,||\,C_{\max}$, $L_h = \sum_{j \in J_h} t_j$ è il **carico** della macchina h, e il vettore $mach$ memorizza la macchina $mach_j$ di ogni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

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

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

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

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



2.3 Il paradigma greedy

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

- A ogni passo bisogna rispondere a tre domande: **quali elementi si possono aggiungere** (i candidati), **quale si aggiunge** (il criterio di selezione) e **se il risultato si può ancora completare in una soluzione ammissibile** (il test di ammissibilità). Un'euristica **greedy** risponde alla seconda domanda nel modo più semplice: aggiunge il candidato che sembra migliore *adesso*, secondo un criterio che misura il beneficio immediato dell'elemento.

Definizione 2.2: Algoritmo greedy

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

In un algoritmo greedy ogni decisione si prende una volta per tutte: *una scelta, una volta fatta, non si rivede mai.*

- Lo pseudocodice qui sotto è uno schema più che un algoritmo: l'insieme dei candidati, il criterio e il test sono le tre parti che ogni euristica di queste note definisce in modo specifico.

Algoritmo

Algorithm: Schema greedy

Input: un'istanza I, l'insieme C degli elementi delle soluzioni
Output: una soluzione ammissibile S

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

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

- **Criteri statici e dinamici.** Un criterio è *statico* se il valore di un elemento non dipende dalla soluzione parziale: gli elementi si possono allora ordinare una sola volta all'inizio, e l'euristica scorre la lista ordinata (efficienza degli oggetti del KP, peso degli oggetti del BPP, tempo di esecuzione dei lavori). Un criterio è *dinamico* se il valore di un elemento cambia man mano che la soluzione cresce, e va ricalcolato a ogni passo (distanza dall'ultima città di un giro parziale, capacità residua dei contenitori, carico delle macchine).

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

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

- **Quando il greedy è esatto.** Per alcuni problemi un algoritmo greedy trova sempre una soluzione ottima: l'algoritmo di Kruskal per il problema dell'albero ricoprente di costo minimo esamina gli archi per costo non decrescente e accetta un arco se non chiude un ciclo; il problema dello zaino frazionario si risolve all'ottimo prendendo gli oggetti per efficienza non crescente. Per i quattro problemi NP-difficili di queste note nessun algoritmo greedy è esatto, e la domanda diventa *quale* criterio dà buone soluzioni.

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

problema	elemento	criterio di selezione	test di ammissibilità
KP	un oggetto	profitto, peso o efficienza (statico)	l'oggetto entra nella capacità residua
BPP	un oggetto e un contenitore	oggetto: ordine di input o peso (statico); contenitore: il primo, il più pieno, il più vuoto (dinamico)	l'oggetto entra nel contenitore
$P \,\|\|\, C_{\max}$	un lavoro e una macchina	lavoro: ordine di input o tempo di esecuzione (statico); macchina: la meno carica (dinamico)	sempre ammissibile
TSP	una città e la sua posizione	la più vicina, la più lontana, la più economica (dinamico)	sempre ammissibile

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

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

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

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

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

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

Algoritmo

Algorithm: Greedy con ordinamento

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

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

```
1  Function GreedyOrderKP(n, c, w, p, ρ)
2      for j := 1 to n do
3          x[j] ← 0                    // all'inizio l'oggetto j non è selezionato
4      end
5      rc := c                                              // capacità residua
6      for k := 1 to n do
7          j := ρ[k]                          // k-esimo oggetto nell'ordinamento
8          if w[j] ≤ rc then
9              x[j] ← 1                                // selezionare l'oggetto j
10             rc ← rc − w[j]                // aggiornare la capacità residua
11         end
12     end
13     return x
```

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

Esempio 2.1: Tre ordinamenti sull'istanza di riferimento del KP

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

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

(gli oggetti 3 e 4 hanno lo stesso peso, gli oggetti 1 e 3 la stessa efficienza, e i pareggi si risolvono con l'indice). Le esecuzioni, con la capacità residua dopo ogni passo, sono:

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

Quindi

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

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

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

  In tutti e tre i casi il rapporto $\frac{z(I)}{\mathscr{A}(I)}$ cresce con c.

- Un rimedio è **combinare** due soluzioni: l'algoritmo Ext-Greedy GR^e restituisce la migliore fra la soluzione di GR e il miglior oggetto singolo, e questa semplice combinazione garantisce $\frac{z(I)}{GR^e(I)} \leq 2$. Eseguire più euristiche greedy e tenere la soluzione migliore è un modo generale ed economico per proteggersi dalla debolezza di ogni singolo criterio.

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

- Quattro euristiche greedy classiche per il BPP elaborano gli oggetti **nell'ordine di input** e differiscono per la regola con cui scelgono il contenitore: Next Fit (NF) tiene aperto un solo contenitore, mentre First Fit (FF), Best Fit (BF) e Worst Fit (WF) tengono tutti i contenitori e scelgono, fra quelli in cui l'oggetto entra, il primo aperto, il più pieno o il più vuoto. Sull'istanza di riferimento del BPP ($c = 7$, $w = (4, 4, 5, 3, 2, 3)$, $z(I) = 3$) danno

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

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

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

2.5.1 First Fit Decreasing

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

Algoritmo

Algorithm: First Fit Decreasing

Input: istanza I del BPP con n oggetti, pesi $w[1..n]$, capacità c
Output: assegnamento oggetti-contenitori $bin[1..n]$ e numero di contenitori usati nb

```
1  Function FirstFitDecreasing(n, w, c)
2      ρ := gli oggetti ordinati per peso non crescente        // prima ordinare
3      bin := nuovo vettore[1..n]          // assegnamento oggetti-contenitori
4      rc := nuovo vettore[1..n]                              // capacità residue
5      nb := 0                                              // numero di contenitori
6      for k := 1 to n do
7          j := ρ[k]              // poi riempire:  k-esimo oggetto più grande
8          s := −1                                          // contenitore selezionato
9          for b := 1 to nb do
10             if w[j] ≤ rc[b] then
11                 s ← b
12                 break
13             end
14         end
15         if s = −1 then
16             nb ← nb + 1                          // aprire un nuovo contenitore
17             rc[nb] ← c
18             s ← nb
19         end
20         bin[j] ← s                                       // assegnare l'oggetto
21         rc[s] ← rc[s] − w[j]
22     end
23     return bin, nb
```

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

Esempio 2.2: FF e FFD sull'istanza di riferimento del BPP

FF, nell'ordine di input: gli oggetti 1, 2 e 3 (pesi $4, 4, 5$) aprono un contenitore ciascuno, perché nessuna coppia di essi entra insieme; l'oggetto 4 (peso 3) va nel contenitore 1 ($rc[1] = 0$); l'oggetto 5 (peso 2) va nel contenitore 2 ($rc[2] = 1$); l'oggetto 6 (peso 3) non entra in

nessun contenitore, le cui capacità residue sono $0, 1, 2$, e apre il contenitore 4. Quindi $FF(I) = 4$.

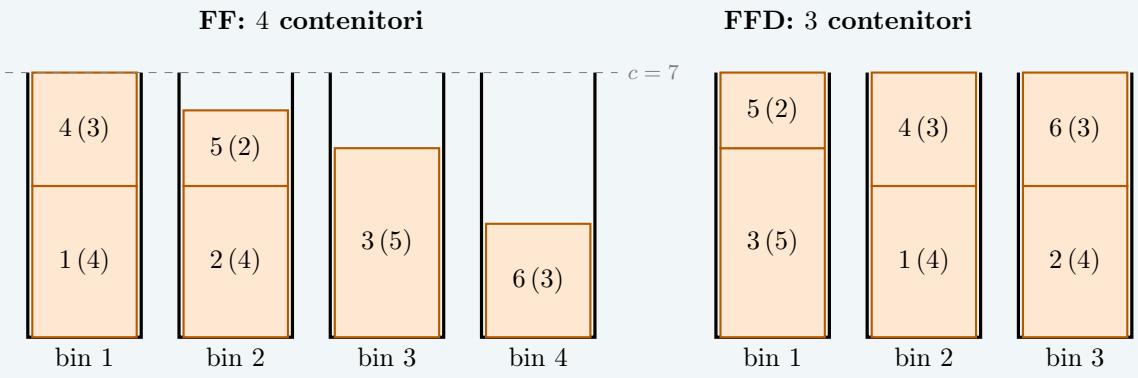
FFD. I pesi ordinati per valore non crescente sono $5, 4, 4, 3, 3, 2$, cioè $\rho = (3, 1, 2, 4, 6, 5)$ (pareggi per indice). Quindi:

1. Oggetto 3 (peso 5): nessun contenitore è aperto, FFD apre il contenitore 1; $rc[1] = 7 - 5 = 2$.
2. Oggetto 1 (peso 4): $4 > rc[1] = 2$, FFD apre il contenitore 2; $rc[2] = 7 - 4 = 3$.
3. Oggetto 2 (peso 4): $4 > rc[1] = 2$, $4 > rc[2] = 3$, FFD apre il contenitore 3; $rc[3] = 7 - 4 = 3$.
4. Oggetto 4 (peso 3): $3 > rc[1] = 2$, $3 \leq rc[2] = 3$: contenitore 2; $rc[2] = 0$.
5. Oggetto 6 (peso 3): $3 > rc[1] = 2$, $3 > rc[2] = 0$, $3 \leq rc[3] = 3$: contenitore 3; $rc[3] = 0$.
6. Oggetto 5 (peso 2): $2 \leq rc[1] = 2$: contenitore 1; $rc[1] = 0$.

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

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

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



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

2.5.2 Best Fit Decreasing

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

Algoritmo

Algorithm: Best Fit Decreasing

Input: istanza I del BPP con n oggetti, pesi $w[1..n]$, capacità c
Output: assegnamento oggetti-contenitori $bin[1..n]$ e numero di contenitori usati nb

```
1  Function BestFitDecreasing(n, w, c)
2      ρ := gli oggetti ordinati per peso non crescente          // prima ordinare
3      bin := nuovo vettore[1..n]          // assegnamento oggetti-contenitori
4      rc := nuovo vettore[1..n]                                 // capacità residue
5      nb := 0                                                   // numero di contenitori
6      for k := 1 to n do
7          j := ρ[k]                  // poi riempire:  k-esimo oggetto più grande
8          s := −1                                               // contenitore selezionato
9          for b := 1 to nb do
10             if w[j] ≤ rc[b] and (s = −1 or rc[b] < rc[s]) then
11                 s ← b                  // contenitore più pieno trovato finora
12             end
13         end
14         if s = −1 then
15             nb ← nb + 1                                       // aprire un nuovo contenitore
16             rc[nb] ← c
17             s ← nb
18         end
19         bin[j] ← s                                            // assegnare l'oggetto
20         rc[s] ← rc[s] − w[j]
21     end
22     return bin, nb
```

- Anche qui il ciclo di riempimento è quello di BF, preceduto dall'ordinamento.

Esempio 2.3: BFD sull'istanza di riferimento del BPP

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

$P_1 = \{3, 5\}$, $P_2 = \{1, 4\}$, $P_3 = \{2, 6\}$, e $BFD(I) = 3$.

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

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

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

  e che il bound è tight. Un bound della forma $\frac{11}{9} z(I) + 4$ fu dimostrato per la prima volta sia per FFD sia per BFD da D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey e R.L. Graham (*Worst-case performance bounds for simple one-dimensional packing algorithms*, SIAM Journal on Computing 3 (1974) 299–325); la costante additiva tight è dovuta a G. Dósa (*The tight bound of First Fit Decreasing bin-packing algorithm is* $FFD(I) \leq 11/9\, OPT(I) + 6/9$, ESCAPE 2007, LNCS 4614, pp. 1–11).

- **Non una soluzione migliore su ogni istanza.** Una garanzia nel caso peggiore confronta l'algoritmo con l'ottimo, non con un altro algoritmo: su alcune istanze l'ordine di input è più fortunato di quello ordinato.

Esempio 2.4: L'ordinamento può peggiorare

Siano $c = 7$ e $w = (2, 2, 3, 2, 2, 3)$, con peso totale 14, così che $z(I) \geq \lceil \frac{14}{7} \rceil = 2$.

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

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

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

2.6 Scheduling su macchine parallele: prima i lavori più lunghi

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

2.6.1 List scheduling

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

Algoritmo

Algorithm: List Scheduling

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

Function *ListScheduling*(n, m, t, ρ)
  $mach :=$ nuovo vettore$[1..n]$ `// assegnamento lavori-macchine`
  for $h := 1$ **to** m **do**
    $L[h] \leftarrow 0$ `// carichi delle macchine`
  end
  for $k := 1$ **to** n **do**
    $j := \rho[k]$ `// k-esimo lavoro della lista`
    $s :=$ una macchina con $L[s]$ minimo (indice minimo) `// macchina meno carica`
    $mach[j] \leftarrow s$ `// assegnare il lavoro` j `alla macchina` s
    $L[s] \leftarrow L[s] + t[j]$
  end
  return $mach$, $C_{\max} = \max_h L[h]$

- $LIST$ è il corrispondente, nello scheduling, di una regola *online*: non ha bisogno di conoscere i lavori futuri, e si usa, per esempio, quando i lavori arrivano uno alla volta. Con una scansione lineare delle macchine richiede tempo $O(nm)$, e tempo $O(n \log m)$ con un heap dei carichi.

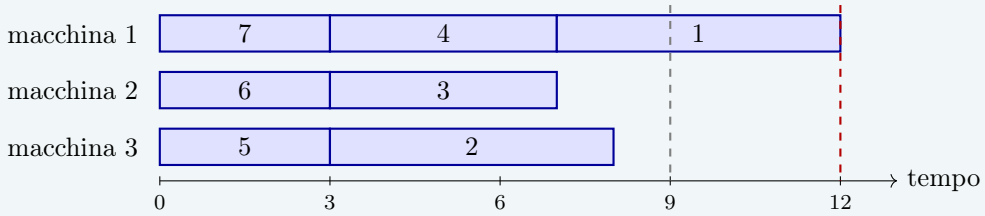
- Nell'istanza di riferimento, $t = (5, 5, 4, 4, 3, 3, 3)$, i lavori sono elencati per tempo di esecuzione non crescente, che è un ordine favorevole. Per vedere l'effetto dell'ordine, l'esempio esamina gli stessi lavori nella lista inversa.

Esempio 2.5: *LIST* sull'istanza di riferimento, prima i lavori più brevi

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

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

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



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

Proposizione 2.3

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

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

Dimostrazione. Dimostrazione omessa. Il risultato è dovuto a

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

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

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

2.6.2 Longest processing time first

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

Algoritmo

Algorithm: Longest Processing Time first

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

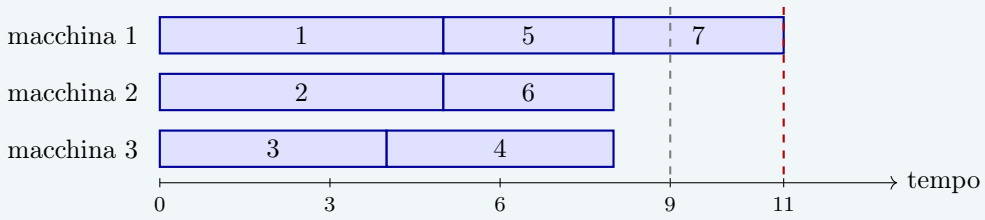
```
1 Function LPT(n, m, t)
2     ρ := i lavori ordinati per t[j] non crescente (pareggi per indice)   // prima i
        lavori più lunghi
3     return ListScheduling(n, m, t, ρ)
```

- **Perché prima i lavori più lunghi.** Il makespan è deciso dal lavoro che finisce per ultimo. Se i lavori in fondo alla lista sono brevi, gli ultimi assegnamenti possono aggiungere solo poco alla macchina meno carica, e fanno da "sabbia" che livella i carichi. Il parallelo con FFD è esatto: prima gli elementi grandi, poi quelli piccoli a riempire i vuoti.

- LPT richiede tempo $O(n \log n + n \log m)$.

Esempio 2.6: LPT sull'istanza di riferimento

I lavori sono già ordinati, $\rho = (1, 2, 3, 4, 5, 6, 7)$. I lavori 1, 2, 3 vanno sulle tre macchine vuote, con carichi $(5, 5, 4)$; il lavoro 4 va sulla macchina 3, la meno carica: $(5, 5, 8)$; il lavoro 5 va sulla macchina 1: $(8, 5, 8)$; il lavoro 6 sulla macchina 2: $(8, 8, 8)$; il lavoro 7 trova tre macchine con carico 8 e va sulla macchina 1: $(11, 8, 8)$. Quindi $J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$, $mach = (1, 2, 3, 3, 1, 2, 1)$, e $LPT(I) = 11$:



Il lower bound elementare del capitolo «Algoritmi euristici: definizioni» dà $z(I) \geq \max\left\{\max_j t_j, \left\lceil \frac{\sum_j t_j}{m} \right\rceil\right\} = \max\{5, 9\} = 9$, e $z(I) = 9$. L'ordinamento migliora il makespan da 12 a 11, ma LPT non è ottimo: dopo sei lavori i carichi sono perfettamente bilanciati, $(8, 8, 8)$, e il settimo lavoro li sbilancia di 3.

Proposizione 2.4

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

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

Dimostrazione. Dimostrazione omessa. Il risultato è dovuto a

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

LPT è un algoritmo di $(\frac{4}{3} - \frac{1}{3m})$-approssimazione per $P\,||\,C_{\max}$: ordinare la lista abbassa la garanzia di $LIST$ da $2 - \frac{1}{m}$ a $\frac{4}{3} - \frac{1}{3m}$.

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

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

L'istanza di riferimento raggiunge esattamente il bound di Graham: appartiene alla famiglia di istanze, data da Graham, con $2m+1$ lavori di tempi di esecuzione $2m-1, 2m-1, 2m-2, 2m-2, \ldots, m+1, m+1, m, m, m$, su cui LPT dà $4m-1$ e l'ottimo è $3m$. Per $m = 3$ questi sono i tempi $(5, 5, 4, 4, 3, 3, 3)$. Il bound della Proposizione 2.4 non si può quindi migliorare.

2.7 Il problema del commesso viaggiatore: accodare o inserire?

- Per il TSP ogni permutazione delle città è un giro ammissibile, quindi il test di ammissibilità è banale, e tutto il progetto sta nel criterio di selezione. Due famiglie di euristiche costruttive sono naturali, e differiscono per **dove** si colloca la nuova città:
  - **accodare**: la soluzione parziale è un cammino, e la nuova città si aggiunge alla sua fine;
  - **inserire**: la soluzione parziale è un sottogiro chiuso, e la nuova città si inserisce fra due città consecutive, dove costa meno.

2.7.1 Nearest neighbour

Il *Nearest Neighbour Algorithm* (indicato con NN) è un'euristica costruttiva per il TSP. Partendo da una città data s, si sposta ripetutamente dall'ultima città visitata alla città *più vicina* non ancora visitata (in caso di pareggio, quella con l'indice minimo); quando tutte le città sono state visitate, torna in s.

Algoritmo

Algorithm: Nearest Neighbour

Input: istanza del TSP con n città, distanze $d[1..n][1..n]$; città di partenza s
Output: giro $\sigma[1..n]$

Function *NearestNeighbour*(n, d, s)
  $\sigma[1] \leftarrow s$ `// il cammino parte da` s
  $visited := \{s\}$
  for $len := 2$ **to** n **do**
    $last := \sigma[len-1]$ `// ultima città del cammino`
    $k :=$ la città $j \notin visited$ con $d[last][j]$ minimo (indice minimo)
    $\sigma[len] \leftarrow k$ `// accodare` k `alla fine`
    $visited \leftarrow visited \cup \{k\}$
  end
  return σ `// il giro si chiude con l'arco` $(\sigma[n], s)$

- Il criterio è dinamico: le distanze che contano sono quelle dall'ultima città, che cambia a ogni passo. NN richiede tempo $O(n^2)$.

- **La debolezza di** NN si vede nei suoi ultimi passi: le città rimaste sono quelle che non sono mai state le più vicine a niente, spesso lontane fra loro e da s, e l'arco di chiusura verso s non viene scelto affatto, è forzato.

Esempio 2.7: NN sull'istanza di riferimento del TSP

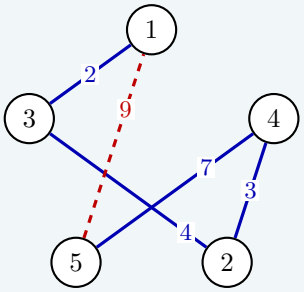
Partendo dalla città 1:

1. dalla città 1 le distanze verso $2, 3, 4, 5$ sono $5, 2, 2, 9$: le città 3 e 4 sono le più vicine, e il pareggio si risolve a favore della città 3;
2. dalla città 3 le distanze verso $2, 4, 5$ sono $4, 4, 7$: città 2 (pareggio con 4);
3. dalla città 2 le distanze verso $4, 5$ sono $3, 4$: città 4;
4. dalla città 4 resta solo la città 5, a distanza 7;
5. il giro si chiude con l'arco $(5, 1)$, di lunghezza 9.

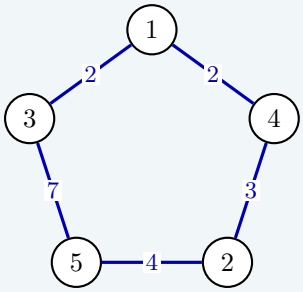
Quindi $\sigma = (1, 3, 2, 4, 5)$ e

$$NN(I) = d_{13} + d_{32} + d_{24} + d_{45} + d_{51} = 2 + 4 + 3 + 7 + 9 = 25,$$

il 39% in più di $z(I) = 18$; è il giro usato nel capitolo «Algoritmi euristici: definizioni» per illustrare il gap. I quattro archi scelti dal criterio hanno lunghezza $2 + 4 + 3 + 7 = 16$, e il solo arco di chiusura forzato costa 9, la distanza più grande dell'istanza.



NN dalla città 1: $L = 25$
(tratteggiato: l'arco di chiusura)



giro ottimo: $L = 18$

2.7.2 Nearest neighbour multi-start

- Il risultato di NN dipende dalla città di partenza, che è una scelta arbitraria: il giro è un ciclo e non ha una prima città naturale. Poiché NN è veloce, lo si può semplicemente eseguire da ogni città.

L'*algoritmo nearest neighbour multi-start* esegue NN una volta da ogni città $s \in \mathcal{V}$ e restituisce il più corto degli n giri ottenuti (in caso di pareggio, il primo trovato).

Algoritmo

Algorithm: Nearest Neighbour multi-start

Input: istanza del TSP con n città, distanze $d[1..n][1..n]$
Output: giro $\sigma^{best}[1..n]$

```
1 Function MultiStartNN(n, d)
2     L^best := +∞
3     for s := 1 to n do
4         σ := NearestNeighbour(n, d, s)
5         if L(σ) < L^best then
6             σ^best ← σ, L^best ← L(σ)          // nuovo giro migliore
7         end
8     end
9     return σ^best
```

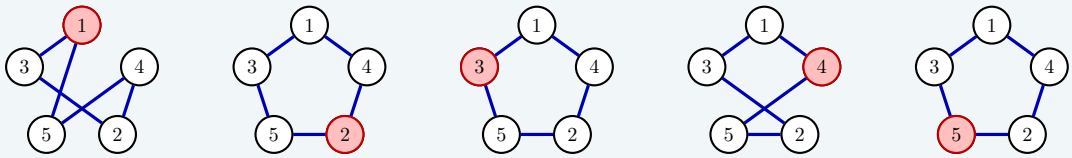
- Il tempo di esecuzione è n volte quello di NN, cioè $O(n^3)$. Il multi-start è il modo più semplice di trasformare una scelta arbitraria in una ricerca: ogni partenza dà una soluzione diversa, e si tiene la migliore.

Esempio 2.8: NN multi-start sull'istanza di riferimento del TSP

Le cinque esecuzioni di NN danno

partenza s	1	2	3	4	5
giro	$(1,3,2,4,5)$	$(2,4,1,3,5)$	$(3,1,4,2,5)$	$(4,1,3,2,5)$	$(5,2,4,1,3)$
L	25	18	18	19	18

I giri dalle città 2, 3 e 5 sono il giro ottimo $(1,3,5,2,4)$, percorso da città diverse e in versi diversi; il multi-start restituisce il giro dalla città 2, di lunghezza $18 = z(I)$.



partenza 1: $L = 25$ partenza 2: $L = 18$ partenza 3: $L = 18$ partenza 4: $L = 19$ partenza 5: $L = 18$

2.7.3 Euristiche di inserimento

- NN può solo aggiungere una città alla fine di un cammino. Un'**euristica di inserimento** mantiene un *sottogiro* chiuso T attraverso alcune delle città, e inserisce la nuova città dove costa meno. Inserire la città k fra due città consecutive i e j di T sostituisce l'arco (i,j) con gli archi (i,k) e (k,j), e aumenta la lunghezza del **costo di inserimento**

$$\Delta(i,k,j) \;=\; d_{ik} + d_{kj} - d_{ij} \qquad \text{per ogni arco } (i,j) \text{ del sottogiro } T.$$

Sulle istanze metriche $\Delta(i,k,j) \geq 0$, per la disuguaglianza triangolare.

- Ogni passo prende due decisioni: **quale** città si inserisce (la regola di selezione) e **dove** (sempre l'arco di costo di inserimento minimo). Le tre euristiche classiche differiscono solo per la regola di selezione. Sia $d(q,T) = \min_{i \in T} d_{qi}$ la distanza della città q dal sottogiro.

Le *euristiche di inserimento* per il TSP partono dal sottogiro $T = (s)$ e, finché tutte le città non sono in T, selezionano una città $k \notin T$ e la inseriscono nell'arco di T di costo di inserimento minimo:

nearest insertion (NI): k minimizza $d(q,T)$, la distanza dal sottogiro

farthest insertion (FI): k massimizza $d(q,T)$

cheapest insertion (CI): k minimizza $\min\limits_{(i,j)\in T} \Delta(i,q,j)$, il costo del suo inserimento

I pareggi si risolvono a favore dell'indice minimo, per la città e per l'arco.

Algoritmo

Algorithm: Inserimento (NI, FI, CI)

Input: istanza del TSP con n città, distanze $d[1..n][1..n]$; città di partenza s; $rule \in \{NI, FI, CI\}$

Output: giro $\sigma[1..n]$

```
1 Function Insertion(n, d, s, rule)
2     σ := (s)                                          // sottogiro con una città
3     for len := 1 to n − 1 do
4         k := la città q ∉ σ selezionata da rule       // quale città
5         a := la posizione a ∈ {1, ..., len} che minimizza Δ(σ[a], k, σ[a + 1]), con
               σ[len + 1] = σ[1]                        // dove
6         inserire k fra σ[a] e σ[a + 1]
7     end
8     return σ
```

- **Il sottogiro con una città.** Il sottogiro (s) è un giro chiuso con un solo "arco" (s, s) di lunghezza 0, quindi inserire k costa $d_{sk} + d_{ks} - 0 = 2d_{sk}$, e il sottogiro (s, k) va da s a k e ritorno. Non serve alcun caso particolare.

- **Accodare o inserire.** Ogni giro costruito da NN potrebbe essere costruito anche da un'euristica di inserimento, ma non viceversa: inserire in qualunque posizione mantiene il sottogiro chiuso a ogni passo, quindi la lunghezza dell'arco di chiusura si paga, e si controlla, fin dall'inizio.

- **L'intuizione dietro FI.** Prendere la città *più lontana* sembra assurdo per un algoritmo greedy, ma così si costruisce prima il contorno del giro, attraverso le città lontane fra loro, e poi si inseriscono le città rimanenti, che sono "di strada" e costano poco.

- Con le distanze $d(q, T)$ e i costi di inserimento minimi aggiornati in modo incrementale dopo ogni inserimento, NI e FI richiedono tempo $O(n^2)$; un'implementazione semplice di CI richiede tempo $O(n^3)$.

Esempio 2.9: NI e CI sull'istanza di riferimento del TSP

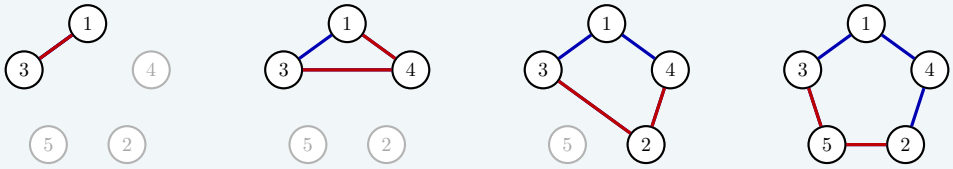
Partiamo da $T = (1)$. A ogni passo elenchiamo, per ogni città $q \notin T$, la sua distanza $d(q, T)$ e i suoi costi di inserimento sugli archi di T, nell'ordine di T.

sottogiro T	$d(q, T)$	costi di inserimento	inserita
(1)	$2\colon 5,\ 3\colon 2,\ 4\colon 2,\ 5\colon 9$	$2\colon 10,\ 3\colon 4,\ 4\colon 4,\ 5\colon 18$	3 su $(1, 1)$, costo 4
$(1, 3)$	$2\colon 4,\ 4\colon 2,\ 5\colon 7$	$2\colon (7, 7),\ 4\colon (4, 4),\ 5\colon (14, 14)$	4 su $(1, 3)$, costo 4
$(1, 4, 3)$	$2\colon 3,\ 5\colon 7$	$2\colon (6, 3, 7),\ 5\colon (14, 10, 14)$	2 su $(4, 3)$, costo 3
$(1, 4, 2, 3)$	$5\colon 4$	$5\colon (14, 8, 7, 14)$	5 su $(2, 3)$, costo 7

NI seleziona la città con la $d(q,T)$ minima, CI la città con il costo di inserimento minimo: su questa istanza le due regole selezionano le stesse città a ogni passo (la città 3 al primo passo per il pareggio con la città 4). Entrambe restituiscono $\sigma = (1,4,2,5,3)$, e

$$NI(I) = CI(I) = 4+4+3+7 = 18 = z(I),$$

il giro ottimo $(1,3,5,2,4)$ percorso nel verso opposto. I sottogiri crescono come segue (nuovi archi in rosso):



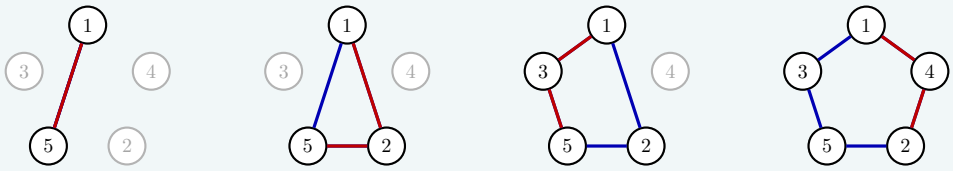
$L = 4$ $L = 8$ $L = 11$ $L = 18$

Esempio 2.10: FI sull'istanza di riferimento del TSP

Partiamo da $T = (1)$:

sottogiro T	$d(q,T)$	costi di inserimento della città selezionata	inserita
(1)	2: 5, 3: 2, 4: 2, 5: 9	5: 18	5 su $(1,1)$, costo 18
$(1,5)$	2: 4, 3: 2, 4: 2	2: $(0,0)$	2 su $(1,5)$, costo 0
$(1,2,5)$	3: 2, 4: 2	3: $(1,7,0)$	3 su $(5,1)$, costo 0
$(1,2,5,3)$	4: 2	4: $(0,6,4,4)$	4 su $(1,2)$, costo 0

Al primo passo FI prende la città 5, la più lontana dalla città 1, e il sottogiro $(1,5)$ ha già lunghezza $2d_{15} = 18$. Le città 2, 3 e 4 sono "di strada" e si inseriscono a costo zero, perché $d_{12} + d_{25} = d_{15}$, $d_{53} + d_{31} = d_{51}$ e $d_{14} + d_{42} = d_{12}$. Quindi $\sigma = (1,4,2,5,3)$ e $FI(I) = 18 = z(I)$: lo stesso giro di NI e CI, costruito in un ordine completamente diverso.



$L = 18$ $L = 18$ $L = 18$ $L = 18$

- **Che cosa si sa nel caso peggiore.** Sulle istanze metriche, NI e CI non restituiscono mai un giro più lungo del doppio di quello ottimo, mentre il rapporto di NN può crescere come $\log n$ e per FI è noto solo un bound logaritmico (D.J. Rosenkrantz, R.E. Stearns, P.M. Lewis, *An analysis of several heuristics for the traveling salesman problem*, SIAM Journal on Computing 6 (1977) 563–581). In pratica, sulle istanze euclidee, FI è di solito la migliore delle euristiche di inserimento, e i giri di NN sono tipicamente circa il 25% più lunghi dell'ottimo.

2.7.4 Un confronto su un'istanza più grande

- Sull'istanza di riferimento le tre euristiche di inserimento, da ogni città di partenza, trovano tutte il giro ottimo, e le differenze fra le regole non si vedono. Per confrontarle usiamo una seconda istanza del TSP, con 6 città nel piano e distanze uguali alle distanze euclidee arrotondate all'intero più vicino. L'istanza è metrica (verificato su tutte le terne di città).

Esempio 2.11: Un'istanza del TSP con 6 città

Le città hanno coordinate

$$1\colon(2,1),\quad 2\colon(5,5),\quad 3\colon(4,4),\quad 4\colon(0,5),\quad 5\colon(7,0),\quad 6\colon(4,1),$$

e distanze

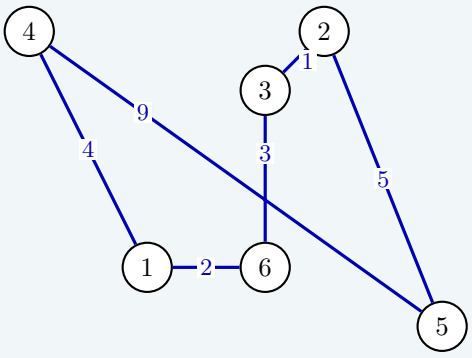
d_{ij}	1	2	3	4	5	6
1	—	5	4	4	5	2
2	5	—	1	5	5	4
3	4	1	—	4	5	3
4	4	5	4	—	9	6
5	5	5	5	9	—	3
6	2	4	3	6	3	—

L'istanza ha $\frac{5!}{2} = 60$ giri distinti; enumerandoli, l'unico giro ottimo è $(1,4,3,2,5,6)$, con $z(I) = 19$, e solo 7 giri hanno lunghezza minore di 24. Le euristiche di questa sezione danno:

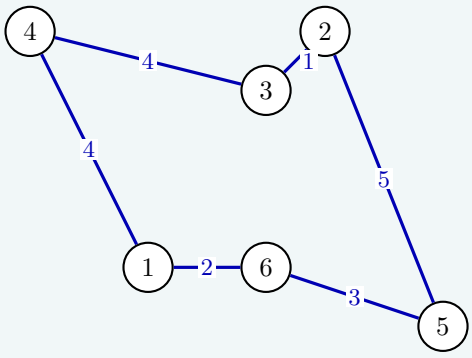
	città di partenza					
	1	2	3	4	5	6
NN	25	24	25	24	24	24

dalla città 1	L
NI	23
FI	19
CI	20

Il giro migliore del NN multi-start ha lunghezza 24, il 26% sopra l'ottimo, mentre FI trova il giro ottimo $(1,6,5,2,3,4)$ (il giro ottimo nel verso opposto):



NN multi-start (dalla città 2): $L = 24$



FI dalla città 1: $L = 19 = z(I)$

Tutti e sei i giri di NN contengono l'arco $(4,5)$, di lunghezza 9, il più lungo dell'istanza: le città 4 e 5 sono agli estremi opposti dell'istanza, e NN, da qualunque città parta, lascia almeno una delle due agli ultimi passi, quando l'unica strada rimasta è quella lunga.

2.8 Colorazione dei vertici: greedy e DSATUR

- Il problema di questa sezione è il problema della colorazione dei vertici (VCP): la sua definizione e le sue ipotesi sono nella sua pagina. Il grafo è $G = (V, E)$, e il valore ottimo è il numero cromatico $\chi(G)$.

2.8.1 La colorazione greedy

Il *Greedy Coloring Algorithm* (indicato con GC) è un'euristica costruttiva per il VCP. I colori sono gli interi $1, 2, 3, \ldots$; l'algoritmo esamina i vertici nell'ordine di una lista data λ e dà a ogni vertice il colore minimo non usato dai suoi vicini già colorati.

Algoritmo

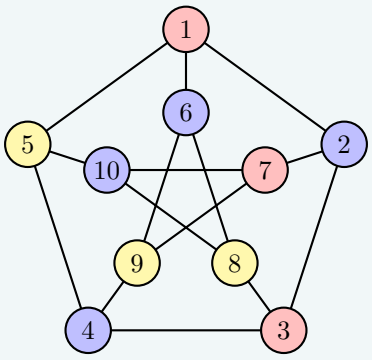
Algorithm: Colorazione greedy

Input: istanza del VCP $G = (V, E)$ con n vertici; lista $\lambda[1..n]$
Output: assegnamento vertici-colori $col[1..n]$ e numero di colori nc

Function *GreedyColoring*(n, E, λ)
  $col :=$ nuovo vettore$[1..n]$, con tutte le componenti 0 `// 0 = non ancora colorato`
  for $k := 1$ **to** n **do**
    $v := \lambda[k]$
    $U := \{\, col[u] : u \in N(v),\ col[u] \neq 0 \,\}$ `// colori dei vicini colorati`
    $col[v] \leftarrow \min\{\, c \in \mathbb{Z}_{\geq 1} : c \notin U \,\}$ `// colore libero minimo`
  end
  return col, $nc = \max_v col[v]$

- Le classi di colore $Q_c = \{v : col[v] = c\}$ sono insiemi stabili per costruzione, quindi GC restituisce sempre una colorazione ammissibile. Richiede tempo $O(n + |E|)$.
- Come $LIST$, GC dipende dall'ordine della lista, e l'ordine può fare una grande differenza.

Esempio 2.12: *GC* **sul grafo di Petersen, due ordini**



Lista $\lambda = (1, 2, \ldots, 10)$**.** I colori dei vicini colorati U e il colore assegnato sono:

v	1	2	3	4	5	6	7	8	9	10
U	$\emptyset$	$\{1\}$	$\{2\}$	$\{1\}$	$\{1,2\}$	$\{1\}$	$\{2\}$	$\{1,2\}$	$\{1,2\}$	$\{1,3\}$
$col[v]$	1	2	1	2	3	2	1	3	3	2

GC usa 3 colori (rosso = 1, blu = 2, giallo = 3 nella figura). Il 5-ciclo esterno $1, 2, 3, 4, 5$ è un ciclo dispari e richiede 3 colori, quindi $\chi(G) = 3$ e questa colorazione è ottima.

Lista $\lambda = (1, 2, 3, 4, 5, 7, 8, 6, 9, 10)$. I primi cinque vertici ricevono gli stessi colori di prima; poi

v	7	8	6	9	10
U	$\{2\}$	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$	$\{1,2,3\}$
$col[v]$	1	2	3	4	4

e GC usa 4 colori: il vertice 8 prende il colore 2, che il primo ordine dava al vertice 6, e i vertici 9 e 10 vedono allora tutti e tre i colori fra i loro vicini.

2.8.2 DSATUR

Il *DSATUR Algorithm* (Degree of SATURation, indicato con *DSATUR*) è un'euristica costruttiva per il VCP. Non usa una lista fissa: a ogni passo colora, con il colore libero minimo, un vertice non colorato di **grado di saturazione** massimo, cioè con il maggior numero di colori *diversi* fra i suoi vicini; i pareggi si risolvono con il grado massimo nel sottografo dei vertici non colorati, e poi con l'indice minimo.

Algoritmo

Algorithm: DSATUR

Input: istanza del VCP $G = (V, E)$ con n vertici
Output: assegnamento vertici-colori $col[1..n]$ e numero di colori nc

Function *DSATUR*(n, E)
  $col :=$ nuovo vettore$[1..n]$, con tutte le componenti 0
  for $k := 1$ **to** n **do**
    for *ogni u non colorato* **do**
      $sat[u] \leftarrow |\{\, col[w] : w \in N(u),\ col[w] \neq 0 \,\}|$ `// grado di saturazione`
      $du[u] \leftarrow |\{\, w \in N(u) : col[w] = 0 \,\}|$ `// grado fra i non colorati`
    end
    $v :=$ il vertice non colorato che massimizza $(sat[v], du[v])$ in ordine lessicografico (indice minimo)
    $col[v] \leftarrow \min\{\, c \in \mathbb{Z}_{\geq 1} : c \neq col[w]\ \forall w \in N(v) \,\}$
  end
  return col, $nc = \max_v col[v]$

- **Perché il grado di saturazione.** Un vertice con molti colori diversi intorno ha pochi colori rimasti: se lo rimandiamo, può restare senza nessuno dei colori già in uso. *DSATUR*

colora prima i vertici più vincolati, la stessa idea di "prima gli oggetti più difficili" di FFD e LPT, ma aggiornata dinamicamente dopo ogni assegnamento.

- $DSATUR$ è GC con una lista costruita al volo, quindi ogni proprietà di GC che vale per ogni lista vale anche per $DSATUR$. Un'implementazione diretta richiede tempo $O(n^2)$.

Esempio 2.13: $DSATUR$ sul grafo di Petersen

Tutti i vertici hanno grado 3. Il vertice scelto a ogni passo, il suo grado di saturazione, il suo grado fra i vertici non colorati e il suo colore sono:

passo	1	2	3	4	5	6	7	8	9	10
v	1	2	3	4	5	6	8	7	9	10
$sat[v]$	0	1	1	1	2	1	2	1	2	2
$du[v]$	3	2	2	2	1	2	1	2	0	0
$col[v]$	1	2	1	2	3	2	3	1	3	2

Al passo 1 tutti i vertici sono in pareggio e si sceglie il vertice 1. Al passo 2 i vertici 2, 5, 6 hanno grado di saturazione 1 e due vicini non colorati ciascuno: si sceglie l'indice minimo, 2. Al passo 7 il vertice 8 (vicini 3 e 6, colori 1 e 2) ha grado di saturazione 2 e passa prima del vertice 7. $DSATUR$ usa $3 = \chi(G)$ colori, la stessa colorazione di GC con la lista $(1, 2, \ldots, 10)$.

2.8.3 Che cosa garantisce la colorazione greedy

Proposizione 2.5

Per ogni grafo G e ogni lista λ,

$$GC(I) \;\leq\; \Delta(G) + 1.$$

Dimostrazione. Quando si colora il vertice v, l'insieme U dei colori dei suoi vicini colorati ha al più $\deg(v) \leq \Delta(G)$ elementi, quindi uno dei colori $1, 2, \ldots, \Delta(G) + 1$ non è in U, e il colore libero minimo è al più $\Delta(G) + 1$. □

- Il bound vale anche per $DSATUR$, che è GC con una lista particolare, ed è raggiunto dai grafi completi ($\Delta = n - 1$, $\chi = n$) e dai cicli dispari ($\Delta = 2$, $\chi = 3$), qualunque sia la lista.

- *Non* è una garanzia di prestazione: confronta $GC(I)$ con $\Delta(G) + 1$, non con $\chi(G)$, e $\Delta(G) + 1$ può essere molto più grande di $\chi(G)$. La prossima proposizione mostra che una cattiva lista può davvero imporre così tanti colori a un grafo che ne richiede solo due.

- D'altra parte, **qualche** lista è sempre ottima: se $Q_1, \ldots, Q_{\chi(G)}$ sono le classi di colore di una colorazione ottima, GC con prima i vertici di Q_1, poi quelli di Q_2, e così via, dà a ogni vertice di Q_c un colore al più c, quindi usa al più $\chi(G)$ colori. La difficoltà sta tutta nel trovare la lista.

Proposizione 2.6

Per ogni $k \geq 3$ esistono un grafo G_k con $n = 2k$ vertici, $\chi(G_k) = 2$ e $\Delta(G_k) = k - 1$, e una lista λ, tali che

$$GC(I_k) \;=\; k \;=\; \frac{n}{2}.$$

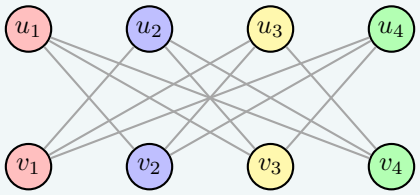
Dimostrazione. G_k è il **grafo corona**: i vertici sono $u_1, \ldots, u_k$ e $v_1, \ldots, v_k$, e u_i è adiacente a v_j se e solo se $i \neq j$ (il grafo bipartito completo $K_{k,k}$ privato di un matching perfetto). È bipartito, con almeno un arco, quindi $\chi(G_k) = 2$; ogni vertice ha grado $k - 1$. La lista è $\lambda = (u_1, v_1, u_2, v_2, \ldots, u_k, v_k)$.

Mostriamo per induzione su i che GC dà il colore i sia a u_i sia a v_i. Per $i = 1$ nessun vicino è colorato quando si esamina u_1, e v_1 non è adiacente a u_1: entrambi ricevono il colore 1. Quando si esamina u_i, i suoi vicini colorati sono $v_1, \ldots, v_{i-1}$, con colori $1, \ldots, i-1$ per induzione, quindi u_i riceve il colore i; quando si esamina v_i, i suoi vicini colorati sono $u_1, \ldots, u_{i-1}$ (u_i non è un vicino), con colori $1, \ldots, i-1$, quindi anche v_i riceve il colore i. Quindi $GC(I_k) = k$. □

- Sui grafi corona $\frac{GC(I_k)}{\chi(G_k)} = \frac{k}{2} = \frac{n}{4}$, che cresce senza limite: con una lista arbitraria, GC non ha una garanzia di prestazione costante. La lista $(u_1, \ldots, u_k, v_1, \ldots, v_k)$, prima le classi di colore, dà 2 colori.
- I grafi corona soddisfano le ipotesi del VCP per $k \geq 3$: nessun vertice isolato, e $N(u_i) = \{v_j : j \neq i\}$ non è contenuto in $N(u_{i'})$ né in alcun $N(v_j) \subseteq \{u_1, \ldots, u_k\}$.

Esempio 2.14: *GC* e *DSATUR* sul grafo corona G_4

Numeriamo i vertici $u_i = 2i - 1$ e $v_i = 2i$, così che $u_1, v_1, \ldots, u_4, v_4$ siano $1, 2, \ldots, 8$, e $E = \{\{2i-1, 2j\} : i \neq j\}$, con 12 archi.



Con la lista $(1, 2, \ldots, 8) = (u_1, v_1, \ldots, u_4, v_4)$, GC usa 4 colori, uno per ogni coppia $\{u_i, v_i\}$ (come in figura); con la lista $(1, 3, 5, 7, 2, 4, 6, 8)$ usa 2 colori.

DSATUR sceglie il vertice 1 (u_1, colore 1), poi i vertici 4, 5, 2, 3, 6, 7, 8, con colori $2, 1, 2, 1, 2, 1, 2$: i vertici u_i ricevono il colore 1 e i vertici v_i il colore 2, e *DSATUR* usa $2 = \chi(G_4)$ colori. Dopo il primo passo, ogni vertice adiacente a un vertice colorato ha grado di saturazione 1, e il vertice scelto sta sempre dal lato opposto all'ultimo colorato.

2.8.4 DSATUR come euristica

- Sui grafi bipartiti $DSATUR$ usa sempre al più 2 colori, come sui grafi corona; questo è dimostrato nell'articolo che lo ha introdotto (Brélaz, 1979).

- Sui grafi generali $DSATUR$ non ha una garanzia di prestazione, poiché il VCP non è approssimabile. È comunque una delle euristiche costruttive più usate per il VCP, perché sulla maggior parte dei grafi usa pochi colori in più di $\chi(G)$, spesso nessuno; è anche la base di algoritmi esatti di branch-and-bound per il VCP, che fanno branching sul vertice scelto dalla regola di $DSATUR$.

2.9 Costruzione randomizzata

- Un'euristica greedy è deterministica: eseguita due volte sulla stessa istanza, restituisce la stessa soluzione. Il NN multi-start ottiene soluzioni diverse cambiando la città di partenza, ma il numero di partenze è limitato a n. Un modo più flessibile per ottenere molte soluzioni diverse, e comunque buone, è **randomizzare la scelta**: a ogni passo, invece del candidato migliore, si sceglie a caso uno dei candidati *buoni*.

- I candidati buoni formano la **lista ristretta dei candidati** (restricted candidate list, RCL). Per un criterio di minimo con valori $g(q)$ dei candidati q, siano $g^{\min}$ e $g^{\max}$ il valore minimo e il valore massimo; la RCL *basata sul valore* con parametro $\alpha \in [0,1]$ è

$$RCL \;=\; \Big\{ q \;:\; g(q) \;\le\; g^{\min} + \alpha\,\big(g^{\max} - g^{\min}\big) \Big\}.$$

Con $\alpha = 0$ la RCL contiene solo i candidati migliori, e la costruzione è greedy (con pareggi risolti a caso); con $\alpha = 1$ contiene tutti i candidati, e la costruzione è puramente casuale.

- Un'alternativa è la RCL *basata sulla cardinalità*, formata dai k candidati migliori per un k fissato. La costruzione randomizzata è il primo ingrediente della metaeuristica GRASP (T.A. Feo, M.G.C. Resende, *Greedy randomized adaptive search procedures*, Journal of Global Optimization 6 (1995) 109–133).

- **Numeri casuali.** Per rendere riproducibili esempi ed esercizi, le note sugli algoritmi euristici usano il generatore *minimal standard* di Park e Miller, che produce una successione di numeri $u_1, u_2, \ldots$ in $(0,1)$ a partire da un **seme** intero:

$$seed \;\leftarrow\; (16807 \cdot seed) \bmod 2147483647, \qquad\qquad u \;=\; \frac{seed}{2147483647}.$$

- Il modulo è $2^{31} - 1$, un numero primo; con interi a 64 bit il prodotto non va mai in overflow. Lo stesso seme dà la stessa successione su ogni computer e in ogni linguaggio di programmazione. Un candidato si estrae da una RCL con $|RCL|$ elementi, elencati per indice crescente, prendendo l'elemento in posizione $\lfloor u \cdot |RCL| \rfloor + 1$; si usa un numero casuale solo quando $|RCL| \ge 2$.

L'*algoritmo nearest neighbour randomizzato* (indicato con RNN_α) è un'euristica costruttiva per il TSP. Partendo dalla città s, a ogni passo costruisce la RCL delle città non visitate la cui distanza dall'ultima città è al più $d^{\min} + \alpha\,(d^{\max} - d^{\min})$, e accoda una città estratta in modo uniforme a caso dalla RCL.

Algoritmo

Algorithm: Nearest Neighbour randomizzato

Input: istanza del TSP con n città, distanze $d[1..n][1..n]$; città di partenza s; $\alpha \in [0,1]$; *seed*

Output: giro $\sigma[1..n]$

Function *RandomizedNN*$(n, d, s, \alpha, seed)$
  $\sigma[1] \leftarrow s$, $visited := \{s\}$
  for $len := 2$ **to** n **do**
    $last := \sigma[len-1]$
    $d^{\min} := \min_{j \notin visited} d[last][j]$, $d^{\max} := \max_{j \notin visited} d[last][j]$
    $RCL := \{\, j \notin visited : d[last][j] \le d^{\min} + \alpha\,(d^{\max} - d^{\min}) \,\}$ `// per indice`
    if $|RCL| \ge 2$ **then**
      $u :=$ ParkMiller(*seed*) `// numero casuale in` $(0,1)$
      $k :=$ l'elemento in posizione $\lfloor u \cdot |RCL| \rfloor + 1$ di RCL
    else
      $k :=$ l'unico elemento di RCL
    end
    $\sigma[len] \leftarrow k$, $visited \leftarrow visited \cup \{k\}$
  end
  return σ

- Ripetendo la costruzione con lo stesso generatore — il seme *non* si reinizializza fra due esecuzioni — si ottengono giri diversi; si tiene il migliore. Il parametro α bilancia qualità e varietà: un α piccolo dà giri vicini al giro di NN, un α grande dà giri molto diversi ma di solito scadenti.

Esempio 2.15: $RNN_{0.3}$ sull'istanza di riferimento del TSP

Eseguiamo RNN_α quattro volte dalla città 1, con $\alpha = 0.3$ e seme iniziale 1. Il generatore dà

u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	u_9
0.0000	0.1315	0.7556	0.4587	0.5328	0.2190	0.0470	0.6789	0.6793

($u_1 = \frac{16807}{2147483647} \approx 0.0000078$). Dalla città 1 le distanze verso $2,3,4,5$ sono $5,2,2,9$, quindi la soglia è $2 + 0.3 \cdot (9-2) = 4.1$ e $RCL = \{3,4\}$.

Esecuzione 1. u_1: posizione $\lfloor 0.0000 \cdot 2 \rfloor + 1 = 1$, città 3. Da 3 le distanze verso $2,4,5$ sono $4,4,7$, soglia $4 + 0.3 \cdot 3 = 4.9$, $RCL = \{2,4\}$; u_2: posizione 1, città 2. Da 2 le distanze verso $4,5$ sono $3,4$, soglia 3.3, $RCL = \{4\}$: città 4, poi città 5. Giro $(1,3,2,4,5)$, $L = 25$: il giro di NN.

Esecuzione 2. u_3: posizione $\lfloor 0.7556 \cdot 2 \rfloor + 1 = 2$, città 4. Da 4 le distanze verso $2, 3, 5$ sono $3, 4, 7$, soglia $3 + 0.3 \cdot 4 = 4.2$, $RCL = \{2, 3\}$; u_4: posizione 1, città 2. Da 2 le distanze verso $3, 5$ sono $4, 4$: $d^{\min} = d^{\max} = 4$ e $RCL = \{3, 5\}$; u_5: posizione $\lfloor 0.5328 \cdot 2 \rfloor + 1 = 2$, città 5; poi città 3. Giro $(1, 4, 2, 5, 3)$, $L = 18 = z(I)$.

Esecuzione 3. u_6: città 3; u_7: città 2; poi le città 4 e 5. Giro $(1, 3, 2, 4, 5)$, $L = 25$.

Esecuzione 4. u_8: città 4; $RCL = \{2, 3\}$ e u_9: posizione 2, città 3. Da 3 le distanze verso $2, 5$ sono $4, 7$, soglia 4.9, $RCL = \{2\}$: città 2, poi città 5. Giro $(1, 4, 3, 2, 5)$, $L = 2 + 4 + 4 + 4 + 9 = 23$.

Le quattro esecuzioni danno $25, 18, 25, 23$: la seconda trova il giro ottimo, che NN dalla città 1 non trova. La randomizzazione ha trasformato un'unica risposta deterministica in una piccola ricerca.

2.10 Look-ahead e regret

- Un criterio greedy chiede: *quale scelta è la migliore adesso?* La sua miopia si può ridurre ponendo due domande diverse:
  - **regret**: *che cosa perdo se rimando questa scelta?* Un elemento che ha una sola buona opzione va servito prima che la perda; un elemento con molte opzioni equivalenti può aspettare;
  - **look-ahead**: *dove porta questa scelta?* Ogni candidato si valuta completando la soluzione parziale con un'euristica veloce, e si sceglie il candidato con la migliore soluzione completata.

  Entrambe le idee costano più tempo per passo del semplice criterio greedy, ed entrambe mantengono la caratteristica essenziale di un'euristica costruttiva: la scelta fatta è definitiva.

2.10.1 Inserimento con regret

- Per un'euristica di inserimento, siano $\Delta_q^1 \leq \Delta_q^2$ i due costi di inserimento minimi della città q sugli archi del sottogiro corrente (con $\Delta_q^2 = \Delta_q^1$ se il sottogiro ha un solo arco). Il **regret** di q è

$$reg_q \;=\; \Delta_q^2 - \Delta_q^1 \;\geq\; 0 :$$

il costo aggiuntivo pagato per q se il suo arco migliore viene usato da un'altra città prima che q sia inserita.

L'*algoritmo di inserimento regret-2* (indicato con REG_2) è un'euristica costruttiva per il TSP. Partendo dal sottogiro (s), a ogni passo inserisce la città con il *regret massimo*, in caso di pareggio quella con Δ_q^1 minimo e poi con l'indice minimo, nel suo arco di costo di inserimento minimo.

Algoritmo

Algorithm: Inserimento regret-2

Input: istanza del TSP con n città, distanze $d[1..n][1..n]$; città di partenza s
Output: giro $\sigma[1..n]$

Function *Regret2Insertion*(n, d, s)
  $\sigma := (s)$
  for $len := 1$ **to** $n - 1$ **do**
    foreach *città* $q \notin \sigma$ **do**
      $\Delta_q^1, \Delta_q^2 :=$ i due valori minimi di $\Delta(\sigma[a], q, \sigma[a+1])$, $a = 1, \ldots, len$
      $reg_q := \Delta_q^2 - \Delta_q^1$ `// regret di` q
    end
    $k :=$ la città con reg_q massimo (pareggi: Δ_q^1 minimo, poi indice minimo)
    inserire k nel suo arco di costo di inserimento minimo
  end
  return σ

- Le euristiche regret-k generalizzano l'idea, sommando le differenze fra l'opzione migliore e le $k-1$ successive; sono molto usate per i problemi di vehicle routing, dove un cliente può avere una sola route ammissibile rimasta (J.-Y. Potvin, J.-M. Rousseau, *A parallel route building algorithm for the vehicle routing and scheduling problem with time windows*, European Journal of Operational Research 66 (1993) 331–340). La stessa idea si applica a qualunque costruzione di tipo assegnamento, come oggetti a contenitori.

- Finché tutti i regret sono 0, la regola di risoluzione dei pareggi fa comportare REG_2 come CI.

Esempio 2.16: REG_2 contro CI sull'istanza con 6 città

Sull'istanza dell'Esempio 2.11, dalla città 1. Nei primi due passi tutti i regret sono 0 (i sottogiri (1) e (1, 6) hanno un arco, o due copie dello stesso arco), e sia REG_2 sia CI inseriscono la città 6 (costo 4) e poi la città 3 (costo 5), ottenendo $(1, 3, 6)$ di lunghezza 9. Al terzo passo i costi di inserimento sugli archi $(1, 3)$, $(3, 6)$, $(6, 1)$ sono

città q	costi	Δ_q^1	Δ_q^2	reg_q
2	$(2, 2, 7)$	2	2	0
4	$(4, 7, 8)$	4	7	3
5	$(6, 5, 6)$	5	6	1

CI inserisce la città 2, la più economica, sull'arco $(1, 3)$. REG_2 inserisce la città 4, quella con il regret massimo, sull'arco $(1, 3)$: questo arco è l'unica buona posizione per la città 4, mentre la città 2 ha due posizioni ugualmente buone e può aspettare. Le due costruzioni proseguono poi così:

	CI	REG_2
passo 3	2 su $(1,3)$, costo 2: $(1,2,3,6)$	4 su $(1,3)$, costo 4: $(1,4,3,6)$
passo 4	4 su $(1,2)$, costo 4: $(1,4,2,3,6)$	5 su $(3,6)$, costo 5: $(1,4,3,5,6)$
passo 5	5 su $(3,6)$, costo 5: $(1,4,2,3,5,6)$	2 su $(3,5)$, costo 1: $(1,4,3,2,5,6)$
lunghezza	$4+5+2+4+5=20$	$4+5+4+5+1=19$

Al passo 4 di REG_2 i costi della città 2 sugli archi di $(1,4,3,6)$ sono $(6,2,2,7)$, regret 0, e quelli della città 5 sono $(10,10,5,6)$, regret 1: la città 5 passa prima. La città 2, rimandata due volte, si inserisce infine per 1 invece che per 2, fra le città 3 e 5. $REG_2(I)=19=z(I)$, mentre $CI(I)=20$.

2.10.2 Look-ahead: il metodo pilota

Il *metodo pilota* trasforma un'euristica costruttiva $\mathscr{H}$, il *pilota*, in un'euristica più forte. A ogni passo prova ogni candidato: la soluzione parziale estesa con il candidato viene completata da $\mathscr{H}$, e il candidato la cui soluzione completata è la migliore si aggiunge in modo definitivo. Con NN come pilota (indicato con $PILOT\text{-}NN$), i candidati sono le città successive del giro.

Algoritmo

Algorithm: Metodo pilota con nearest neighbour

Input: istanza del TSP con n città, distanze $d[1..n][1..n]$; città di partenza s
Output: giro $\sigma[1..n]$

```
1  Function PilotNN(n, d, s)
2      σ := (s)
3      for len := 2 to n do
4          foreach città q ∉ σ do
5              τ_q := il cammino σ esteso con q e completato da NN        // guardare avanti
6          end
7          k := la città q con L(τ_q) minimo (indice minimo)
8          accodare k a σ                                  // la scelta è definitiva
9      end
10     return σ
```

- Il metodo pilota è dovuto a C. Duin e S. Voß (*The pilot method: a strategy for heuristic repetition with application to the Steiner problem in graphs*, Networks 34 (1999) 181–191). Ogni passo esegue il pilota una volta per candidato: con NN come pilota il tempo di esecuzione è $O(n^4)$ invece di $O(n^2)$.

- **Il metodo pilota non è mai peggiore del suo pilota.** Al primo passo, il candidato scelto da NN è fra quelli provati, e il suo completamento è il giro di NN da s, quindi il

miglior completamento trovato ha lunghezza al più $NN(I)$. A ogni passo successivo, il completamento del candidato scelto è di nuovo fra i completamenti provati al passo seguente (il candidato che NN accoderebbe), quindi la lunghezza del miglior completamento non aumenta mai. Il giro finale è l'ultimo completamento, quindi $PILOT\text{-}NN(I) \leq NN(I)$ dalla stessa città di partenza.

Esempio 2.17: *PILOT-NN* sull'istanza di riferimento del TSP

Dalla città 1, i candidati e i loro completamenti con NN sono:

cammino parziale	candidati: giro completato e lunghezza	scelta
(1)	2: $(1,2,4,3,5)$ 28; 3: $(1,3,2,4,5)$ 25; 4: $(1,4,2,3,5)$ 25; 5: $(1,5,2,4,3)$ 22	5
$(1,5)$	2: $(1,5,2,4,3)$ 22; 3: $(1,5,3,2,4)$ 25; 4: $(1,5,4,2,3)$ 25	2
$(1,5,2)$	3: $(1,5,2,3,4)$ 23; 4: $(1,5,2,4,3)$ 22	4
$(1,5,2,4)$	3: $(1,5,2,4,3)$ 22	3

$PILOT\text{-}NN$ va prima nella città 5, la *più lontana* dalla città 1, cosa che NN non farebbe mai, e restituisce $(1,5,2,4,3)$ con $L = 9+4+3+4+2 = 22$, contro $NN(I) = 25$ dalla stessa città. L'ottimo 18 non si raggiunge: il look-ahead è buono solo quanto il pilota che lo esegue. Sull'istanza con 6 città dell'Esempio 2.11, $PILOT\text{-}NN$ dalla città 1 dà 20, contro 25 per NN.

2.11 Che cosa danno le euristiche costruttive, e che cosa non danno

- La tabella raccoglie i risultati sulle quattro istanze di riferimento, con i bound duali del capitolo «Algoritmi euristici: definizioni» (UB per il KP, LB per gli altri problemi) e il gap della migliore soluzione euristica.

problema	euristica	valori	bound duale	gap	$z(I)$
KP	GR_p, GR, GR_w	17, 16, 10	UB = 17	0%	17
BPP	FF, FFD, BFD	4, 3, 3	LB = 3	0%	3
$P\,\|\|\,C_{\max}$	$LIST$ (lista inversa), LPT	12, 11	LB = 9	22.2%	9
TSP	NN, multi-start NN, NI, FI, CI	25, 18, 18, 18, 18	LB = 17	5.9%	18

- **Un bound primale, e niente di più.** Un'euristica costruttiva restituisce una soluzione ammissibile, e il suo valore è un bound primale. Per il KP e il BPP il miglior valore euristico coincide con il bound duale, e la soluzione è *dimostrata* ottima; per il TSP il giro di lunghezza 18 è ottimo, ma il bound duale 17 non può dimostrarlo; per $P\,||\,C_{\max}$ il gap del 22.2% è reale. L'euristica stessa non sa mai in quale di queste situazioni si trova.

- **Una scelta non si rivede mai.** Ogni euristica di queste note si ferma appena la soluzione è completa, e ogni decisione presa lungo il percorso è definitiva. Nello schedule di LPT, il lavoro 7 va sulla macchina 1 quando i tre carichi sono uguali, e il makespan 11 è fissato in quel momento; nel giro di NN dalla città 1, l'arco $(5, 1)$ di lunghezza 9 è la conseguenza delle primissime scelte. Guardando il diagramma di Gantt o il giro, una soluzione migliore sembra vicina; ma un'euristica costruttiva non ha alcun meccanismo per tornare indietro.

- **L'ordine decide tutto.** Gli esempi di queste note mostrano la stessa lezione su quattro problemi: ordinare gli elementi nell'ordine giusto (efficienza, pesi decrescenti, prima i lavori più lunghi) è la principale scelta di progetto; tuttavia nessun ordine è il migliore su ogni istanza (Esempio 2.1, Esempio 2.4), ed eseguire più ordini, più punti di partenza o più costruzioni randomizzate, tenendo la soluzione migliore, è una protezione economica ed efficace.

- **Economiche, e utili come punto di partenza.** Le euristiche costruttive sono veloci, e raramente si usano da sole: le loro soluzioni sono i punti di partenza dei metodi che cercano di migliorare una soluzione completa.

Un'euristica costruttiva costruisce una soluzione, un elemento alla volta, scegliendo a ogni passo l'elemento che sembra migliore adesso. L'ordine delle scelte decide la qualità del risultato, e una scelta, una volta fatta, non si rivede mai.

2.12 Errori tipici

Attenzione

- **Aspettarsi che il greedy per efficienza sia ottimo.** Può essere arbitrariamente cattivo, e così ciascuno degli altri due ordinamenti: le note contengono una famiglia di istanze per ognuno di essi.
- **Credere che ordinare aiuti sempre.** FFD può essere peggiore di FF (Esempio 2.4).
- **Risolvere i pareggi nel modo sbagliato.** Ogni esempio usa una regola fissata (indice minimo, primo contenitore in cui l'oggetto entra, contenitore più pieno per BF): cambiarla cambia la soluzione.
- **Dimenticare l'arco di chiusura.** La lunghezza di un giro include l'arco dall'ultima città di ritorno alla prima; il costo di un inserimento è $d_{ik} + d_{kj} - d_{ij}$, e va provata ogni posizione, arco di chiusura compreso.
- **Leggere $\mathscr{A}(I)$ come un bound su $z(I)$.** È solo un bound primale.

2.13 In sintesi

In sintesi

- Un'euristica costruttiva costruisce una soluzione, un elemento alla volta, e non rivede mai una scelta.
- L'ordine delle scelte è la principale scelta di progetto: efficienza per il KP, pesi decrescenti per il BPP, prima i lavori più lunghi per $P\,||\,C_{\max}$; nessun ordine vince su ogni istanza.
- Più ordini, punti di partenza o costruzioni randomizzate, tenendo la soluzione migliore, sono una protezione economica.
- Le soluzioni sono punti di partenza per la ricerca locale.

Quadro delle istanze di riferimento. Miglior valore costruttivo e bound duale:

problema	miglior valore euristico	bound duale	gap	$z(I)$
KP (max)	17 (GR_p)	$\mathrm{UB}(I) = 17$	0%	17
BPP (min)	3 (FFD, BFD)	$\mathrm{LB}(I) = 3$	0%	3
$P\,||\,C_{\max}$ (min)	11 (LPT)	$\mathrm{LB}(I) = 9$	22.2%	9
TSP (min)	18 (multi-start NN, NI, FI, CI)	$\mathrm{LB}(I) = 17$	5.9%	18

2.14 Implementazione in C++

- La cartella `CONSTR_HEUR_CODE` contiene un'implementazione in `C++` delle euristiche di queste note, una funzione per file, con gli stessi nomi e gli stessi passi degli pseudocodici. Oggetti, contenitori, lavori, macchine e città sono indicizzati da 0; la logica algoritmica è invariata, e i pareggi si risolvono sempre a favore dell'indice minimo.
- Il comando `make` compila il programma, e `make test` esegue ogni esempio di queste note e ogni istanza degli esercizi, confrontando ogni valore e ogni soluzione con quelli calcolati da un'implementazione indipendente in Python: 202 controlli, 0 errori.
- L'header `Constr_heuristics.h` dichiara tutte le funzioni:

```
#ifndef CONSTR_HEURISTICS_HEADER
#define CONSTR_HEURISTICS_HEADER

// constructive heuristics for the KP, the BPP, P||Cmax and the TSP
// items, bins, jobs, machines and cities are indexed starting from 0
// ties are always broken in favour of the smallest index

// sorting
void KPOrder(int n, int* p, int* w, int criterion, int* order);   // 0 profit, 1 weight, 2 efficiency
void DecreasingOrder(int n, int* a, int* order);
```

```
11
12 // KP: returns the profit, x[i] = 1 if item i is selected
13 int GreedyOrderKP(int n, int c, int* w, int* p, int* order, int* x);
14
15 // BPP: returns the number of bins, bin[i] = bin of item i
16 int FirstFitDecreasing(int n, int c, int* w, int* bin);
17 int BestFitDecreasing(int n, int c, int* w, int* bin);
18
19 // P||Cmax: returns the makespan, mach[j] = machine of job j
20 int ListScheduling(int n, int m, int* t, int* order, int* mach);
21 int LPT(int n, int m, int* t, int* mach);
22
23 // TSP: d is the n x n distance matrix; every function returns the length of
       the tour
24 int TourLength(int n, int** d, int* tour);
25 int NearestNeighbour(int n, int** d, int s, int* tour);
26 int MultiStartNN(int n, int** d, int* tour);
27 int Insertion(int n, int** d, int s, int rule, int* tour);           // 0 NI, 1
       FI, 2 CI
28 int Regret2Insertion(int n, int** d, int s, int* tour);
29 int PilotNN(int n, int** d, int s, int* tour);
30
31 // randomized construction
32 double ParkMiller(long long* seed);
33 int RandomizedNN(int n, int** d, int s, int alpha_num, int alpha_den, long long
       * seed, int* tour);
34
35 #endif
```

2.14.1 Ordinamento

- Gli ordinamenti degli oggetti del KP (Sezione 2.4) e l'ordinamento decrescente usato da FFD, BFD e *LPT* si calcolano con l'insertion sort, che mantiene l'ordine degli indici fra chiavi uguali. Le efficienze si confrontano senza divisioni: $\frac{p_j}{w_j} > \frac{p_k}{w_k}$ se e solo se $p_j w_k > p_k w_j$.

```
1 // order[0..n-1] receives the items sorted by the criterion:
2 // 0 = non-increasing profit, 1 = non-decreasing weight, 2 = non-increasing
      efficiency p/w
3 // insertion sort: items with equal keys keep their index order
4
5 static bool Before(int i, int j, int* p, int* w, int criterion)
6 {
7     if (criterion == 0) return p[i] > p[j];
8     if (criterion == 1) return w[i] < w[j];
9     return p[i] * w[j] > p[j] * w[i];       // p_i/w_i > p_j/w_j without
      divisions
10 }
11
12 void KPOrder(int n, int* p, int* w, int criterion, int* order)
13 {
14     for (int k = 0; k < n; k++)
15     {
16         int i = k;                          // item to be inserted in order[0..k
      -1]
17         int pos = k;
```

```
        while (pos > 0 && Before(i, order[pos - 1], p, w, criterion))
        {
            order[pos] = order[pos - 1];
            pos--;
        }
        order[pos] = i;
    }
}
```

```
// order[0..n-1] receives the indices sorted by non-increasing a[i]
// insertion sort: indices with equal values keep their order

void DecreasingOrder(int n, int* a, int* order)
{
    for (int k = 0; k < n; k++)
    {
        int pos = k;
        while (pos > 0 && a[k] > a[order[pos - 1]])
        {
            order[pos] = order[pos - 1];
            pos--;
        }
        order[pos] = k;
    }
}
```

2.14.2 Il problema dello zaino 0/1

```
// the items are examined in the order given by order[0..n-1]
// x[i] stores the selection decision for item i
// the function returns the profit of the solution x

int GreedyOrderKP(int n, int c, int* w, int* p, int* order, int* x)
{
    for (int i = 0; i < n; i++)
    {
        x[i] = 0;                           // item i is initially not selected
    }
    int rc = c;                             // residual capacity
    int z = 0;                              // profit of the selected items

    for (int k = 0; k < n; k++)
    {
        int i = order[k];                   // k-th item in the order
        if (w[i] <= rc)                     // feasibility test
        {
            x[i] = 1;                       // select item i
            rc = rc - w[i];                 // update residual capacity
            z = z + p[i];
        }
    }
    return z;
}
```

2.14.3 Il problema del bin packing

```
// the items are sorted by non-increasing weight and packed by First Fit
// the function returns the number of bins used
// bin[i] stores the bin assigned to item i

int FirstFitDecreasing(int n, int c, int* w, int* bin)
{
    int* order = new int[n];
    DecreasingOrder(n, w, order);          // sort: largest items first

    int* rc = new int[n];                  // residual capacities (at most n
    bins)
    int nb = 0;                            // number of bins opened so far

    for (int k = 0; k < n; k++)
    {
        int i = order[k];                  // k-th largest item
        int s = -1;                        // selected bin (-1 means: not found
     yet)
        for (int b = 0; b < nb; b++)       // first bin in which item i fits
        {
            if (w[i] <= rc[b])
            {
                s = b;
                break;
            }
        }
        if (s == -1)                       // no bin fits: open a new one
        {
            rc[nb] = c;
            s = nb;
            nb++;
        }
        bin[i] = s;                        // pack item i into bin s
        rc[s] = rc[s] - w[i];
    }

    delete[] rc;
    delete[] order;
    return nb;
}
```

```
// the items are sorted by non-increasing weight and packed by Best Fit
// the function returns the number of bins used
// bin[i] stores the bin assigned to item i

int BestFitDecreasing(int n, int c, int* w, int* bin)
{
    int* order = new int[n];
    DecreasingOrder(n, w, order);          // sort: largest items first

    int* rc = new int[n];                  // residual capacities (at most n
    bins)
    int nb = 0;                            // number of bins opened so far

    for (int k = 0; k < n; k++)
    {
```

```
        int i = order[k];                       // k-th largest item
        int s = -1;                             // selected bin (-1 means: not found
    yet)
        for (int b = 0; b < nb; b++)            // fullest bin in which item i fits
        {
            if (w[i] <= rc[b] && (s == -1 || rc[b] < rc[s]))
            {
                s = b;
            }
        }
        if (s == -1)                            // no bin fits: open a new one
        {
            rc[nb] = c;
            s = nb;
            nb++;
        }
        bin[i] = s;                             // pack item i into bin s
        rc[s] = rc[s] - w[i];
    }

    delete[] rc;
    delete[] order;
    return nb;
}
```

2.14.4 Scheduling su macchine parallele

```
// the jobs are examined in the order given by order[0..n-1]
// each job is assigned to the machine with the smallest current load
// the function returns the makespan; mach[j] stores the machine of job j

int ListScheduling(int n, int m, int* t, int* order, int* mach)
{
    int* L = new int[m];                        // loads of the machines
    for (int h = 0; h < m; h++)
    {
        L[h] = 0;
    }

    for (int k = 0; k < n; k++)
    {
        int j = order[k];                       // k-th job of the list
        int s = 0;                              // least loaded machine
        for (int h = 1; h < m; h++)
        {
            if (L[h] < L[s]) s = h;
        }
        mach[j] = s;                            // assign job j to machine s
        L[s] = L[s] + t[j];
    }

    int Cmax = 0;                               // makespan = largest load
    for (int h = 0; h < m; h++)
    {
        if (L[h] > Cmax) Cmax = L[h];
    }
    delete[] L;
```

```
    return Cmax;
}
```

```
// Longest Processing Time first: list scheduling with the jobs
// sorted by non-increasing processing time
// the function returns the makespan; mach[j] stores the machine of job j

int LPT(int n, int m, int* t, int* mach)
{
    int* order = new int[n];
    DecreasingOrder(n, t, order);          // longest jobs first
    int Cmax = ListScheduling(n, m, t, order, mach);
    delete[] order;
    return Cmax;
}
```

2.14.5 Il problema del commesso viaggiatore

- Le distanze sono memorizzate in una matrice $n \times n$ `d`, e un giro in un vettore di n città.

```
// length of the tour tour[0], tour[1], ..., tour[n-1], tour[0]

int TourLength(int n, int** d, int* tour)
{
    int L = 0;
    for (int k = 0; k < n - 1; k++)
    {
        L = L + d[tour[k]][tour[k + 1]];
    }
    return L + d[tour[n - 1]][tour[0]];    // closing edge
}
```

```
// starting from city s, the tour is extended at its end with the
// nearest city not yet visited
// the function returns the length of the tour stored in tour[0..n-1]

int NearestNeighbour(int n, int** d, int s, int* tour)
{
    bool* visited = new bool[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
    }
    tour[0] = s;                           // the tour starts at city s
    visited[s] = true;

    for (int len = 1; len < n; len++)
    {
        int last = tour[len - 1];          // last city of the partial tour
        int k = -1;                        // nearest unvisited city
        for (int j = 0; j < n; j++)
        {
            if (!visited[j] && (k == -1 || d[last][j] < d[last][k])) k = j;
        }
        tour[len] = k;                     // append k at the end
        visited[k] = true;
    }
```

```
26
27      delete[] visited;
28      return TourLength(n, d, tour);
29  }
```

```
1   // runs the nearest neighbour heuristic from every city
2   // and keeps the shortest tour (the first one found, in case of ties)
3   // the function returns its length; the tour is stored in tour[0..n-1]
4
5   int MultiStartNN(int n, int** d, int* tour)
6   {
7       int* cur = new int[n];
8       int best = -1;                          // length of the best tour found
9       for (int s = 0; s < n; s++)
10      {
11          int L = NearestNeighbour(n, d, s, cur);
12          if (best == -1 || L < best)
13          {
14              best = L;
15              for (int k = 0; k < n; k++) tour[k] = cur[k];
16          }
17      }
18      delete[] cur;
19      return best;
20  }
```

- In `Insertion` le tre regole di selezione condividono lo stesso ciclo: il valore di un candidato è la sua distanza dal sottogiro (NI), la stessa distanza cambiata di segno, così che il minimo corrisponda alla città più lontana (FI), oppure il suo costo di inserimento minimo (CI).

```
1   // insertion heuristics: starting from the subtour (s), the city selected by
2   // the rule is inserted into the subtour where it increases the length the
        least
3   // rule: 0 = nearest insertion (NI), 1 = farthest insertion (FI), 2 = cheapest
        insertion (CI)
4   // the function returns the length of the tour stored in tour[0..n-1]
5
6   // cheapest position for city k in the subtour tour[0..len-1]:
7   // returns the increase of length, and in *pos the edge (tour[a], tour[a+1]) to
        break
8   static int BestPosition(int len, int** d, int* tour, int k, int* pos)
9   {
10      int best = -1;
11      for (int a = 0; a < len; a++)
12      {
13          int i = tour[a], j = tour[(a + 1) % len];
14          int cost = d[i][k] + d[k][j] - d[i][j];
15          if (best == -1 || cost < best)
16          {
17              best = cost;
18              *pos = a;
19          }
20      }
21      return best;
22  }
```

```
23
24 int Insertion(int n, int** d, int s, int rule, int* tour)
25 {
26     bool* in = new bool[n];                 // in[i] = true if city i is in the
       subtour
27     for (int i = 0; i < n; i++)
28     {
29         in[i] = false;
30     }
31     tour[0] = s;                            // the subtour (s)
32     in[s] = true;
33
34     for (int len = 1; len < n; len++)
35     {
36         // selection step: city k to be inserted
37         int k = -1, keyk = 0, pos = 0;
38         for (int q = 0; q < n; q++)
39         {
40             if (in[q]) continue;
41             int key;                        // value of city q for the rule
42             if (rule == 2)
43             {
44                 int a;
45                 key = BestPosition(len, d, tour, q, &a);   // insertion cost
46             }
47             else
48             {
49                 key = d[q][tour[0]];        // distance from the subtour
50                 for (int a = 1; a < len; a++)
51                 {
52                     if (d[q][tour[a]] < key) key = d[q][tour[a]];
53                 }
54                 if (rule == 1) key = -key; // farthest: maximize the distance
55             }
56             if (k == -1 || key < keyk)
57             {
58                 k = q;
59                 keyk = key;
60             }
61         }
62         // insertion step: k goes where it increases the length the least
63         BestPosition(len, d, tour, k, &pos);
64         for (int a = len; a > pos + 1; a--)
65         {
66             tour[a] = tour[a - 1];
67         }
68         tour[pos + 1] = k;
69         in[k] = true;
70     }
71
72     delete[] in;
73     return TourLength(n, d, tour);
74 }
```

2.14.6 Costruzione randomizzata

- La soglia della RCL si verifica in aritmetica intera, con α dato come frazione $\frac{\alpha_{num}}{\alpha_{den}}$, così che nessun errore di arrotondamento possa cambiare la lista.

```
// Park-Miller minimal standard generator:
// seed <- 16807 * seed mod (2^31 - 1), and u = seed / (2^31 - 1) in (0,1)

double ParkMiller(long long* seed)
{
    *seed = (16807LL * (*seed)) % 2147483647LL;
    return (double)(*seed) / 2147483647.0;
}
```

```
// randomized nearest neighbour with a value-based restricted candidate list:
// RCL = { j unvisited : d[last][j] <= dmin + alpha (dmax - dmin) },
// with alpha = alpha_num / alpha_den; the next city is drawn uniformly from the RCL
// (a random number is used only when the RCL has at least two cities)
// the function returns the length of the tour stored in tour[0..n-1]

int RandomizedNN(int n, int** d, int s, int alpha_num, int alpha_den,
                 long long* seed, int* tour)
{
    bool* visited = new bool[n];
    int* RCL = new int[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
    }
    tour[0] = s;
    visited[s] = true;

    for (int len = 1; len < n; len++)
    {
        int last = tour[len - 1];
        int dmin = -1, dmax = -1;          // smallest and largest distance from last
        for (int j = 0; j < n; j++)
        {
            if (visited[j]) continue;
            if (dmin == -1 || d[last][j] < dmin) dmin = d[last][j];
            if (dmax == -1 || d[last][j] > dmax) dmax = d[last][j];
        }
        int size = 0;                      // restricted candidate list
        for (int j = 0; j < n; j++)
        {
            // d <= dmin + alpha (dmax - dmin), in integer arithmetic
            if (!visited[j] && alpha_den * d[last][j] <= alpha_den * dmin + alpha_num * (dmax - dmin))
            {
                RCL[size] = j;
                size++;
            }
        }
        int k = RCL[0];
        if (size > 1)
```

```
        {
            double u = ParkMiller(seed);    // random number in (0,1)
            k = RCL[(int)(u * size)];       // uniform choice in the RCL
        }
        tour[len] = k;
        visited[k] = true;
    }

    delete[] RCL;
    delete[] visited;
    return TourLength(n, d, tour);
}
```

2.14.7 Regret e look-ahead

```
// regret-2 insertion: starting from the subtour (s), insert the city with the largest
// regret = (second cheapest insertion cost) - (cheapest insertion cost),
// ties broken by the smaller cheapest cost, then by the smaller index,
// at its cheapest position
// the function returns the length of the tour stored in tour[0..n-1]

int Regret2Insertion(int n, int** d, int s, int* tour)
{
    bool* in = new bool[n];
    for (int i = 0; i < n; i++)
    {
        in[i] = false;
    }
    tour[0] = s;
    in[s] = true;

    for (int len = 1; len < n; len++)
    {
        int k = -1, regk = 0, c1k = 0, posk = 0;
        for (int q = 0; q < n; q++)
        {
            if (in[q]) continue;
            int c1 = -1, c2 = -1, pos = 0; // cheapest and second cheapest cost
            for (int a = 0; a < len; a++)
            {
                int i = tour[a], j = tour[(a + 1) % len];
                int cost = d[i][q] + d[q][j] - d[i][j];
                if (c1 == -1 || cost < c1)
                {
                    c2 = c1;
                    c1 = cost;
                    pos = a;
                }
                else if (c2 == -1 || cost < c2)
                {
                    c2 = cost;
                }
            }
            if (c2 == -1) c2 = c1;          // a single position: no regret
            int reg = c2 - c1;
            if (k == -1 || reg > regk || (reg == regk && c1 < c1k))
```

```
            {
                k = q;
                regk = reg;
                c1k = c1;
                posk = pos;
            }
        }
        for (int a = len; a > posk + 1; a--)   // insert k after position posk
        {
            tour[a] = tour[a - 1];
        }
        tour[posk + 1] = k;
        in[k] = true;
    }

    delete[] in;
    return TourLength(n, d, tour);
}
```

```
// pilot method with the nearest neighbour heuristic as pilot:
// every candidate next city is evaluated by completing the partial tour
// with nearest neighbour; the candidate with the shortest completed tour is appended
// the function returns the length of the tour stored in tour[0..n-1]

int PilotNN(int n, int** d, int s, int* tour)
{
    bool* visited = new bool[n];
    int* trial = new int[n];                // partial tour completed by the pilot
    bool* used = new bool[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
    }
    tour[0] = s;
    visited[s] = true;

    for (int len = 1; len < n; len++)
    {
        int k = -1, bestL = -1;
        for (int q = 0; q < n; q++)         // candidate next city q
        {
            if (visited[q]) continue;
            for (int i = 0; i < n; i++) used[i] = visited[i];
            for (int a = 0; a < len; a++) trial[a] = tour[a];
            trial[len] = q;
            used[q] = true;
            for (int t = len + 1; t < n; t++)  // pilot: nearest neighbour completion
            {
                int last = trial[t - 1], next = -1;
                for (int j = 0; j < n; j++)
                {
                    if (!used[j] && (next == -1 || d[last][j] < d[last][next])) next = j;
                }
                trial[t] = next;
```

```
36                  used[next] = true;
37              }
38              int L = TourLength(n, d, trial);
39              if (k == -1 || L < bestL)
40              {
41                  k = q;
42                  bestL = L;
43              }
44          }
45          tour[len] = k;                      // append the best candidate
46          visited[k] = true;
47      }
48
49      delete[] used;
50      delete[] trial;
51      delete[] visited;
52      return TourLength(n, d, tour);
53  }
```

Capitolo 3

Ricerca locale

Prerequisiti

- Da questo modulo, il capitolo «Algoritmi euristici: definizioni»: intorni, ottimi locali, intensificazione e diversificazione.
- Da questo modulo, il capitolo «Euristiche costruttive»: le soluzioni di partenza usate qui.

Obiettivi

Alla fine di questo capitolo saremo in grado di:

- definire gli intorni del KP, del BPP, del TSP e di $P \,||\, C_{\max}$;
- eseguire una ricerca locale con le regole first improvement e best improvement;
- valutare una mossa in modo incrementale, in $O(1)$ per la mossa 2-opt;
- uscire da un plateau con un obiettivo ausiliario, e gestire le soluzioni non ammissibili con una penalità;
- combinare più intorni (variable neighbourhood descent) e più ripartenze (multi-start).

3.1 Ricerca locale

Definizione 3.1: Ricerca locale

Dato un intorno N, una **ricerca locale** (indicata con LS) è un'euristica di miglioramento. Parte da una soluzione ammissibile e, a ogni iterazione, cerca nell'intorno della soluzione corrente una soluzione *migliore*. Se la trova, si sposta su di essa e ripete; se nessun vicino è migliore, la soluzione corrente è un ottimo locale rispetto a N e la ricerca si ferma.

- L'**intorno** $N(S)$ di una soluzione ammissibile S, la **mossa** da S a un vicino e l'**ottimo locale** rispetto a N sono definiti nel capitolo «Algoritmi euristici: definizioni».

Una ricerca locale migliora una soluzione ammissibile spostandosi da una soluzione a un vicino migliore, e si ferma in un ottimo locale rispetto a N.

3.2 I problemi, le ipotesi e le soluzioni di partenza

- Gli algoritmi sono illustrati sui quattro problemi del capitolo «Algoritmi euristici: definizioni»: il problema dello zaino 0/1 (KP), il problema del bin packing (BPP), il problema del commesso viaggiatore (TSP) e $P\,||\,C_{\max}$. Le loro definizioni e ipotesi sono nelle rispettive pagine; la tabella richiama la notazione.

problema	dati	soluzione	obiettivo
KP	p_j, w_j ($j \in \mathscr{I}$), capacità c	vettore binario x	max profitto
BPP	w_j ($j \in \mathscr{I}$), capacità c	vettore bin, schemi P_b	min numero di contenitori nb
TSP	d_{ij} ($i, j \in \mathscr{V}$)	giro σ	min lunghezza $L(\sigma)$
$P\,||\,C_{\max}$	t_j ($j \in \mathscr{J}$), m macchine	vettore $mach$, insiemi J_h	min makespan $C_{\max}$

- Per $P\,||\,C_{\max}$, $L_h = \sum_{j \in J_h} t_j$ è il **carico** della macchina h e $C_{\max} = \max_h L_h$. Per il BPP, $L_b = \sum_{j \in P_b} w_j$ è il **carico** del contenitore b. Una macchina (o un contenitore) il cui carico è uguale al valore obiettivo si dice **critica** (o critico).

- Le **istanze di riferimento** sono quelle del capitolo «Algoritmi euristici: definizioni»: il KP con $p = (10, 7, 6, 4)$, $w = (5, 4, 3, 3)$, $c = 9$; il BPP con $w = (4, 4, 5, 3, 2, 3)$, $c = 7$; il TSP con 5 città; $P\,||\,C_{\max}$ con $t = (5, 5, 4, 4, 3, 3, 3)$ e $m = 3$. I loro valori ottimi sono 17, 3, 18 e 9.

3.2.1 Soluzioni di partenza

- Una ricerca locale ha bisogno di una **soluzione di partenza** ammissibile. Usiamo le soluzioni calcolate nel capitolo «Euristiche costruttive», richiamate qui sotto.

problema	algoritmo	soluzione di partenza	valore	$z(I)$
KP	GR (greedy per efficienza)	$x = (1, 0, 1, 0)$, oggetti $\{1, 3\}$	16	17
KP	greedy per peso GR_w	$x = (0, 0, 1, 1)$, oggetti $\{3, 4\}$	10	17
BPP	NF	$\{1\}, \{2\}, \{3\}, \{4, 5\}, \{6\}$	5	3
BPP	FF	$\{1, 4\}, \{2, 5\}, \{3\}, \{6\}$	4	3
BPP	FFD	$\{3, 5\}, \{1, 4\}, \{2, 6\}$	3	3
TSP	NN dalla città 1	$\sigma = (1, 3, 2, 4, 5)$	25	18
$P\,||\,C_{\max}$	LPT	$J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$	11	9

- La soluzione di FFD è già ottima, quindi nessuna ricerca locale può migliorarla: per il BPP partiamo dalle soluzioni di NF e FF.

- Per studiare il comportamento di una ricerca locale su molte soluzioni di partenza usiamo anche **soluzioni di partenza casuali**. Sono generate con il generatore minimal standard di Park–Miller: dato un **seme** intero $s_0 \in \{1, \ldots, 2^{31} - 2\}$, il k-esimo numero pseudo-casuale è

$$s_k = 16807 \cdot s_{k-1} \bmod (2^{31} - 1), \qquad u_k = \frac{s_k}{2^{31} - 1} \in (0, 1),$$

e un intero pseudo-casuale in $\{0, 1, \ldots, q-1\}$ è $\lfloor q\, u_k \rfloor$.

- La sequenza è interamente determinata dal seme: lo stesso seme dà le stesse soluzioni di partenza, nelle note, negli esercizi e nel codice. Un **giro casuale** tiene la città 1 in prima posizione e rimescola le altre città (Fisher–Yates: per $k = n, n-1, \ldots, 3$, la città in posizione k si scambia con la città in una posizione casuale fra $2, \ldots, k$); un'**assegnazione casuale** dei lavori estrae la macchina dei lavori $1, 2, \ldots, n$ in quest'ordine; una **soluzione casuale del BPP** si calcola con FF su un ordine casuale degli oggetti (Fisher–Yates su tutte le posizioni: per $k = n, n-1, \ldots, 2$, l'oggetto in posizione k si scambia con l'oggetto in una posizione casuale fra $1, \ldots, k$).

3.3 Ricerca locale: lo schema

- Lo pseudocodice di LS è il seguente. La funzione *Improve* esplora l'intorno e restituisce un vicino migliorante di x, oppure *NULL* se non ne esiste nessuno.

Algoritmo

Algorithm: Ricerca locale

Input: una soluzione ammissibile x^0, un intorno N, una regola $\in \{first, best\}$
Output: un ottimo locale x rispetto a N

```
1 Function LocalSearch(x^0, N, rule)
2     x := x^0                                          // soluzione corrente
3     while true do
4         y := Improve(x, N, rule)                      // vicino migliorante, oppure NULL
5         if y = NULL then
6             return x                                  // nessun vicino migliorante: ottimo locale
7         end
8         x ← y                                         // spostarsi sul vicino migliorante
9     end
```

- Una ricerca locale è definita da quattro ingredienti:

  1. la **soluzione di partenza** x^0, calcolata da un'euristica costruttiva o a caso;
  2. l'**intorno** N, cioè la famiglia delle mosse, che dipende dalla rappresentazione delle soluzioni;
  3. il **criterio di accettazione**: qui un vicino è accettato solo se è strettamente migliore della soluzione corrente;

4. il **criterio di arresto**: qui la ricerca si ferma nel primo ottimo locale; in pratica si aggiunge spesso un limite di tempo.

- La funzione f che decide se un vicino è "migliore" è di solito l'obiettivo del problema, ma non è obbligatorio: la ricerca può essere guidata da una funzione diversa, purché la soluzione restituita sia ammissibile.

Proposizione 3.2

Per ogni soluzione di partenza, LS si ferma dopo un numero finito di mosse, in un ottimo locale rispetto a N. Se la funzione obiettivo assume valori interi e il problema è di minimizzazione, il numero di mosse è al più $f(x^0) - z(I)$; per un problema di massimizzazione, al più $z(I) - f(x^0)$.

Dimostrazione. Ogni mossa migliora strettamente l'obiettivo, quindi la sequenza delle soluzioni visitate non visita mai due volte la stessa soluzione. L'insieme delle soluzioni ammissibili è finito, quindi la sequenza è finita. Quando si ferma, *Improve* ha restituito *NULL*, cioè nessun vicino è migliore: l'ultima soluzione è un ottimo locale. Se f assume valori interi, ogni mossa migliora f di almeno 1, e f non può mai andare oltre $z(I)$: da qui i limiti sul numero di mosse. □

- Sull'istanza di riferimento del TSP, una ricerca locale che parte dal giro di NN (lunghezza 25) esegue al più $25 - 18 = 7$ mosse. Sulle istanze grandi questo limite è debole, e per alcuni intorni esistono istanze su cui il numero di mosse cresce esponenzialmente con n (per l'intorno 2-opt del TSP, si veda Englert, Röglin, Vöcking, *Algorithmica* 68, 2014). In pratica il numero di mosse è piccolo, e il tempo di esecuzione è dominato dall'esplorazione degli intorni.

- La ricerca locale **intensifica**: scende finché non raggiunge il fondo della valle in cui è partita. Quale valle, e quindi quale ottimo locale, dipende dalla soluzione di partenza, dall'intorno e dalla regola usata per scegliere la mossa.

3.4 Intorni per i quattro problemi

- Gli intorni sono definiti da piccole modifiche della rappresentazione delle soluzioni. All'intorno appartengono solo i vicini **ammissibili**: una modifica che produce un oggetto non ammissibile non è una mossa. Per ogni intorno fissiamo l'**ordine** in cui le mosse sono esaminate, che conta quando la ricerca accetta il primo vicino migliorante che trova.

3.4.1 KP: flip e swap

- L'intorno **flip**, definito nel capitolo «Algoritmi euristici: definizioni», cambia una componente di x, cioè aggiunge o toglie un oggetto; ha n elementi prima del test di ammissibilità. I flip sono esaminati per $j = 1, 2, \ldots, n$.

- L'intorno **swap** toglie un oggetto selezionato j e aggiunge un oggetto non selezionato k. Se S è l'insieme degli oggetti selezionati, ha $|S|\,(n - |S|) \leq n^2/4$ elementi; gli swap sono esaminati per $j \in S$ in ordine crescente e, per ogni j, per $k \notin S$ in ordine crescente.

- I due intorni sono complementari: un flip cambia il numero di oggetti selezionati, uno swap lo mantiene. L'intorno **flip+swap** è la loro unione, con i flip esaminati per primi.

3.4.2 BPP: move e swap

- L'intorno **move** sposta un oggetto j dal suo contenitore $a = bin_j$ a un altro contenitore non vuoto b con $L_b + w_j \leq c$. Ha al più $n\,(nb - 1)$ elementi; le mosse sono esaminate per $j = 1, 2, \ldots, n$ e, per ogni j, per i contenitori in ordine crescente di etichetta.

- Spostare un oggetto in un contenitore vuoto non è mai utile: non può diminuire il numero di contenitori. I contenitori mantengono le etichette della soluzione di partenza; un contenitore svuotato da una mossa semplicemente scompare.

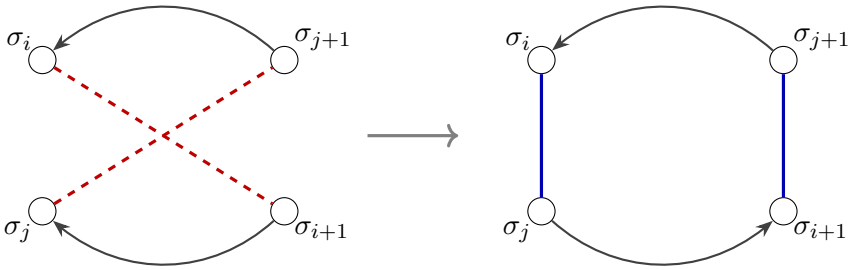
- L'intorno **swap** scambia due oggetti $j < k$ in contenitori diversi, purché entrambi i contenitori restino ammissibili: $L_{bin_j} - w_j + w_k \leq c$ e $L_{bin_k} - w_k + w_j \leq c$. Scambiare due oggetti di peso uguale non cambia nessun carico, e non si considera. Ha al più $\frac{n(n-1)}{2}$ elementi, esaminati per $j = 1, \ldots, n$ e $k = j + 1, \ldots, n$.

3.4.3 TSP: 2-opt e Or-opt

- Un giro è una permutazione $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ delle città, con $\sigma_{n+1} = \sigma_1$. Teniamo la città 1 in prima posizione, $\sigma_1 = 1$.

- La mossa **2-opt** con posizioni $i < j$ toglie i due lati $\{\sigma_i, \sigma_{i+1}\}$ e $\{\sigma_j, \sigma_{j+1}\}$, e richiude il giro nell'unico altro modo possibile: con i lati $\{\sigma_i, \sigma_j\}$ e $\{\sigma_{i+1}, \sigma_{j+1}\}$, percorrendo il cammino $\sigma_{i+1}, \ldots, \sigma_j$ nel verso opposto (Croes, *Operations Research* 6, 1958):

$$(\sigma_1, \ldots, \sigma_i, \underbrace{\sigma_{i+1}, \sigma_{i+2}, \ldots, \sigma_j}_{\text{cammino}}, \sigma_{j+1}, \ldots, \sigma_n)$$
$$\downarrow$$
$$(\sigma_1, \ldots, \sigma_i, \underbrace{\sigma_j, \ldots, \sigma_{i+2}, \sigma_{i+1}}_{\text{cammino invertito}}, \sigma_{j+1}, \ldots, \sigma_n).$$

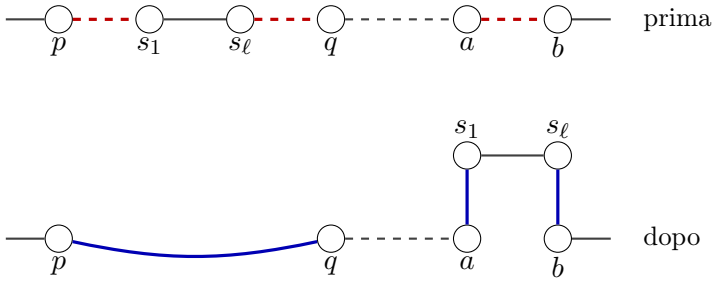


prima: i due lati tratteggiati sono tolti

dopo: due lati nuovi, il cammino in basso è invertito

- Le mosse sono esaminate per $i = 1, \ldots, n-2$ e $j = i+2, \ldots, n$, escluso $i = 1$, $j = n$ (i due lati avrebbero in comune la città σ_1). L'intorno 2-opt ha $\frac{n(n-3)}{2}$ elementi: 5 per l'istanza di riferimento.

- La mossa **Or-opt** (Or, tesi di dottorato, Northwestern University, 1976) prende un **segmento** di $\ell \in \{1, 2, 3\}$ città consecutive $\sigma_i, \ldots, \sigma_{i+\ell-1}$ ($2 \le i$, $i + \ell - 1 \le n$) e lo inserisce, con lo stesso orientamento, fra due città consecutive σ_k e σ_{k+1} esterne al segmento ($k \notin \{i-1, \ldots, i+\ell-1\}$). Siano $p = \sigma_{i-1}$ e $q = \sigma_{i+\ell}$ le città prima e dopo il segmento, $s_1 = \sigma_i$ e $s_\ell = \sigma_{i+\ell-1}$ la sua prima e la sua ultima città, $a = \sigma_k$ e $b = \sigma_{k+1}$:



- Le mosse sono esaminate per $\ell = 1, 2, 3$, poi per $i = 2, \ldots, n - \ell + 1$, poi per $k = 1, \ldots, n$. Per ogni ℓ ci sono $n - \ell$ segmenti e $n - \ell - 1$ punti di inserimento, quindi l'intorno ha

$\sum_{\ell=1}^{3}(n-\ell)(n-\ell-1)$ elementi, cioè $O(n^2)$: 170 per $n = 10$. Alcune di queste mosse danno lo stesso giro (spostare una sola città in avanti di una posizione equivale a spostare il suo successore indietro di una posizione).

- Una mossa Or-opt toglie *tre* lati, $\{p, s_1\}$, $\{s_\ell, q\}$ e $\{a, b\}$. L'intorno **3-opt** (Lin, *Bell System Technical Journal* 44, 1965) contiene tutti i modi di togliere tre lati e ricollegare i tre cammini: ha $O(n^3)$ elementi, e Or-opt ne è la parte che si può esplorare in tempo $O(n^2)$.

3.4.4 $P\,||\,C_{\max}$: move e swap

- L'intorno **move** sposta un lavoro j dalla sua macchina $a = mach_j$ a un'altra macchina $h \neq a$. Ogni mossa è ammissibile, e l'intorno ha $n\,(m-1)$ elementi, esaminati per $j = 1, \ldots, n$ e, per ogni j, per $h = 1, \ldots, m$.

- L'intorno **swap** scambia le macchine di due lavori $j < k$ su macchine diverse con $t_j \neq t_k$ (altrimenti nessun carico cambia). Ha al più $\frac{n(n-1)}{2}$ elementi, esaminati per $j = 1, \ldots, n$ e $k = j+1, \ldots, n$. Uno swap non cambia mai il numero di lavori su una macchina.

3.4.5 Riepilogo

problema	intorno	mossa	dimensione
KP	flip	aggiungere o togliere l'oggetto j	n
KP	swap	oggetto j fuori, oggetto k dentro	$\lvert S\rvert(n-\lvert S\rvert) \leq \frac{n^2}{4}$
BPP	move	oggetto j in un altro contenitore	$\leq n\,(nb-1)$
BPP	swap	scambiare gli oggetti j e k	$\leq \frac{n(n-1)}{2}$
TSP	2-opt	togliere 2 lati, invertire un cammino	$\frac{n(n-3)}{2}$
TSP	Or-opt	spostare un segmento di 1, 2 o 3 città	$\sum_{\ell=1}^{3}(n-\ell)(n-\ell-1)$
TSP	3-opt	togliere 3 lati, ricollegare	$O(n^3)$
$P\,\|\|\,C_{\max}$	move	lavoro j su un'altra macchina	$n\,(m-1)$
$P\,\|\|\,C_{\max}$	swap	scambiare i lavori j e k	$\leq \frac{n(n-1)}{2}$

- Le dimensioni sono quelle *prima* del test di ammissibilità (KP, BPP) e comprendono mosse che non cambiano l'obiettivo. La dimensione di un intorno è il numero di vicini da valutare a ogni iterazione, quindi determina il costo di un'iterazione.

3.5 First improvement e best improvement

- La funzione *Improve* di *LS* può fermarsi al *primo* vicino migliorante che trova, oppure esaminare tutto l'intorno e restituire il vicino *migliore*. Gli pseudocodici sono scritti per un problema di minimizzazione; per un problema di massimizzazione, "<" diventa ">".

3.5.1 First improvement

La regola *first improvement* esamina i vicini della soluzione corrente in un ordine fissato e restituisce il primo vicino migliore della soluzione corrente. Se nessun vicino è migliore, restituisce *NULL*.

Algoritmo

Algorithm: First improvement

Input: una soluzione ammissibile x, un intorno N, una funzione obiettivo f (da minimizzare)

Output: il primo vicino $y \in N(x)$ con $f(y) < f(x)$, oppure *NULL*

```
1 Function FirstImprovement(x, N, f)
2     for y ∈ N(x), nell'ordine dell'intorno do
3         if f(y) < f(x) then
4             return y                    // vicino migliorante: accettarlo subito
5         end
6     end
7     return NULL                         // x è un ottimo locale
```

3.5.2 Best improvement

La regola *best improvement* esamina tutti i vicini della soluzione corrente e restituisce il migliore, se è migliore della soluzione corrente; i pareggi si risolvono a favore del vicino esaminato per primo. Se nessun vicino è migliore, restituisce *NULL*.

Algoritmo

Algorithm: Best improvement

Input: una soluzione ammissibile x, un intorno N, una funzione obiettivo f (da minimizzare)

Output: un vicino migliore $y \in N(x)$ se $f(y) < f(x)$, altrimenti *NULL*

```
1  Function BestImprovement(x, N, f)
2      ȳ := NULL                    // miglior vicino migliorante trovato finora
3      f̄ := f(x)                                                  // il suo valore
4      for y ∈ N(x), nell'ordine dell'intorno do
5          if f(y) < f̄ then
6              ȳ ← y                // migliore di tutti i vicini visti finora
7              f̄ ← f(y)
8          end
9      end
10     return ȳ                                  // NULL se x è un ottimo locale
```

3.5.3 Confronto

- In un ottimo locale entrambe le regole esaminano tutto l'intorno: l'ultima iterazione costa lo stesso.

- Prima di raggiungerlo, **first improvement** esegue iterazioni **meno costose**, perché si ferma al primo vicino migliorante, ma ogni mossa di solito migliora meno, quindi tende a eseguire **più** mosse. **Best improvement** esegue la discesa più ripida: ogni iterazione costa un'esplorazione completa dell'intorno, e ogni mossa migliora il più possibile.

- Le due regole seguono percorsi diversi, quindi possono fermarsi in **ottimi locali diversi**. Nessuna delle due regole è migliore dell'altra su tutte le istanze: la mossa più ripida ora può portare in una valle peggiore dopo.

- Con first improvement il risultato dipende dall'ordine in cui sono esaminati i vicini; rendere casuale quell'ordine è un modo semplice per ottenere ottimi locali diversi dalla stessa soluzione di partenza.

Esempio 3.1: First e best improvement sul KP

Partiamo dalla soluzione del greedy per peso GR_w sull'istanza di riferimento del KP ($c = 9$), cioè $x = (0, 0, 1, 1)$, di peso 6 e profitto 10, con l'intorno flip+swap. I vicini sono esaminati nell'ordine della Sezione 3.4:

mossa	vicino	peso	profitto	
flip 1	$(1,0,1,1)$	11	—	non ammissibile
flip 2	$(0,1,1,1)$	10	—	non ammissibile
flip 3	$(0,0,0,1)$	3	4	
flip 4	$(0,0,1,0)$	3	6	
swap 3 fuori, 1 dentro	$(1,0,0,1)$	8	14	← first improvement
swap 3 fuori, 2 dentro	$(0,1,0,1)$	7	11	
swap 4 fuori, 1 dentro	$(1,0,1,0)$	8	16	← best improvement
swap 4 fuori, 2 dentro	$(0,1,1,0)$	7	13	

First improvement accetta $(1,0,0,1)$, di profitto 14, dopo aver esaminato 3 vicini ammissibili. Da $(1,0,0,1)$ (peso 8) i vicini sono esaminati di nuovo dall'inizio:

mossa	vicino	peso	profitto	
flip 1	$(0,0,0,1)$	3	4	
flip 2	$(1,1,0,1)$	12	—	non ammissibile
flip 3	$(1,0,1,1)$	11	—	non ammissibile
flip 4	$(1,0,0,0)$	5	10	
swap 1 fuori, 2 dentro	$(0,1,0,1)$	7	11	
swap 1 fuori, 3 dentro	$(0,0,1,1)$	6	10	
swap 4 fuori, 2 dentro	$(1,1,0,0)$	9	17	← first improvement

e $(1,1,0,0)$, di profitto 17, è accettata dopo 5 vicini ammissibili. È la soluzione ottima, quindi nessun vicino è migliore: i 6 vicini ammissibili sono esaminati e la ricerca si ferma. In totale: 2 mosse, $3+5+6=14$ vicini ammissibili esaminati.

Best improvement esamina tutti i 6 vicini ammissibili di $(0,0,1,1)$ e si sposta su $(1,0,1,0)$, di profitto 16. È la soluzione di GR, il cui intorno è nel capitolo «Algoritmi euristici: definizioni»: il vicino migliore è $(1,1,0,0)$, di profitto 17, raggiunto con lo swap "3 fuori, 2 dentro". La ricerca si ferma lì dopo aver esaminato i suoi 6 vicini ammissibili. In totale: 2 mosse, $6+6+6=18$ vicini ammissibili esaminati.

Le due regole raggiungono l'ottimo lungo percorsi diversi, $10 \to 14 \to 17$ e $10 \to 16 \to 17$, e first improvement esamina meno vicini. Il solo intorno flip non si sarebbe mosso affatto: da $(0,0,1,1)$ ogni flip ammissibile toglie un oggetto, quindi $(0,0,1,1)$, di profitto 10, è un ottimo locale per flip.

3.6 Valutare una mossa in modo incrementale

- Ogni iterazione di LS valuta molti vicini, e la maggior parte del tempo di esecuzione si spende lì. Calcolare $f(y)$ **da zero** costa $O(n)$ per ciascuno dei quattro problemi: il profitto di una soluzione del KP, i carichi dei contenitori o delle macchine, la lunghezza di un giro sono somme di n termini.

- Una mossa cambia solo poche componenti della soluzione. La **valutazione incrementale** calcola la **variazione** $\Delta = f(y) - f(x)$ dalle componenti che cambiano, conservando alcune informazioni sulla soluzione corrente (il suo profitto e il suo peso, i suoi carichi), senza ricalcolare $f(y)$.

Proposizione 3.3

Sia σ' il giro ottenuto da σ con la mossa 2-opt di posizioni $i < j$, e siano $u_1 = \sigma_i$, $v_1 = \sigma_{i+1}$, $u_2 = \sigma_j$, $v_2 = \sigma_{j+1}$. Allora $L(\sigma') = L(\sigma) + \Delta$, con

$$\Delta \;=\; d_{u_1u_2} + d_{v_1v_2} - d_{u_1v_1} - d_{u_2v_2}.$$

Dimostrazione. I giri σ e σ' hanno gli stessi lati, tranne che σ contiene $\{u_1, v_1\}$ e $\{u_2, v_2\}$ mentre σ' contiene $\{u_1, u_2\}$ e $\{v_1, v_2\}$. Il cammino invertito $v_1, \ldots, u_2$ usa gli stessi lati nel verso opposto, e la sua lunghezza non cambia perché le distanze sono simmetriche. □

La variazione di lunghezza di una mossa 2-opt si calcola in tempo $O(1)$:

$$\Delta \;=\; \underbrace{d_{u_1u_2} + d_{v_1v_2}}_{\text{lati aggiunti}} \;-\; \underbrace{d_{u_1v_1} + d_{u_2v_2}}_{\text{lati tolti}}$$

e la mossa è migliorante se e solo se $\Delta < 0$.

- Per esempio, sul giro $(1, 3, 2, 4, 5)$ dell'istanza di riferimento, di lunghezza 25, la mossa con posizioni $i = 2$, $j = 4$ toglie $\{3, 2\}$ e $\{4, 5\}$ e aggiunge $\{3, 4\}$ e $\{2, 5\}$: $\Delta = d_{34} + d_{25} - d_{32} - d_{45} = 4 + 4 - 4 - 7 = -3$. Il nuovo giro $(1, 3, 4, 2, 5)$ ha lunghezza $25 - 3 = 22$, come conferma il calcolo diretto: $2 + 4 + 3 + 4 + 9 = 22$.

- **Perché conta.** L'intorno 2-opt ha $\frac{n(n-3)}{2}$ mosse. Con la valutazione incrementale un'iterazione di best improvement costa $O(n^2)$; ricalcolare ogni giro da zero costerebbe $O(n^3)$. Per $n = 1000$ città, un'iterazione valuta 498 500 mosse: circa $5 \cdot 10^5$ operazioni invece di circa $5 \cdot 10^8$. L'inversione del cammino, che costa $O(n)$, si esegue una sola volta per iterazione, per la mossa accettata.

- La stessa idea si applica agli altri intorni:

mossa	variazione della soluzione	tempo
KP, flip di j	profitto $\pm p_j$, peso $\pm w_j$	$O(1)$
KP, swap j fuori, k dentro	profitto $+p_k - p_j$, peso $+w_k - w_j$	$O(1)$
BPP, move di j da a a b	$L_a - w_j$, $L_b + w_j$; il contenitore a si svuota sse $L_a = w_j$	$O(1)$
TSP, 2-opt	$d_{u_1u_2} + d_{v_1v_2} - d_{u_1v_1} - d_{u_2v_2}$	$O(1)$
TSP, Or-opt	$d_{pq} + d_{as_1} + d_{s_\ell b} - d_{ps_1} - d_{s_\ell q} - d_{ab}$	$O(1)$
$P \,\|\|\, C_{\max}$, move di j da a a h	$L_a - t_j$, $L_h + t_j$; $C_{\max} = \max\{L_a - t_j,\, L_h + t_j,\, \max_{q \neq a,h} L_q\}$	$O(m)$

- La formula di Or-opt si ottiene come per 2-opt: la mossa toglie i lati $\{p, s_1\}$, $\{s_\ell, q\}$, $\{a, b\}$ e aggiunge $\{p, q\}$, $\{a, s_1\}$, $\{s_\ell, b\}$, e il segmento mantiene il suo orientamento.

- Il costo di un'iterazione di *LS* è la dimensione dell'intorno per il costo della valutazione di una mossa (più il costo di applicare la mossa accettata, che si paga una volta per iterazione):

problema	intorno	valutazione di una mossa	costo di un'iterazione
KP	flip	$O(1)$	$O(n)$
KP	swap	$O(1)$	$O(n^2)$
BPP	move	$O(1)$	$O(n \cdot nb)$
BPP	swap	$O(1)$	$O(n^2)$
TSP	2-opt, Or-opt	$O(1)$	$O(n^2)$
TSP	3-opt	$O(1)$	$O(n^3)$
$P \,\|\|\, C_{\max}$	move	$O(m)$	$O(nm^2)$
$P \,\|\|\, C_{\max}$	swap	$O(m)$	$O(n^2m)$

- La valutazione incrementale non cambia gli ottimi locali, né il percorso seguito dalla ricerca: cambia solo il tempo necessario a seguirlo. Per questo il progetto di una ricerca locale parte sempre da due domande: *quali mosse?* e *quanto velocemente si può valutare una mossa?*

3.7 TSP: 2-opt e Or-opt

3.7.1 La ricerca locale 2-opt

La ricerca locale *2-opt* (indicata con $2OPT$) è una ricerca locale per il TSP con l'intorno 2-opt. A ogni iterazione valuta le mosse 2-opt del giro corrente con la formula incrementale, applica una mossa migliorante scelta con la regola first improvement o best improvement, e si ferma quando nessuna mossa accorcia il giro.

Algoritmo

Algorithm: Passo 2-opt

Input: n città, distanze $d[1..n][1..n]$, giro $\sigma[1..n]$ con $\sigma[1] = 1$, regola $\in \{\mathit{first}, \mathit{best}\}$
Output: *true* se a σ è stata applicata una mossa 2-opt migliorante, *false* altrimenti

```
1  Function TwoOptStep(n, d, σ, rule)
2      Δ̄ := 0                          // miglior variazione di lunghezza trovata finora
3      (ī, j̄) := (0, 0)                // miglior mossa trovata finora
4      for i := 1 to n − 2 do
5          for j := i + 2 to n do
6              if i = 1 and j = n then continue      // i due lati sono adiacenti
7              u1 := σ[i]; v1 := σ[i + 1]            // primo lato tolto
8              u2 := σ[j]; v2 := σ[j mod n + 1]      // secondo lato tolto
9              Δ := d[u1][u2] + d[v1][v2] − d[u1][v1] − d[u2][v2]
10             if Δ < Δ̄ then
11                 Δ̄ ← Δ; (ī, j̄) ← (i, j)           // mossa migliorante
12                 if rule = first then
13                     uscire da entrambi i cicli
14                 end
15             end
16         end
17     end
18     if ī = 0 then return false   // nessuna mossa migliorante: ottimo locale
19     invertire il cammino σ[ī + 1], . . . , σ[j̄]   // applicare la mossa
20     return true
```

- Nell'implementazione, *Improve* e lo spostamento sul vicino migliorante sono riuniti in un'unica funzione *Step*, che esplora l'intorno con la valutazione incrementale e, se trova una mossa migliorante, la applica alla soluzione corrente e restituisce *true*; altrimenti restituisce *false*. *LS* chiama *Step* finché non restituisce *false*.

- $2OPT$ è *LocalSearch* con *TwoOptStep*; non cambia mai $\sigma[1] = 1$, perché $i \geq 1$ e il cammino invertito parte dalla posizione $i + 1 \geq 2$.

- Un'iterazione costa tempo $O(n^2)$; l'inversione del cammino costa $O(n)$.

- Quando le distanze sono euclidee, un ottimo locale 2-opt non ha lati che si incrociano: se i lati $\{u_1, v_1\}$ e $\{u_2, v_2\}$ si incrociano in un punto X, la disuguaglianza triangolare dà

$d_{u_1u_2} \leq d_{u_1X} + d_{Xu_2}$ e $d_{v_1v_2} \leq d_{v_1X} + d_{Xv_2}$, quindi $\Delta \leq 0$, con $\Delta < 0$ a meno che le quattro città non siano allineate. La mossa 2-opt "scioglie" l'incrocio dei due lati.

Esempio 3.2: $2OPT$ sull'istanza di riferimento del TSP

Partiamo dal giro di NN dalla città 1, $\sigma = (1, 3, 2, 4, 5)$, di lunghezza 25. Le sue 5 mosse sono

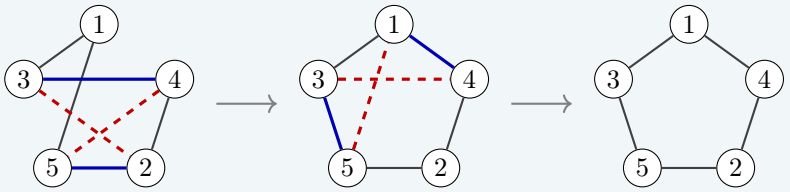
(i, j)	lati tolti	lati aggiunti	Δ	nuovo giro, lunghezza
$(1, 3)$	$\{1, 3\}, \{2, 4\}$	$\{1, 2\}, \{3, 4\}$	$5 + 4 - 2 - 3 = +4$	$(1, 2, 3, 4, 5)$, 29
$(1, 4)$	$\{1, 3\}, \{4, 5\}$	$\{1, 4\}, \{3, 5\}$	$2 + 7 - 2 - 7 = 0$	$(1, 4, 2, 3, 5)$, 25
$(2, 4)$	$\{3, 2\}, \{4, 5\}$	$\{3, 4\}, \{2, 5\}$	$4 + 4 - 4 - 7 = -3$	$(1, 3, 4, 2, 5)$, 22
$(2, 5)$	$\{3, 2\}, \{5, 1\}$	$\{3, 5\}, \{2, 1\}$	$7 + 5 - 4 - 9 = -1$	$(1, 3, 5, 4, 2)$, 24
$(3, 5)$	$\{2, 4\}, \{5, 1\}$	$\{2, 5\}, \{4, 1\}$	$4 + 2 - 3 - 9 = -6$	$(1, 3, 2, 5, 4)$, 19

First improvement accetta la mossa $(2, 4)$, la prima con $\Delta < 0$, dopo 3 valutazioni: $\sigma = (1, 3, 4, 2, 5)$, di lunghezza 22. Le sue mosse hanno $\Delta = +1, +6, +3, -4, 0$; la mossa $(2, 5)$, che toglie $\{3, 4\}$ e $\{5, 1\}$ e aggiunge $\{3, 5\}$ e $\{4, 1\}$, è accettata dopo 4 valutazioni: $\sigma = (1, 3, 5, 2, 4)$, di lunghezza 18. Tutte le sue 5 mosse hanno $\Delta > 0$ $(+7, +4, +1, +4, +6)$: la ricerca si ferma dopo $3 + 4 + 5 = 12$ valutazioni.

Best improvement accetta la mossa $(3, 5)$, con $\Delta = -6$: $\sigma = (1, 3, 2, 5, 4)$, di lunghezza 19. Le sue mosse hanno $\Delta = +6, +4, -1, +3, +6$, e la mossa $(2, 4)$, che toglie $\{3, 2\}$ e $\{5, 4\}$ e aggiunge $\{3, 5\}$ e $\{2, 4\}$, dà $\sigma = (1, 3, 5, 2, 4)$, di lunghezza 18. La ricerca si ferma dopo $5 + 5 + 5 = 15$ valutazioni.

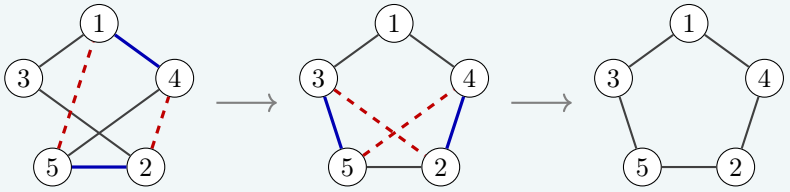
Entrambe le regole raggiungono il giro ottimo, con 2 mosse ciascuna. La figura mostra i due percorsi; le città sono disposte nell'ordine del giro ottimo, quindi il giro ottimo è il bordo del pentagono (l'istanza non è euclidea: il disegno è solo uno schema). I lati rossi tratteggiati sono tolti, quelli blu sono aggiunti.

first improvement



$(1, 3, 2, 4, 5)$, 25 $(1, 3, 4, 2, 5)$, 22 $(1, 3, 5, 2, 4)$, 18

best improvement



$(1, 3, 2, 4, 5)$, 25 $(1, 3, 2, 5, 4)$, 19 $(1, 3, 5, 2, 4)$, 18

3.7.2 La ricerca locale Or-opt

La ricerca locale *Or-opt* (indicata con *OROPT*) è una ricerca locale per il TSP con l'intorno Or-opt. A ogni iterazione valuta, con la formula incrementale, l'inserimento di ogni segmento di 1, 2 o 3 città consecutive fra altre due città consecutive, applica una mossa migliorante scelta con la regola first improvement o best improvement, e si ferma quando nessuna mossa accorcia il giro.

Algoritmo

Algorithm: Passo Or-opt

Input: n città, distanze $d[1..n][1..n]$, giro $\sigma[1..n]$ con $\sigma[1] = 1$, regola $\in \{\mathit{first}, \mathit{best}\}$
Output: *true* se a σ è stata applicata una mossa Or-opt migliorante, *false* altrimenti

Function *OrOptStep*$(n, d, \sigma, \mathit{rule})$
  $\bar{\Delta} := 0$; $(\bar{\ell}, \bar{\imath}, \bar{k}) := (0, 0, 0)$ `// miglior mossa trovata finora`
  for $\ell := 1$ **to** 3 **do**
    for $i := 2$ **to** $n - \ell + 1$ **do**
      $p := \sigma[i-1]$; $s_1 := \sigma[i]$ `// predecessore, prima città del segmento`
      $s_\ell := \sigma[i + \ell - 1]$; $q := \sigma[(i + \ell - 1) \bmod n + 1]$ `// ultima città, successore`
      for $k := 1$ **to** n **do**
        if $i - 1 \le k \le i + \ell - 1$ **then continue** `// stesso posto, o dentro il segmento`
        $a := \sigma[k]$; $b := \sigma[k \bmod n + 1]$ `// lato in cui va il segmento`
        $\Delta := d[p][q] + d[a][s_1] + d[s_\ell][b] - d[p][s_1] - d[s_\ell][q] - d[a][b]$
        if $\Delta < \bar{\Delta}$ **then**
          $\bar{\Delta} \leftarrow \Delta$; $(\bar{\ell}, \bar{\imath}, \bar{k}) \leftarrow (\ell, i, k)$ `// mossa migliorante`
          if *rule* = *first* **then**
            uscire dai tre cicli
          end
        end
      end
    end
  end
  if $\bar{\ell} = 0$ **then return** *false* `// nessuna mossa migliorante: ottimo locale`
  spostare $\sigma[\bar{\imath}], \ldots, \sigma[\bar{\imath} + \bar{\ell} - 1]$ fra $\sigma[\bar{k}]$ e $\sigma[\bar{k} \bmod n + 1]$ `// applicare`
  return *true*

- Or-opt sposta un breve tratto del giro in un posto migliore, senza invertire nulla: è l'intorno naturale quando alcune città sono state "lasciate indietro" dalla costruzione. Un'iterazione costa tempo $O(n^2)$.

- Un ottimo locale 2-opt non è necessariamente un ottimo locale Or-opt, e viceversa: i due intorni contengono mosse diverse.

3.7.3 First e best improvement su un'istanza media

- Con 5 città l'intorno 2-opt è piccolo ed entrambe le regole raggiungono l'ottimo. Per vedere la differenza fra le due regole, e il ruolo di Or-opt, usiamo un'istanza media con 10 città. Questa istanza non è metrica: $d_{68} = 8 > d_{67}+d_{78} = 4+3$ e $d_{6,10} = 11 > d_{67}+d_{7,10} = 4+6$.

Esempio 3.3: Un'istanza del TSP con 10 città

Le città hanno coordinate intere, e d_{ij} è la distanza euclidea arrotondata all'intero più vicino:

città	1	2	3	4	5	6	7	8	9	10
coordinate	$(1,1)$	$(3,8)$	$(6,10)$	$(10,9)$	$(13,6)$	$(12,1)$	$(8,3)$	$(5,4)$	$(8,7)$	$(2,5)$

d_{ij}	1	2	3	4	5	6	7	8	9	10
1	—	7	10	12	13	11	7	5	9	4
2	7	—	4	7	10	11	7	4	5	3
3	10	4	—	4	8	11	7	6	4	6
4	12	7	4	—	4	8	6	7	3	9
5	13	10	8	4	—	5	6	8	5	11
6	11	11	11	8	5	—	4	8	7	11
7	7	7	7	6	6	4	—	3	4	6
8	5	4	6	7	8	8	3	—	4	3
9	9	5	4	3	5	7	4	4	—	6
10	4	3	6	9	11	11	6	3	6	—

Fra i suoi 181 440 giri distinti, l'unico ottimo è $(1,10,2,3,9,4,5,6,7,8)$, di lunghezza $z(I) = 39$. Il giro di NN dalla città 1 è $(1,10,2,3,4,9,7,8,5,6)$, di lunghezza 49, e l'intorno 2-opt ha $\frac{10\cdot 7}{2} = 35$ mosse.

First improvement esegue 4 mosse:

lati tolti	nuovo giro	Δ	lunghezza	valutazioni
$\{3,4\},\{8,5\}$	$(1,10,2,3,8,7,9,4,5,6)$	-2	47	23
$\{3,8\},\{6,1\}$	$(1,10,2,3,6,5,4,9,7,8)$	-1	46	25
$\{3,6\},\{4,9\}$	$(1,10,2,3,4,5,6,9,7,8)$	-3	43	22
$\{6,9\},\{7,8\}$	$(1,10,2,3,4,5,6,7,9,8)$	-2	41	33

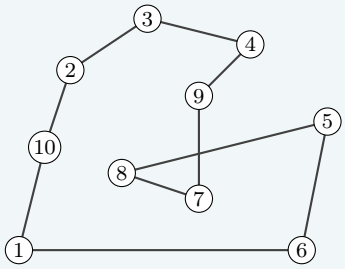
e si ferma nell'ottimo locale $(1,10,2,3,4,5,6,7,9,8)$, di lunghezza 41, dopo 138 valutazioni in totale (le ultime 35 per certificare l'ottimo locale).

Best improvement esegue 3 mosse:

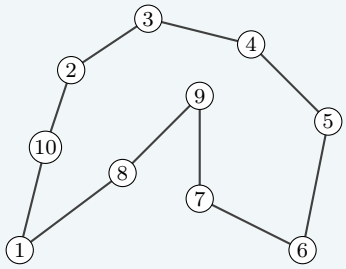
lati tolti	nuovo giro	Δ	lunghezza
$\{7,8\},\{6,1\}$	$(1,10,2,3,4,9,7,6,5,8)$	-5	44
$\{9,7\},\{5,8\}$	$(1,10,2,3,4,9,5,6,7,8)$	-4	40
$\{3,4\},\{9,5\}$	$(1,10,2,3,9,4,5,6,7,8)$	-1	39

e si ferma nel giro ottimo dopo $4 \cdot 35 = 140$ valutazioni.

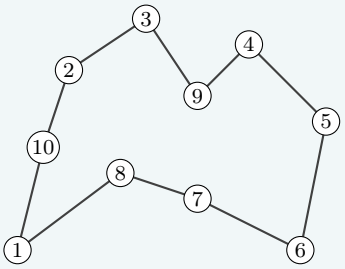
Dallo stesso giro di partenza, le due regole si fermano in **ottimi locali diversi**, dopo quasi lo stesso numero di valutazioni. Fra tutti i 181 440 giri, solo 3 sono ottimi locali 2-opt, di lunghezza 39, 41 e 43.



NN: 49



first improvement: 41



best improvement: 39

Esempio 3.4: *OROPT* da un ottimo locale 2-opt

Sull'istanza dell'Esempio 3.3, first improvement si è fermato nell'ottimo locale 2-opt $\sigma = (1, 10, 2, 3, 4, 5, 6, 7, 9, 8)$, di lunghezza 41. La mossa Or-opt che prende il segmento (9), con $\ell = 1$, fra $p = 7$ e $q = 8$, e lo inserisce fra $a = 3$ e $b = 4$ ha

$$\Delta = d_{78} + d_{39} + d_{94} - d_{79} - d_{98} - d_{34} = 3 + 4 + 3 - 4 - 4 - 4 = -2.$$

È la prima mossa migliorante trovata, dopo 60 valutazioni, e anche la migliore delle 170 mosse. Il nuovo giro $(1, 10, 2, 3, 9, 4, 5, 6, 7, 8)$, di lunghezza 39, è ottimo. Fra tutti i 181 440 giri, solo 3 sono ottimi locali Or-opt, di lunghezza 39, 43 e 44, e solo il giro ottimo è un ottimo locale sia per 2-opt sia per Or-opt.

3.7.4 3-opt e Lin–Kernighan

- L'intorno **3-opt** toglie tre lati e ricollega i tre cammini in tutti i modi possibili (Lin, 1965). I suoi ottimi locali sono migliori di quelli di 2-opt, ma un'iterazione costa tempo $O(n^3)$, e in pratica l'intorno si esplora solo in parte.

- L'euristica di **Lin–Kernighan** (Lin, Kernighan, *Operations Research* 21, 1973) costruisce un numero *variabile* k di scambi di lati, uno alla volta, scegliendo ogni scambio in modo greedy e fermandosi quando il guadagno totale non può più diventare positivo. Esplora parte dell'intorno k-opt per ogni k, in un tempo vicino a quello di 2-opt, e le sue implementazioni sono fra le migliori euristiche per il TSP.

- La sequenza 2-opt, 3-opt, k-opt mostra il compromesso del capitolo «Algoritmi euristici: definizioni»: un intorno più grande ha ottimi locali migliori, e costa di più da esplorare.

3.8 $P\,||\,C_{\max}$: move e swap

- Le mosse di $P\,||\,C_{\max}$ 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 $L_a - \delta$ e $L_b + \delta$, e $F(L, a, b, \delta)$ indica la variazione risultante dell'obiettivo.

3.8.1 La ricerca locale move

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

Algoritmo

Algorithm: Passo move per $P\,||\,C_{\max}$

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      Δ̄ := 0; (j̄, h̄) := (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
9              Δ := F(L, a, h, t[j])                  // variazione dell'obiettivo
10             if Δ < Δ̄ then
11                 Δ̄ ← Δ; (j̄, h̄) ← (j, h)                  // mossa migliorante
12                 if rule = first then
13                     uscire da entrambi i cicli
14                 end
15             end
16         end
17     end
18     if j̄ = 0 then return false  // nessuna mossa migliorante: ottimo locale
19     L[mach[j̄]] ← L[mach[j̄]] − t[j̄]; L[h̄] ← L[h̄] + t[j̄]   // aggiornare i carichi
20     mach[j̄] ← h̄                                    // applicare la mossa
21     return true
```

3.8.2 La ricerca locale swap

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

Algoritmo

Algorithm: Passo swap per $P\,||\,C_{\max}$

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      Δ̄ := 0; (j̄, k̄) := (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
7              Δ := F(L, a, b, t[j] − t[k])          // variazione dell'obiettivo
8              if Δ < Δ̄ then
9                  Δ̄ ← Δ; (j̄, k̄) ← (j, k)                  // swap migliorante
10                 if rule = first then
11                     uscire da entrambi i cicli
12                 end
13             end
14         end
15     end
16     if j̄ = 0 then return false   // nessuno swap migliorante:  ottimo locale
17     a := mach[j̄]; b := mach[k̄]; δ := t[j̄] − t[k̄]
18     L[a] ← L[a] − δ; L[b] ← L[b] + δ          // aggiornare i carichi
19     mach[j̄] ← b; mach[k̄] ← a                    // applicare lo swap
20     return true
```

- Per il makespan, la variazione dell'obiettivo è

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

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

- 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.

- 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, e i due intorni si completano a vicenda.

Esempio 3.5: La soluzione di *LPT* è un ottimo locale per move, ma non per swap

La soluzione di *LPT* sull'istanza di riferimento di $P\,||\,C_{\max}$ ha $J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$, carichi $(11, 8, 8)$ e $C_{\max} = 11$. La macchina 1 è l'unica macchina critica.

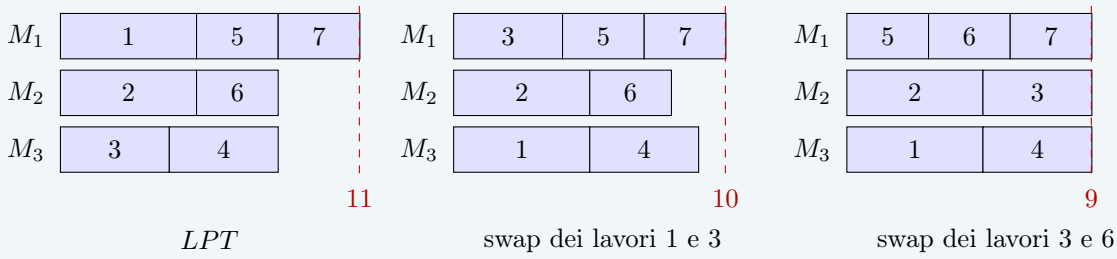
Move. Le 14 mosse danno i carichi

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

Nessuna mossa diminuisce il makespan: spostare il lavoro 1 via dalla macchina 1 porta la macchina ricevente a $8 + 5 = 13$, e 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 $C_{\max} = 11$.

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

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 $J_1 = \{5, 6, 7\}$, $J_2 = \{2, 3\}$, $J_3 = \{1, 4\}$ ha $C_{\max} = 9 = \lceil 27/3 \rceil$: è **ottima**.



- 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**.

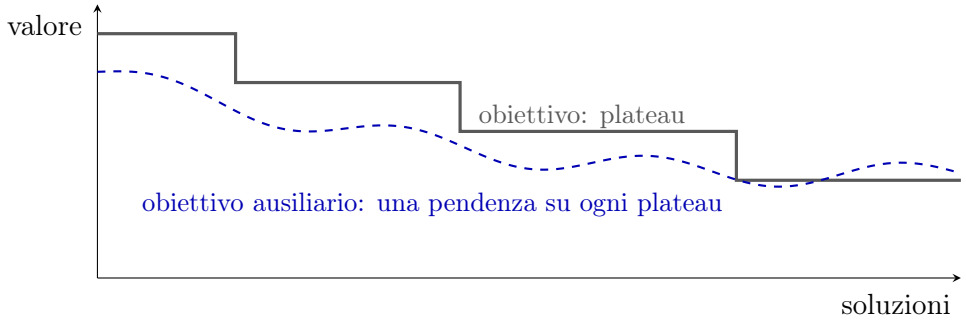
- Un intorno più grande non sempre aiuta. Sull'istanza con $m = 4$ macchine e 12 lavori con tempi $t = (11, 10, 9, 8, 7, 7, 6, 5, 4, 4, 3, 2)$, il cui makespan ottimo è $z(I) = 19 = \lceil 76/4 \rceil$, LPT dà $J_1 = \{1, 8, 10\}$, $J_2 = \{2, 7, 11\}$, $J_3 = \{3, 6, 12\}$, $J_4 = \{4, 5, 9\}$, con carichi $(20, 19, 18, 19)$: questa soluzione è un ottimo locale sia per move sia per swap.

3.9 Paesaggi piatti e obiettivi ausiliari

3.9.1 Plateau

- Gli obiettivi del BPP e di $P\,||\,C_{\max}$ 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.



- Quanto piatto? Sull'istanza di $P\,||\,C_{\max}$ 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.

3.9.2 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:

$$\text{BPP:}\quad \max \sum_b L_b^2 = \sum_b \Big(\sum_{j \in P_b} w_j \Big)^2, \qquad P \,||\, C_{\max}:\quad \min \sum_{h=1}^{m} L_h^2.$$

- 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 3.4

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

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

Dimostrazione. Sviluppando i quadrati, $(L_a-\delta)^2 - L_a^2 = -2\delta L_a + \delta^2$ e $(L_b+\delta)^2 - L_b^2 = 2\delta L_b + \delta^2$; la somma è $2\delta(L_b - L_a + \delta)$. Gli altri carichi non cambiano. □

- 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.

3.9.3 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.

Algoritmo

Algorithm: Passo move per il BPP

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      Ḡ := 0; (j̄, b̄) := (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
7              if f = SQ then
8                  G := 2 w[j] (L[b] − L[a] + w[j])                    // Proposizione 3.4
9              else
10                 G := 1 if L[a] = w[j] else 0          // il contenitore a si svuota
11             end
12             if G > Ḡ then
13                 Ḡ ← G; (j̄, b̄) ← (j, b)                          // mossa migliorante
14                 if rule = first then
15                     uscire da entrambi i cicli
16                 end
17             end
18         end
19     end
20     if j̄ = 0 then return false  // nessuna mossa migliorante:  ottimo locale
21     L[bin[j̄]] ← L[bin[j̄]] − w[j̄]; L[b̄] ← L[b̄] + w[j̄]        // aggiornare i carichi
22     bin[j̄] ← b̄                                          // applicare la mossa
23     return true
```

3.9.4 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.

Algoritmo

Algorithm: Passo swap per il BPP

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      Ḡ := 0; (j̄, k̄) := (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              δ := w[j] − w[k]                          // peso trasferito da a a b
8              if L[b] + δ > c or L[a] − δ > c then continue   // un contenitore
                sarebbe troppo pieno
9              G := 2 δ (L[b] − L[a] + δ)                    // Proposizione 3.4
10             if G > Ḡ then
11                 Ḡ ← G; (j̄, k̄) ← (j, k)                   // swap migliorante
12                 if rule = first then
13                     uscire da entrambi i cicli
14                 end
15             end
16         end
17     end
18     if j̄ = 0 then return false   // nessuno swap migliorante:  ottimo locale
19     a := bin[j̄]; b := bin[k̄]; δ := w[j̄] − w[k̄]
20     L[a] ← L[a] − δ; L[b] ← L[b] + δ                  // aggiornare i carichi
21     bin[j̄] ← b; bin[k̄] ← a                              // applicare lo swap
22     return true
```

3.9.5 Proprietà dell'obiettivo ausiliario

Proposizione 3.5

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 3.4 con $\delta = w_j$ la variazione è $2w_j(L_b - w_j + w_j) = 2w_j L_b > 0$. 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 3.6

Per $P \,||\, C_{\max}$, 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. 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 3.4, $\sum_h L_h^2$ 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. (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. □

- Le due proposizioni dicono che, 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 $P \,||\, C_{\max}$, 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.

- 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 $25 + 25 = 50$, mentre 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.

3.9.6 Esempi

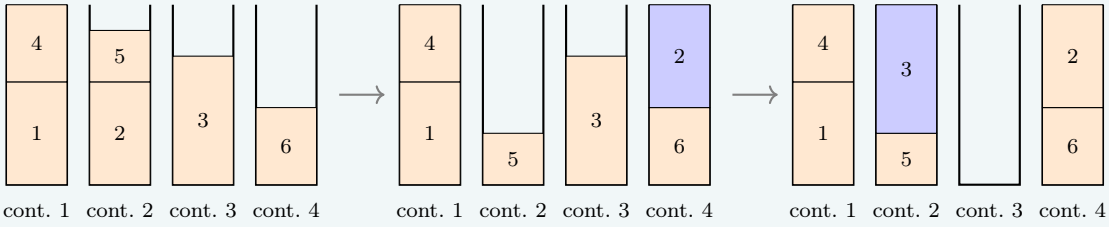
Esempio 3.6: Il numero di contenitori è cieco, l'obiettivo ausiliario no

La soluzione di FF sull'istanza di riferimento del BPP ($c = 7$) ha gli schemi di carico $P_1 = \{1, 4\}$, $P_2 = \{2, 5\}$, $P_3 = \{3\}$, $P_4 = \{6\}$, con carichi $(7, 6, 5, 3)$, $nb = 4$ e $SQ = 49 + 36 + 25 + 9 = 119$. Le sue mosse ammissibili sono

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

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 3.5.

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.



FF: $nb = 4$, $SQ = 119$ | oggetto 2 nel contenitore 4: $SQ = 127$ | oggetto 3 nel contenitore 2: $SQ = 147$

Esempio 3.7: Partendo da NF

La soluzione di NF sull'istanza di riferimento del BPP ha gli schemi di carico $\{1\}$, $\{2\}$, $\{3\}$, $\{4, 5\}$, $\{6\}$ (contenitori $1, \ldots, 5$), 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.

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.

3.9.7 Esperimenti su due istanze medie

- 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.

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

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

- 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.

- 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 FF)	14	14
move	14	76
swap	—	14

- 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.

3.10 KP: ammissibilità e penalità

- 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.

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

$$g_\alpha(x) \;=\; \sum_{j \in \mathscr{I}} p_j x_j \;-\; \alpha \cdot \max\Big\{0,\; \sum_{j \in \mathscr{I}} w_j x_j - c\Big\},$$

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.

3.10.1 Restare ammissibili

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

Algoritmo

Algorithm: Passo flip+swap per il KP

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 := Σ_{j=1}^{n} w[j] x[j]                          // peso della soluzione corrente
3      Δ̄ := 0; (j̄, k̄) := (0, 0)      // miglior mossa trovata finora (k̄ = 0: flip)
4      for j := 1 to n do                                   // flip dell'oggetto j
5          Δp := (1 − 2x[j]) p[j]; Δw := (1 − 2x[j]) w[j]  // togliere, o aggiungere,
            l'oggetto j
6          if W + Δw ≤ c and Δp > Δ̄ then
7              Δ̄ ← Δp; (j̄, k̄) ← (j, 0)
8              if rule = first then
9                  uscire da tutti i cicli
10             end
11         end
12     end
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             Δp := p[k] − p[j]; Δw := w[k] − w[j]
16             if W + Δw ≤ c and Δp > Δ̄ then
17                 Δ̄ ← Δp; (j̄, k̄) ← (j, k)
18                 if rule = first then
19                     uscire da tutti i cicli
20                 end
21             end
22         end
23     end
24     if j̄ = 0 then return false  // nessuna mossa migliorante: ottimo locale
25     x[j̄] ← 1 − x[j̄]                 // flip, oppure togliere l'oggetto che esce
26     if k̄ ≠ 0 then
27         x[k̄] ← 1                                  // aggiungere l'oggetto che entra
28     end
29     return true
```

- 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.

3.10.2 Penalizzare il peso in eccesso

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

Algoritmo

Algorithm: Passo penalty flip per il KP

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, α, x, rule)
2      P := Σ_{j=1}^{n} p[j] x[j]; W := Σ_{j=1}^{n} w[j] x[j]   // profitto e peso di x
3      g := P − α max{0, W − c}                                  // valore di x
4      Δ̄ := 0; j̄ := 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          Δ := P' − α max{0, W' − c} − g                        // variazione di g_α
8          if Δ > Δ̄ then
9              Δ̄ ← Δ; j̄ ← j                                     // flip migliorante
10             if rule = first then
11                 uscire dal ciclo
12             end
13         end
14     end
15     if j̄ = 0 then return false    // nessun flip migliorante: ottimo locale
16     x[j̄] ← 1 − x[j̄]                                          // applicare il flip
17     return true
```

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

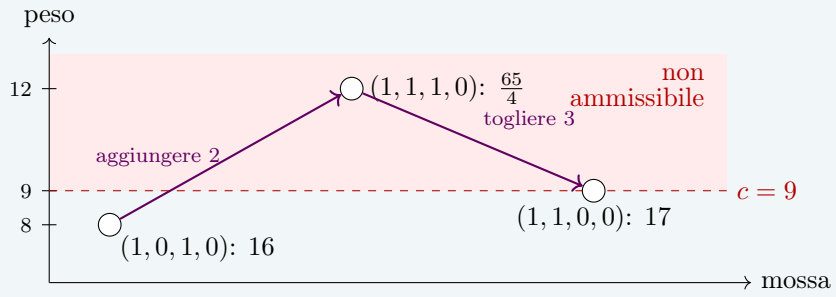
Esempio 3.8: Penalità sull'istanza di riferimento del KP

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

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.
- $\alpha = \frac{9}{4}$: 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).
- $\alpha = 3$: nessun flip migliora, e la ricerca si ferma subito in $(1, 0, 1, 0)$, come la ricerca ammissibile.



Proposizione 3.7

Siano i dati interi e sia $\alpha > \max_{j \in \mathscr{J}} p_j$. 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 $-p_j + \alpha \min\{w_j, e\} \geq -p_j + \alpha > 0$. □

- 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), quindi 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.

3.11 Variable neighbourhood descent

- 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.

La *variable neighbourhood descent* (indicata con VND) esplora gli intorni $N_1, \dots, N_K$ 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 intorni.

Algoritmo

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, ..., Step_K, rule)
2      k := 1                                              // intorno corrente
3      while k ≤ K do
4          if Step_k(x, rule) then
5              k ← 1          // mossa migliorante applicata: tornare al primo
                intorno
6          else
7              k ← k + 1          // ottimo locale per N_k: intorno successivo
8          end
9      end
10     return x
```

- VND si ferma dopo un numero finito di mosse, per l'argomento della Proposizione 3.2. 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 $N_1 \cup \dots \cup N_K$.

- 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 3.9: VND su $P\,||\,C_{\max}$ con move e swap

Sull'istanza di riferimento, dalla soluzione di LPT, con N_1 = move e N_2 = 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 3.10: VND sul TSP con 2-opt e Or-opt

Sull'istanza con 10 città dell'Esempio 3.3, dal giro di NN (lunghezza 49), con N_1 = 2-opt, N_2 = Or-opt e la regola first improvement, VND esegue le quattro mosse 2-opt della ricerca locale 2-opt ($49 \to 47 \to 46 \to 43 \to 41$), poi la mossa Or-opt dell'Esempio 3.4, che inserisce la città 9 fra le città 3 e 4 ($41 \to 39$), e si ferma: il giro ottimo è un ottimo locale per entrambi gli intorni. Con la regola best improvement, il solo 2-opt raggiunge già l'ottimo.

- Sulle due istanze medie della Sezione 3.9, con 100 soluzioni di partenza casuali (seme 1) e best improvement, VND con N_1 = move e N_2 = swap dà:

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

- Rispetto ai singoli intorni della Sezione 3.9, VND con l'obiettivo ausiliario raggiunge l'ottimo in 68 esecuzioni su 100 per $P \,||\, C_{\max}$ (solo move: 4) e in 89 per il BPP (solo move: 76). Combinare gli intorni e usare l'obiettivo ausiliario sono miglioramenti indipendenti, e si sommano.

3.12 Ricerca locale multi-start

- 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, e restituisce il miglior ottimo locale trovato.

Algoritmo

Algorithm: 2-opt multi-start

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      L̄ := +∞                          // lunghezza del miglior giro trovato finora
3      for r := 1 to M do
4          σ := RandomTour(n, s) // giro di partenza casuale; aggiorna il seme
5          while TwoOptStep(n, d, σ, rule) do
               // ripetere finché σ non è un ottimo locale 2-opt
6          end
7          if L(σ) < L̄ then
8              L̄ ← L(σ); σ̄ ← σ                    // nuovo miglior ottimo locale
9          end
10     end
11     return (σ̄, L̄)
```

- 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 3.11: $MSLS$ sul TSP con 10 città

Sull'istanza dell'Esempio 3.3, 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	—

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, e VND trova sempre l'ottimo, l'unico giro che è un ottimo locale per entrambi gli intorni.

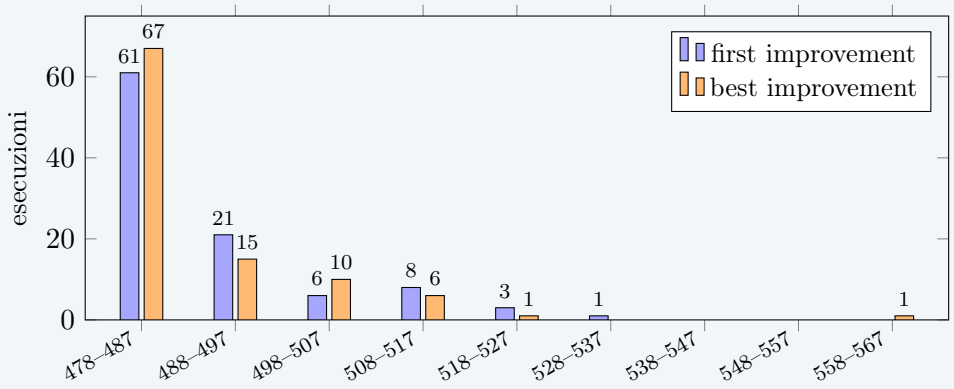
Esempio 3.12: $MSLS$ sul TSP con 30 città

Le 30 città hanno coordinate $(\lfloor 101u_{2k-1} \rfloor, \lfloor 101u_{2k} \rfloor)$, $k = 1, \ldots, 30$, dove $u_1, u_2, \ldots$ 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). 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

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



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.

3.12.1 I limiti del multi-start

- **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.

- **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.

3.12.2 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: 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.

3.13 Errori tipici

Attenzione

- **Dimenticare che un ottimo locale dipende dall'intorno.** La soluzione di LPT è un ottimo locale per move, ma non per swap.
- **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.
- **Ignorare l'ordine di esame.** Le mosse sono esaminate in un ordine fissato e i pareggi vanno alla prima esaminata (nell'Esempio 3.6 l'oggetto 3 viene prima dell'oggetto 5).
- **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.
- **Concludere che un ottimo locale è ottimo.** Solo un bound duale può dirlo.

3.14 In sintesi

In sintesi

- 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.

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

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

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

3.15 Implementazione in C++

- 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 di queste note e tutte le istanze degli esercizi, e confronta i risultati (i valori lungo ogni traiettoria, e le statistiche degli esperimenti) con quelli calcolati da un motore indipendente.

- In questa sezione 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 e la **variable neighbourhood descent** ricevono i passi come funzioni senza argomenti, che agiscono sulla soluzione corrente:

```
// generic local search: Step looks for an improving neighbour of the current solution
// and, if it finds one, moves to it and returns true
// the function returns the number of moves performed

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

```
// variable neighbourhood descent: Steps[0], ..., Steps[K-1] explore the neighbourhoods N_1, ..., N_K
// after every improving move the search restarts from the first neighbourhood
// the function returns the number of moves performed

int VND(const std::vector<std::function<bool()>>& Steps)
{
    int K = (int)Steps.size();              // number of neighbourhoods
    int k = 0;                              // current neighbourhood
    int moves = 0;                          // number of moves performed
    while (k < K)
    {
        if (Steps[k]())                     // improving neighbour found in N_k, and accepted
        {
            moves = moves + 1;
            k = 0;                          // back to the first neighbourhood
```

```
        }
        else
        {
            k = k + 1;                    // local optimum for N_k: try the next neighbourhood
        }
    }
    return moves;                         // local optimum for all the K neighbourhoods
}
```

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

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

- Il **passo 2-opt**:

```
// one step of the 2-opt local search on the tour pi[0..n-1] (pi[0] never moves)
// the move (i, j) removes the edges {pi[i], pi[i+1]} and {pi[j], pi[j+1]},
// reverses the path pi[i+1..j] and adds the edges {pi[i], pi[j]} and {pi[i+1], pi[j+1]}

bool TwoOptStep(int n, const Mat& d, Vec& pi, int rule)
{
    int bestDelta = 0;                    // best length change found so far
    int bi = -1, bj = -1;                 // best move found so far
    bool stop = false;
    for (int i = 0; i <= n - 3 && !stop; ++i)
    {
        for (int j = i + 2; j <= n - 1; ++j)
        {
            if (i == 0 && j == n - 1) continue;          // the two edges are adjacent
            int u1 = pi[i], v1 = pi[i + 1];               // first removed edge
            int u2 = pi[j], v2 = pi[(j + 1) % n];         // second removed edge
            int delta = d[u1][u2] + d[v1][v2] - d[u1][v1] - d[u2][v2];
            if (delta < bestDelta)
            {
                bestDelta = delta; bi = i; bj = j;        // improving move
                if (rule == FIRST) { stop = true; break; }
            }
        }
    }
    if (bi == -1) return false;           // no improving move: local optimum

    for (int a = bi + 1, b = bj; a < b; ++a, --b)        // reverse pi[bi+1..bj]
    {
        int tmp = pi[a]; pi[a] = pi[b]; pi[b] = tmp;
```

```
    }
    return true;
}
```

- Il **passo Or-opt**:

```
// one step of the Or-opt local search on the tour pi[0..n-1] (pi[0] never moves)
// the move (l, i, k) takes the segment pi[i..i+l-1] of l = 1, 2, 3 consecutive cities
// and inserts it, with the same orientation, between pi[k] and pi[k+1]

bool OrOptStep(int n, const Mat& d, Vec& pi, int rule)
{
    int bestDelta = 0;                  // best length change found so far
    int bl = -1, bi = -1, bk = -1;      // best move found so far
    bool stop = false;
    for (int l = 1; l <= 3 && !stop; ++l)
    {
        for (int i = 1; i + l - 1 <= n - 1 && !stop; ++i)
        {
            int p = pi[i - 1], s1 = pi[i];                // predecessor and first city of the segment
            int sl = pi[i + l - 1], q = pi[(i + l) % n];  // last city of the segment and successor
            for (int k = 0; k <= n - 1; ++k)
            {
                if (k >= i - 1 && k <= i + l - 1) continue;   // same position, or inside the segment
                int a = pi[k], b = pi[(k + 1) % n];           // edge where the segment is inserted
                int delta = d[p][q] + d[a][s1] + d[sl][b] - d[p][s1] - d[sl][q] - d[a][b];
                if (delta < bestDelta)
                {
                    bestDelta = delta; bl = l; bi = i; bk = k;   // improving move
                    if (rule == FIRST) { stop = true; break; }
                }
            }
        }
    }
    if (bl == -1) return false;         // no improving move: local optimum

    Vec newpi;                          // the new tour
    for (int k = 0; k <= n - 1; ++k)
    {
        if (k >= bi && k <= bi + bl - 1) continue;        // the segment is copied after pi[bk]
        newpi.push_back(pi[k]);
        if (k == bk)
            for (int r = bi; r <= bi + bl - 1; ++r) newpi.push_back(pi[r]);
    }
    pi = newpi;
    return true;
}
```

- La variazione F dell'obiettivo di $P\,||\,C_{\max}$, per il makespan e per la somma dei carichi al quadrato, e i **passi move** e **swap** per $P\,||\,C_{\max}$:

```
// change of the objective of P||Cmax when the time delta is transferred from machine a to machine b
// (a move transfers delta = t_j, a swap transfers delta = t_j - t_k)
// obj = CMAX: change of the makespan; obj = SQ: change of the sum of the squared loads

int ObjectiveChange(int m, const Vec& L, int a, int b, int delta, int obj)
{
    if (obj == SQ)
        return 2 * delta * (L[b] - L[a] + delta);

    int Cmax = 0, newC = 0;                                  // makespan before and after
    for (int q = 0; q < m; ++q)
    {
        if (L[q] > Cmax) Cmax = L[q];
        int newL = L[q];
        if (q == a) newL = L[q] - delta;
        if (q == b) newL = L[q] + delta;
        if (newL > newC) newC = newL;
    }
    return newC - Cmax;
}
```

```
// one step of the move local search for P||Cmax
// mach[j] is the machine of job j, L[h] the load of machine h
// obj = CMAX (makespan) or SQ (sum of the squared loads); both are minimized

bool PCmaxMoveStep(int n, int m, const Vec& t, Vec& mach, Vec& L, int obj, int rule)
{
    int bestDelta = 0;                                       // best change of the objective found so far
    int bj = -1, bh = -1;                                    // best move found so far
    bool stop = false;
    for (int j = 0; j < n && !stop; ++j)
    {
        int a = mach[j];                                     // machine of job j
        for (int h = 0; h < m; ++h)
        {
            if (h == a) continue;
            int delta = ObjectiveChange(m, L, a, h, t[j], obj);   // job j from a to h
            if (delta < bestDelta)
            {
                bestDelta = delta; bj = j; bh = h;           // improving move
                if (rule == FIRST) { stop = true; break; }
            }
        }
    }
    if (bj == -1) return false;                              // no improving move: local optimum
```

```
25
26      L[mach[bj]] = L[mach[bj]] - t[bj];                  // apply the move
27      L[bh] = L[bh] + t[bj];
28      mach[bj] = bh;
29      return true;
30  }
```

```
1   // one step of the swap local search for P||Cmax
2   // the move (j, k) exchanges the machines of the jobs j and k
3   // obj = CMAX (makespan) or SQ (sum of the squared loads); both are minimized
4
5   bool PCmaxSwapStep(int n, int m, const Vec& t, Vec& mach, Vec& L, int obj, int
        rule)
6   {
7       int bestDelta = 0;                                  // best change of the
        objective found so far
8       int bj = -1, bk = -1;                               // best move found so
        far
9       bool stop = false;
10      for (int j = 0; j < n && !stop; ++j)
11      {
12          for (int k = j + 1; k < n; ++k)
13          {
14              int a = mach[j], b = mach[k];
15              if (a == b || t[j] == t[k]) continue;       // the loads would not
        change
16              int delta = ObjectiveChange(m, L, a, b, t[j] - t[k], obj);   // t_j
         - t_k from a to b
17              if (delta < bestDelta)
18              {
19                  bestDelta = delta; bj = j; bk = k;      // improving move
20                  if (rule == FIRST) { stop = true; break; }
21              }
22          }
23      }
24      if (bj == -1) return false;                         // no improving move:
        local optimum
25
26      int a = mach[bj], b = mach[bk];                     // apply the move
27      L[a] = L[a] - (t[bj] - t[bk]);
28      L[b] = L[b] + (t[bj] - t[bk]);
29      mach[bj] = b; mach[bk] = a;
30      return true;
31  }
```

- I **passi move** e **swap** per il BPP:

```
1   // one step of the move local search for the BPP
2   // bin[i] is the bin of item i, L[b] the load of bin b (L[b] = 0 if bin b is
        empty)
3   // obj = NB (number of bins, minimized) or SQ (sum of the squared loads,
        maximized)
4
```

```
bool BPPMoveStep(int n, const Vec& w, int c, Vec& bin, Vec& L, int obj, int
    rule)
{
    int bestGain = 0;                                   // best gain found so
    far
    int bi = -1, bb = -1;                               // best move found so
    far
    bool stop = false;
    for (int i = 0; i < n && !stop; ++i)
    {
        int a = bin[i];                                 // bin of item i
        for (int b = 0; b < (int)L.size(); ++b)
        {
            if (b == a || L[b] == 0) continue;          // same bin, or empty
    bin
            if (L[b] + w[i] > c) continue;              // item i does not fit
    into bin b
            int gain;                                   // improvement of the
    objective
            if (obj == SQ)
                gain = 2 * w[i] * (L[b] - L[a] + w[i]);
            else
                gain = (L[a] == w[i]) ? 1 : 0;          // bin a becomes empty
            if (gain > bestGain)
            {
                bestGain = gain; bi = i; bb = b;        // improving move
                if (rule == FIRST) { stop = true; break; }
            }
        }
    }
    if (bi == -1) return false;                         // no improving move:
    local optimum

    L[bin[bi]] = L[bin[bi]] - w[bi];                    // apply the move
    L[bb] = L[bb] + w[bi];
    bin[bi] = bb;
    return true;
}
```

```
// one step of the swap local search for the BPP, with the auxiliary objective
// sum of the squared loads (maximized); a swap never changes the number of
    bins
// the move (i, k) exchanges the bins of the items i and k

bool BPPSwapStep(int n, const Vec& w, int c, Vec& bin, Vec& L, int rule)
{
    int bestGain = 0;                                   // best gain found so
    far
    int bi = -1, bk = -1;                               // best move found so
    far
    bool stop = false;
    for (int i = 0; i < n && !stop; ++i)
    {
        for (int k = i + 1; k < n; ++k)
        {
            int a = bin[i], b = bin[k];
```

```
            if (a == b || w[i] == w[k]) continue;        // the loads would not change
            int dw = w[i] - w[k];                        // weight transferred from a to b
            if (L[b] + dw > c || L[a] - dw > c) continue;   // a bin would be overfull
            int gain = 2 * dw * (L[b] - L[a] + dw);      // improvement of the objective
            if (gain > bestGain)
            {
                bestGain = gain; bi = i; bk = k;         // improving move
                if (rule == FIRST) { stop = true; break; }
            }
        }
    }
    if (bi == -1) return false;                          // no improving move: local optimum

    int a = bin[bi], b = bin[bk], dw = w[bi] - w[bk];    // apply the move
    L[a] = L[a] - dw;
    L[b] = L[b] + dw;
    bin[bi] = b; bin[bk] = a;
    return true;
}
```

- Il **passo flip+swap** e il **passo penalty flip** per il KP (la penalità $\alpha =$ `alpha_num`/`alpha_den` è una frazione, e tutti i valori sono moltiplicati per `alpha_den` per lavorare con interi):

```
// one step of the local search for the KP with the neighbourhood flip + swap,
// restricted to feasible solutions; x[i] = 1 if item i is selected
// the flips are examined first, then the swaps (item i out, item k in)

bool KPFlipSwapStep(int n, const Vec& p, const Vec& w, int c, Vec& x, int rule)
{
    int W = 0;                                           // weight of the current solution
    for (int i = 0; i < n; ++i) W = W + w[i] * x[i];

    int bestDp = 0;                                      // best profit increase found so far
    int bi = -1, bk = -1;                                // best move found so far (bk = -1: flip)
    bool stop = false;
    for (int i = 0; i < n && !stop; ++i)                 // flip of item i
    {
        int dp = (1 - 2 * x[i]) * p[i];                  // remove, or add, item i
        int dw = (1 - 2 * x[i]) * w[i];
        if (W + dw <= c && dp > bestDp)
        {
            bestDp = dp; bi = i; bk = -1;                // improving move
            if (rule == FIRST) stop = true;
        }
```

```
    }
    for (int i = 0; i < n && !stop; ++i)                // swap: item i out, item k in
    {
        if (x[i] == 0) continue;
        for (int k = 0; k < n; ++k)
        {
            if (x[k] == 1) continue;
            int dp = p[k] - p[i];
            int dw = w[k] - w[i];
            if (W + dw <= c && dp > bestDp)
            {
                bestDp = dp; bi = i; bk = k;            // improving move
                if (rule == FIRST) { stop = true; break; }
            }
        }
    }
    if (bi == -1) return false;                         // no improving move: local optimum

    x[bi] = 1 - x[bi];                                  // apply the flip, or the swap
    if (bk != -1) x[bk] = 1;
    return true;
}
```

```
// one step of the flip local search for the KP with a penalty for the excess weight:
// g(x) = profit(x) - alpha * max(0, weight(x) - c), with alpha = alpha_num / alpha_den
// infeasible solutions are allowed; all the values are multiplied by alpha_den

bool KPPenaltyFlipStep(int n, const Vec& p, const Vec& w, int c, int alpha_num, int alpha_den, Vec& x, int rule)
{
    int P = 0, W = 0;                                   // profit and weight of the current solution
    for (int i = 0; i < n; ++i) { P = P + p[i] * x[i]; W = W + w[i] * x[i]; }
    int excess = (W > c) ? W - c : 0;
    int g = alpha_den * P - alpha_num * excess;         // alpha_den * g(x)

    int bestDelta = 0;                                  // best increase of alpha_den * g found so far
    int bi = -1;                                        // best move found so far
    for (int i = 0; i < n; ++i)                         // flip of item i
    {
        int newP = P + (1 - 2 * x[i]) * p[i];
        int newW = W + (1 - 2 * x[i]) * w[i];
        int newExcess = (newW > c) ? newW - c : 0;
        int delta = alpha_den * newP - alpha_num * newExcess - g;
        if (delta > bestDelta)
        {
            bestDelta = delta; bi = i;                  // improving move
            if (rule == FIRST) break;
        }
    }
```

```
    if (bi == -1) return false;                          // no improving move: local optimum

    x[bi] = 1 - x[bi];                                   // apply the flip
    return true;
}
```

- Il **2-opt multi-start**, il generatore di Park–Miller e i giri casuali:

```
// multi-start local search for the TSP: M random tours, each improved by 2-opt
// the random tours are generated by the Park-Miller generator, starting from seed
// best receives the best local optimum found; the function returns its length

int MultiStartTwoOpt(int n, const Mat& d, int M, long long seed, int rule, Vec& best)
{
    int bestLength = -1;                                 // length of the best tour found so far
    for (int s = 1; s <= M; ++s)
    {
        Vec pi = RandomTour(n, seed);                    // random starting tour
        LocalSearch([&]() { return TwoOptStep(n, d, pi, rule); });
        int length = TourLength(n, d, pi);               // length of the local optimum
        if (bestLength == -1 || length < bestLength)
        {
            bestLength = length;                         // new best tour
            best = pi;
        }
    }
    return bestLength;
}
```

```
// Park-Miller minimal standard generator: seed = 16807 * seed mod (2^31 - 1)
// returns u = seed / (2^31 - 1), a pseudo-random number in (0, 1)

double ParkMiller(long long& seed)
{
    seed = (16807LL * seed) % 2147483647LL;
    return (double)seed / 2147483647.0;
}
```

```
// pseudo-random integer in {0, 1, ..., k-1}: floor(u * k), computed in exact integer arithmetic

int RandomInt(long long& seed, int k)
{
    ParkMiller(seed);
    return (int)((seed * (long long)k) / 2147483647LL);
}
```

```
// random tour with city 0 in the first position (Fisher-Yates shuffle of the
    positions 1..n-1)

Vec RandomTour(int n, long long& seed)
{
    Vec pi(n);
    for (int i = 0; i < n; ++i) pi[i] = i;
    for (int i = n - 1; i >= 2; --i)
    {
        int j = 1 + RandomInt(seed, i);                  // j in {1, ..., i}
        int tmp = pi[i]; pi[i] = pi[j]; pi[j] = tmp;
    }
    return pi;
}
```

Capitolo 4

Metaeuristiche

Prerequisiti

- Da questo modulo, il capitolo «Ricerca locale»: intorni, first e best improvement, variable neighbourhood descent e multi-start.
- Da questo modulo, i capitoli «Euristiche costruttive» (costruzione randomizzata) e «Algoritmi euristici: definizioni» (intensificazione e diversificazione, valutazione sperimentale).

Obiettivi

Alla fine di questo capitolo saremo in grado di:

- eseguire a mano simulated annealing, tabu search, iterated local search e variable neighbourhood search, con numeri casuali assegnati;
- eseguire GRASP, large neighbourhood search e un algoritmo genetico o memetico su istanze piccole;
- spiegare come ciascun metodo esce da un ottimo locale, e quali parametri lo regolano;
- confrontare i metodi sulle stesse istanze, con più semi;
- dire che cosa le metaeuristiche non danno: un bound duale.

4.1 Metaeuristiche

Definizione 4.1: Metaeuristica

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.

- Una metaeuristica è uno *schema*, non un algoritmo: un algoritmo si ottiene scegliendo la rappresentazione, gli intorni, la ricerca locale e i parametri.
- Ogni metaeuristica bilancia **intensificazione** e **diversificazione**, definite nel capitolo «Algoritmi euristici: definizioni».

Una ricerca locale si ferma nel primo ottimo locale; una metaeuristica ne esce accettando soluzioni peggiori, saltando a una nuova soluzione di partenza, o combinando più soluzioni buone.

4.2 I problemi e le ipotesi

- Le metaeuristiche sono illustrate sui quattro problemi del capitolo «Algoritmi euristici: definizioni»: il problema dello zaino 0/1 (KP), il problema del bin packing (BPP), il problema del commesso viaggiatore (TSP) e $P\,||\,C_{\max}$. Le loro definizioni e ipotesi sono nelle rispettive pagine; ne richiamiamo la notazione.

problema	dati	soluzione	obiettivo
KP	oggetti $\mathscr{I}$, profitti p_j, pesi w_j, capacità c	sottoinsieme	max profitto totale
BPP	oggetti $\mathscr{I}$, pesi w_j, capacità c	pattern P_b	min contenitori
TSP	città $\mathscr{V}$, distanze d_{ij}	giro σ	min lunghezza $L(\sigma)$
$P\,\|\|\,C_{\max}$	lavori $\mathscr{J}$, tempi t_j, macchine m	insiemi $J_1, \ldots, J_m$	min makespan

- Ogni istanza soddisfa le ipotesi del suo problema: $w_j \le c$ per tutti gli oggetti e $\sum_j w_j > c$ per il KP e il BPP; $n \ge 3$ e distanze simmetriche per il TSP; $n > m$ per $P\,||\,C_{\max}$. Per $P\,||\,C_{\max}$, $L_h = \sum_{j \in J_h} t_j$ è il **carico** della macchina h e $C_{\max} = \max_h L_h$.
- Come nel capitolo «Algoritmi euristici: definizioni», $\mathscr{A}(I)$ è il valore della soluzione calcolata dall'algoritmo $\mathscr{A}$ sull'istanza I, e $z(I)$ è il valore ottimo. Scriviamo $f(S)$ per il valore di una soluzione S, e descriviamo gli algoritmi per un problema di **minimo**; per il KP, problema di massimo, le disuguaglianze si invertono.

4.2.1 Gli intorni

- Tutte le metaeuristiche di questo capitolo sono costruite sugli intorni del capitolo «Ricerca locale». Li richiamiamo per nome, con le mosse che li definiscono; n è il numero di oggetti, di città o di lavori.

problema	nome	mossa	dimensione
KP	flip	aggiungere un oggetto allo zaino, o toglierne uno	n
	swap	togliere un oggetto selezionato e aggiungere un oggetto non selezionato	$\leq \frac{n^2}{4}$
BPP	move	spostare un oggetto in un altro contenitore	$\leq n \cdot nb$
	swap	scambiare due oggetti posti in contenitori diversi	$\leq \frac{n(n-1)}{2}$
TSP	2-opt	togliere due lati non adiacenti del giro e richiuderlo con gli altri due lati, invertendo il segmento compreso fra loro	$\frac{n(n-3)}{2}$
	Or-opt	spostare un segmento di 1, 2 o 3 città consecutive in un'altra posizione	$O(n^2)$
$P \,\|\|\, C_{\max}$	move	spostare un lavoro su un'altra macchina	$n(m-1)$
	swap	scambiare due lavori assegnati a macchine diverse	$\leq \frac{n(n-1)}{2}$

- Un giro del TSP è memorizzato come una permutazione $\sigma = (\sigma_1, \ldots, \sigma_n)$ con $\sigma_1 = 1$. La **mossa 2-opt** (i, j), con $1 \leq i$, $i + 2 \leq j \leq n$ e $(i, j) \neq (1, n)$, toglie i lati (σ_i, σ_{i+1}) e (σ_j, σ_{j+1}) (con $\sigma_{n+1} = \sigma_1$), aggiunge i lati (σ_i, σ_j) e $(\sigma_{i+1}, \sigma_{j+1})$, e inverte il segmento $\sigma_{i+1}, \ldots, \sigma_j$. Il suo effetto sulla lunghezza si calcola in $O(1)$:

$$\delta(i, j) \;=\; d_{\sigma_i \sigma_j} + d_{\sigma_{i+1} \sigma_{j+1}} - d_{\sigma_i \sigma_{i+1}} - d_{\sigma_j \sigma_{j+1}}.$$

- Gli obiettivi del BPP e di $P \,||\, C_{\max}$ sono **piatti**: la maggior parte delle mosse non cambia il numero di contenitori, né il makespan. Due soluzioni con lo stesso valore si confrontano allora con un **obiettivo ausiliario**:
  - BPP: la somma dei quadrati dei carichi $\sum_b \left(\sum_{j \in P_b} w_j\right)^2$, da *massimizzare*: cresce quando gli oggetti si concentrano in meno contenitori, più pieni;
  - $P \,||\, C_{\max}$: la somma dei quadrati dei carichi $\sum_h L_h^2$, da *minimizzare*: diminuisce quando i carichi si bilanciano.

  Per $P \,||\, C_{\max}$ confrontiamo due soluzioni in modo **lessicografico**: prima per $C_{\max}$, e a parità di makespan per $\sum_h L_h^2$. Per move e swap questo seleziona le stesse mosse miglioranti dell'obiettivo ausiliario $\sum_h L_h^2$ del capitolo «Ricerca locale»: un trasferimento che diminuisce $\sum_h L_h^2$ non aumenta mai $C_{\max}$, e un trasferimento che diminuisce $C_{\max}$ diminuisce anche $\sum_h L_h^2$.

- Per il KP l'intorno flip contiene soluzioni inammissibili (aggiungere un oggetto può superare la capacità). Una ricerca locale le scarta oppure le accetta con una **penalità** sul peso in eccesso; in questo capitolo compaiono entrambe le opzioni.

- Le **ricerche locali** usate dentro le metaeuristiche sono le discese del capitolo «Ricerca locale»: TwoOptLS (2-opt, best improvement) per il TSP, MoveDescent (move, best improvement, confronto lessicografico) per $P\,||\,C_{\max}$, e KPLocalSearch (flip e swap ammissibili, best improvement) per il KP. In ogni discesa best improvement, i pareggi fra mosse ugualmente buone si risolvono a favore della prima mossa trovata, scorrendo gli indici in ordine crescente.

4.2.2 Numeri casuali

- La maggior parte delle metaeuristiche fa **scelte casuali**. Per rendere riproducibili ogni esempio e ogni esercizio, tutti i numeri casuali di questo capitolo, e del suo codice `C++`, sono prodotti dal **minimal standard generator** di Park e Miller (1988):

$$seed \leftarrow 16807 \cdot seed \bmod (2^{31}-1), \qquad u = \frac{seed}{2^{31}-1} \in (0,1).$$

Una chiamata Rand() aggiorna *seed* e restituisce u. Un intero casuale in $\{0,1,\dots,K-1\}$ è $\lfloor u \cdot K \rfloor$, e un elemento casuale di una lista di K elementi è l'elemento in posizione $1+\lfloor u \cdot K \rfloor$.

- Il valore iniziale di *seed*, il **seme**, è un intero in $\{1,\dots,2^{31}-2\}$. Lo stesso seme dà sempre la stessa sequenza di numeri; semi diversi danno esecuzioni diverse dello stesso algoritmo. Con $seed = 1$ i primi numeri sono, arrotondati a quattro decimali,

$$0.0000,\ 0.1315,\ 0.7556,\ 0.4587,\ 0.5328,\ 0.2190,\ 0.0470,\ 0.6789,\ \dots$$

  Il primo numero prodotto da un seme piccolo è sempre molto vicino a 0; è una proprietà del generatore, non un errore.

- Negli esempi svolti i numeri casuali sono scritti nel testo, arrotondati a quattro decimali, così che ogni passo si possa verificare a mano; l'arrotondamento non cambia mai una decisione.

4.2.3 Le istanze

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

problema	istanza di riferimento	valore ottimo e soluzione
KP	$p=(10,7,6,4)$, $w=(5,4,3,3)$, $c=9$	$z(I)=17$, oggetti $\{1,2\}$
BPP	$w=(4,4,5,3,2,3)$, $c=7$	$z(I)=3$, pattern $\{1,4\},\{2,6\},\{3,5\}$
TSP	5 città, distanze d_{ij} qui sotto	$z(I)=18$, giro $(1,3,5,2,4)$
$P\,||\,C_{\max}$	$t=(5,5,4,4,3,3,3)$, $m=3$	$z(I)=9$, insiemi $\{1,3\},\{2,4\},\{5,6,7\}$

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

- Le istanze di riferimento sono abbastanza piccole da seguire a mano ogni iterazione, ma troppo piccole per mostrare il comportamento di una metaeuristica su migliaia di iterazioni. Per gli esperimenti usiamo un'**istanza media** per problema, generata una volta con il generatore di numeri casuali della Sezione 4.2.2 e fissata qui sotto. I loro valori ottimi sono stati calcolati con un algoritmo esatto, e servono solo a misurare la qualità delle soluzioni euristiche.

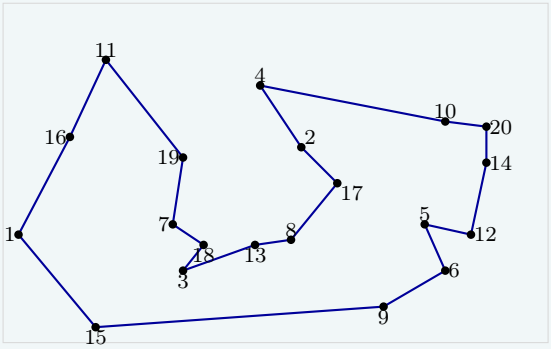
Esempio 4.1: L'istanza media del TSP (TSP20)

20 città nel piano, con coordinate intere (x_i, y_i), e distanze $d_{ij} = \lfloor \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} + \frac{1}{2} \rfloor$ (la distanza euclidea arrotondata all'intero più vicino):

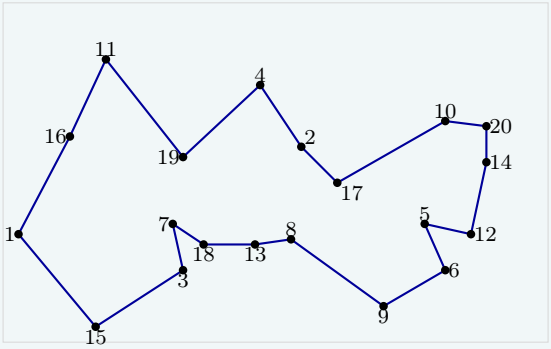
i	1	2	3	4	5	6	7	8	9	10
x_i	0	55	32	47	79	83	30	53	71	83
y_i	18	35	11	47	20	11	20	17	4	40

i	11	12	13	14	15	16	17	18	19	20
x_i	17	88	46	91	15	10	62	36	32	91
y_i	52	18	16	32	0	37	28	16	33	39

nearest neighbour dalla città 1: $L(\sigma) = 326$



un giro ottimo: $L(\sigma) = 291$



Il giro nearest neighbour dalla città 1 ha lunghezza 326; la ricerca locale 2-opt avviata da esso si ferma in un ottimo locale di lunghezza 319, dopo due mosse miglioranti. Il valore ottimo è $z(I) = 291$: l'ottimo locale è superiore del 9.6%.

Esempio 4.2: Le istanze medie del KP, del BPP e di $P\,||\,C_{\max}$

KP20: $n = 20$ oggetti, capacità $c = 389$ (metà del peso totale, arrotondata per difetto),

j	1	2	3	4	5	6	7	8	9	10
p_j	54	55	55	52	58	42	53	53	22	64
w_j	39	43	49	47	44	35	43	44	16	50
j	11	12	13	14	15	16	17	18	19	20
p_j	54	51	39	49	53	46	28	64	58	34
w_j	40	42	28	38	46	39	14	50	45	27

L'algoritmo greedy per efficienza trova profitto 490, e la ricerca locale con flip e swap avviata da esso arriva a 514; il valore ottimo è $z(I) = 521$.

PCM15: $m = 4$ macchine e $n = 15$ lavori con tempi di esecuzione

$$t = (10, 23, 26, 16, 23, 11, 14, 18, 18, 24, 29, 22, 13, 13, 15).$$

Il tempo totale è 275, quindi $z(I) \geq \lceil 275/4 \rceil = 69$, ed esiste un assegnamento di makespan 69: $z(I) = 69$. LPT trova makespan 73.

BPP24: contenitori di capacità $c = 100$ e $n = 24$ oggetti con pesi

$$w = (30, 40, 57, 22, 37, 23, 34, 34, 55, 51, 21, 47, 22, 24, 22, 34, 20, 51, 25, 54, 35, 27, 55, 48).$$

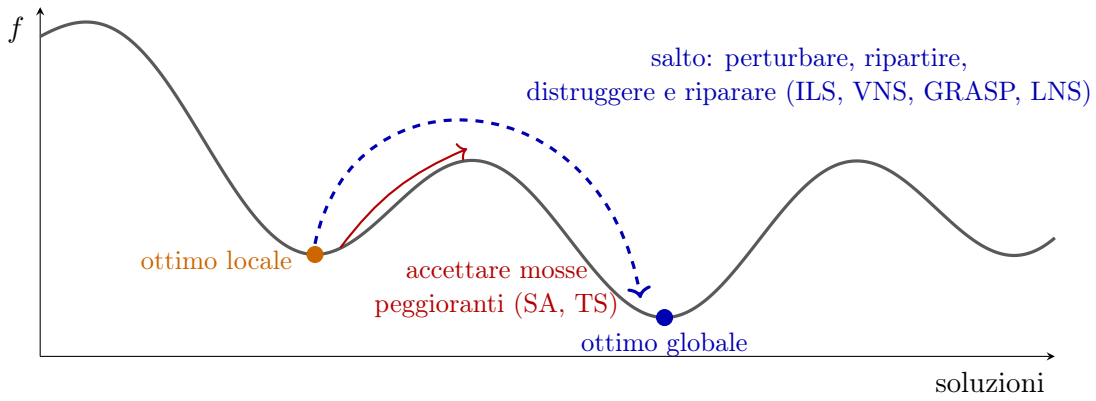
Il peso totale è 868, quindi servono almeno $\lceil 868/100 \rceil = 9$ contenitori, ed esiste un impacchettamento in 9 contenitori: $z(I) = 9$. FF e FFD usano entrambi 10 contenitori.

4.3 Come uscire da un ottimo locale

- Una ricerca locale passa da una soluzione a un vicino migliore, e si ferma nel primo **ottimo locale**: una soluzione senza vicini migliori. Nel paesaggio della funzione obiettivo, scende fino al fondo della valle da cui parte, e lì rimane. È pura **intensificazione**.

- Ripetere la ricerca locale da molte soluzioni di partenza (multi-start) esplora molte valli, ma ogni partenza ignora tutto ciò che è stato appreso prima, e le valli buone si trovano di solito vicino ad altre valli buone.

- La tabella qui sotto elenca i modi per uscire da un ottimo locale; è la mappa del capitolo.



come uscire	idea	metaeuristica
accettare mosse peggioranti *a caso*	un vicino peggiore è accettato con una probabilità che diminuisce con il peggioramento e con il tempo	simulated annealing (Sezione 4.4)
accettare mosse peggioranti *in modo deterministico, con memoria*	passare al miglior vicino anche se è peggiore, e vietare il ritorno alle soluzioni appena visitate	tabu search (Sezione 4.5)
perturbare l'ottimo locale e ripartire	applicare una modifica casuale che la ricerca locale non può annullare, e scendere di nuovo	iterated local search (Sezione 4.6)
cambiare l'intorno	quando la ricerca è bloccata, usare un intorno più grande, di cui l'ottimo locale non è più un ottimo locale	variable neighbourhood search (Sezione 4.7)
ripartire da una costruzione randomizzata	costruire una soluzione buona diversa a ogni ripartenza, e migliorarla	GRASP (Sezione 4.8)
distruggere e riparare	togliere una parte grande della soluzione e ricostruirla con un'euristica o un algoritmo esatto	large neighbourhood search (Sezione 4.9)
lavorare con una *popolazione*	combinare le parti di più soluzioni buone in soluzioni nuove	algoritmi genetici e memetici (Sezione 4.10)

- I primi due modi mantengono **una sola soluzione corrente** e cambiano la **regola di accettazione** della ricerca locale: escono dalla valle passando per soluzioni peggiori. I quattro successivi **saltano**: generano un nuovo punto di partenza, abbastanza lontano da lasciare la valle ma abbastanza vicino da conservare ciò che è buono. L'ultimo mantiene **molte soluzioni** contemporaneamente.

- Ogni metaeuristica deve bilanciare **intensificazione** (scendere fino al fondo della valle corrente) e **diversificazione** (lasciarla per un'altra). Per ciascun metodo indichiamo quale componente fa che cosa.

- Una metaeuristica è uno *schema*, non un algoritmo: per ottenere un algoritmo occorre scegliere la rappresentazione, gli intorni, la ricerca locale e i parametri, e queste scelte contano almeno quanto lo schema. Presentiamo ogni metaeuristica sui problemi in cui la sua idea è più naturale, non su tutti e quattro.

4.4 Simulated annealing

Il *simulated annealing* (indicato con SA) è una ricerca locale che sceglie un vicino *casuale* della soluzione corrente e vi si sposta se è migliore; se è peggiore, vi si sposta comunque con una probabilità che diminuisce con il peggioramento e con un parametro T, la *temperatura*. La temperatura parte alta, così che all'inizio quasi ogni mossa è accettata, e viene abbassata gradualmente, così che alla fine sono accettate solo mosse miglioranti. Si restituisce la migliore soluzione visitata.

- Lo pseudocodice di SA, per un problema di minimo con obiettivo f e intorno N, è il seguente:

Algoritmo

Algorithm: Simulated annealing

Input: soluzione di partenza S, temperatura iniziale T_0, fattore di raffreddamento $\alpha \in (0,1)$, numero L di mosse per temperatura, temperatura finale $T_{\min}$

Output: migliore soluzione trovata

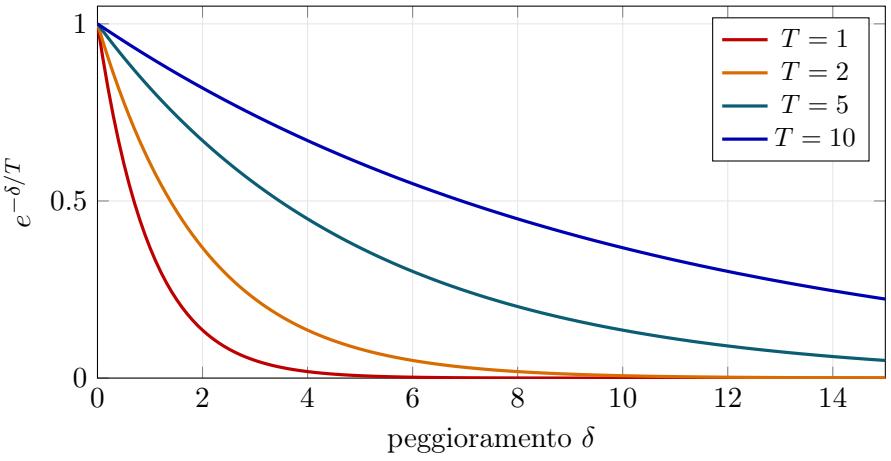
```
1  Function SimulatedAnnealing(S, T_0, α, L, T_min)
2      cur := S;  fcur := f(cur)                          // soluzione corrente
3      best := cur;  fbest := fcur                        // migliore soluzione trovata
4      T := T_0                                           // temperatura
5      while T > T_min do
6          for l := 1 to L do
7              S' := vicino casuale in N(cur)
8              δ := f(S') − fcur
9              v := RAND()
10             if δ ≤ 0 or v < e^{−δ/T} then
11                 cur := S';  fcur := fcur + δ           // mossa accettata
12                 if fcur < fbest then
13                     best := cur;  fbest := fcur
14                 end
15             end
16         end
17         T := α · T                                     // raffreddamento
18     end
19     return best
```

- Il nome viene dalla metallurgia: l'*annealing* (ricottura) è il processo che scalda un metallo e lo raffredda lentamente, così che i suoi atomi si dispongano in uno stato di energia minima; raffreddato troppo in fretta, il metallo si blocca in uno stato disordinato di energia più alta. Kirkpatrick, Gelatt e Vecchi (1983) e, indipendentemente, Černý (1985) hanno proposto di simulare questo processo su un problema di ottimizzazione, con l'obiettivo f nel ruolo dell'energia. La regola di accettazione è quella usata da Metropolis et al. (1953) per simulare gli atomi.

Regola di accettazione di Metropolis. Alla temperatura T, un vicino S' della soluzione corrente, con $\delta = f(S') - f(cur)$, è accettato con probabilità

$$P(\text{accettare } S') = \begin{cases} 1 & \text{se } \delta \leq 0, \\ e^{-\delta/T} & \text{se } \delta > 0. \end{cases}$$

- La probabilità si realizza estraendo $v = \textsc{Rand}()$, uniforme in $(0, 1)$, e accettando se $v < e^{-\delta/T}$, il che accade con probabilità $e^{-\delta/T}$.
- Una mossa peggiorante è accettata più facilmente se il peggioramento δ è **piccolo**, e se la temperatura T è **alta**. Per $T \to \infty$ ogni mossa è accettata, e SA diventa una passeggiata casuale (pura diversificazione); per $T \to 0$ sono accettate solo le mosse con $\delta \leq 0$, e SA diventa una ricerca locale con scelta casuale del vicino (pura intensificazione). Abbassare la temperatura porta gradualmente dal primo comportamento al secondo.
- Le mosse con $\delta = 0$ sono sempre accettate: SA si muove liberamente sui plateau degli obiettivi piatti.
- SA non si ferma mai in un ottimo locale: un ottimo locale viene lasciato appena una mossa peggiorante è estratta e accettata. Per questo la soluzione **migliore** visitata va memorizzata a parte: la soluzione corrente alla fine non è necessariamente la migliore.
- A ogni iterazione si valuta **un solo vicino casuale**, non l'intero intorno: un'iterazione di SA costa pochissimo, e SA esegue molte più iterazioni di una ricerca locale. Per il TSP, una mossa 2-opt casuale si estrae numerando le $M = \frac{n(n-3)}{2}$ mosse (i, j) in ordine lessicografico e scegliendo quella in posizione $1 + \lfloor \textsc{Rand}() \cdot M \rfloor$; il suo δ si calcola in $O(1)$.



- La figura mostra la probabilità di accettazione in funzione del peggioramento, per quattro temperature. A $T = 10$ un peggioramento di 5 è accettato con probabilità $e^{-0.5} \approx 0.61$; a $T = 1$ con probabilità $e^{-5} \approx 0.007$. La temperatura ha la stessa unità di misura dell'obiettivo: T è «grande» o «piccola» solo rispetto ai valori tipici di δ.

4.4.1 Lo schema di raffreddamento e i parametri

- Lo **schema di raffreddamento** è la successione delle temperature. Lo pseudocodice usa lo schema **geometrico** $T \leftarrow \alpha T$, il più comune in pratica, con α fra 0.8 e 0.99, e L mosse a ogni temperatura. Altri schemi sono quello *lineare*, $T_k = T_0 - k\beta$, e quello *logaritmico*, $T_k = \frac{\Gamma}{\ln(k+1)}$, che conta nella teoria di SA ma è decisamente troppo lento in pratica.
- **Temperatura iniziale T_0.** Deve essere abbastanza alta da accettare all'inizio la maggior parte delle mosse peggioranti. Una regola comune fissa un tasso di accettazione iniziale obiettivo χ_0 (per esempio $\chi_0 = 0.8$), stima il peggioramento medio $\bar{\delta}$ di un campione di mosse casuali, e pone $T_0 = -\bar{\delta}/\ln \chi_0$, così che $e^{-\bar{\delta}/T_0} = \chi_0$.
- **Mosse per temperatura L.** Bisogna dare alla ricerca il tempo di adattarsi a ogni temperatura; L si prende spesso proporzionale alla dimensione dell'intorno.
- **Criterio di arresto.** Lo pseudocodice si ferma quando $T \leq T_{\min}$. Con lo schema geometrico il numero di temperature è il più piccolo intero K con $T_0\alpha^K \leq T_{\min}$, cioè $K = \lceil \ln(T_{\min}/T_0)/\ln\alpha \rceil$ quando il rapporto non è intero, e il numero di iterazioni è $K \cdot L$. Altri criteri comuni si fermano dopo un certo numero di temperature senza mosse accettate, o a un limite di tempo.
- Un raffreddamento lento (α vicino a 1) dà soluzioni migliori ma più iterazioni: i parametri scambiano qualità con tempo, come in ogni euristica.

4.4.2 Esempi

Esempio 4.3: *SA* con l'intorno 2-opt sull'istanza di riferimento del TSP

Partiamo dal giro calcolato dall'euristica nearest neighbour dalla città 1 nel capitolo «Euristiche costruttive», $\sigma = (1, 3, 2, 4, 5)$ di lunghezza 25, con $T_0 = 5$, $\alpha = 0.8$, $L = 2$ e $T_{\min} = 3$: le temperature sono 5, 4 e 3.2 (poi $2.56 \leq 3$ ferma la ricerca), per un totale di 6 iterazioni. Le $M = 5$ mosse 2-opt su 5 città sono numerate

$$1 : (1, 3), \quad 2 : (1, 4), \quad 3 : (2, 4), \quad 4 : (2, 5), \quad 5 : (3, 5),$$

e i numeri casuali, con *seed* $= 1$, sono quelli della Sezione 4.2.2: il primo numero di ogni iterazione (u) sceglie la mossa $1 + \lfloor 5u \rfloor$, il secondo (v) serve nel test di accettazione.

it	T	u	mossa	δ	v	$e^{-\delta/T}$	decisione	giro corrente	L
1	5	0.0000	$1 : (1, 3)$	4	0.1315	0.4493	accetta	$(1, 2, 3, 4, 5)$	29
2	5	0.7556	$4 : (2, 5)$	-7	—	—	accetta	$(1, 2, 5, 4, 3)$	22
3	4	0.5328	$3 : (2, 4)$	2	0.2190	0.6065	accetta	$(1, 2, 4, 5, 3)$	24
4	4	0.0470	$1 : (1, 3)$	-6	—	—	accetta	$(1, 4, 2, 5, 3)$	18
5	3.2	0.6793	$4 : (2, 5)$	4	0.9347	0.2865	rifiuta	$(1, 4, 2, 5, 3)$	18
6	3.2	0.3835	$2 : (1, 4)$	4	0.5194	0.2865	rifiuta	$(1, 4, 2, 5, 3)$	18

Alle iterazioni 1 e 3 viene accettato un giro **peggiore** ($25 \to 29$ e $22 \to 24$), perché v è minore della probabilità di accettazione; alle iterazioni 5 e 6 la temperatura è più bassa, le mosse peggioranti hanno meno probabilità di essere accettate, ed entrambe sono rifiutate. Il giro migliore, $(1, 4, 2, 5, 3)$, è il giro ottimo $(1, 3, 5, 2, 4)$ percorso nel verso opposto: $SA(I) = 18 = z(I)$.

- Su questa istanza piccola il giro ottimo è l'unico ottimo locale 2-opt fra i 12 giri, quindi anche una ricerca locale 2-opt lo raggiungerebbe: l'esempio mostra il meccanismo di SA; la sua capacità di uscire dagli ottimi locali si vede su istanze con ottimi locali cattivi.

4.4.3 Soluzioni inammissibili: simulated annealing con penalità

- Per il KP, l'intorno flip di una soluzione vicina alla capacità contiene molte soluzioni inammissibili. Invece di scartarle, SA può **visitarle**, con una penalità sul peso in eccesso che le rende meno attraenti:

$$f_\beta(x) \;=\; \sum_{j\in\mathscr{I}} p_j x_j \;-\; \beta\cdot\max\Big\{0,\; \sum_{j\in\mathscr{I}} w_j x_j - c\Big\}, \qquad \beta>0.$$

SA massimizza f_β su tutti gli $x\in\{0,1\}^n$; il peggioramento di una mossa da x a y è $\delta = f_\beta(x) - f_\beta(y)$, e la soluzione migliore si aggiorna solo con soluzioni **ammissibili**.

- Su una soluzione ammissibile f_β è il profitto. Passando per una soluzione inammissibile, SA può raggiungere soluzioni ammissibili che non sono raggiungibili con soli flip ammissibili: la regione inammissibile fa da **ponte** fra le valli.

- Il **coefficiente di penalità** β è un parametro: se è troppo piccolo la ricerca vaga nella regione inammissibile, dove il profitto è maggiore; se è troppo grande le soluzioni inammissibili non sono mai accettate, e il ponte è chiuso.

Esempio 4.4: SA con penalità sull'istanza di riferimento del KP

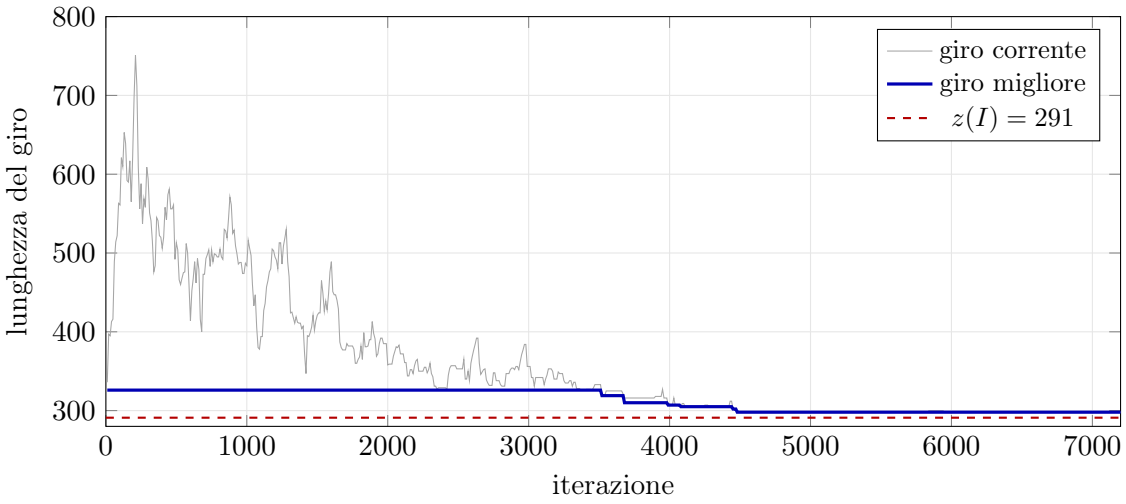
Partiamo da $x=(1,0,1,0)$, la soluzione $\{1,3\}$ dell'algoritmo greedy per efficienza, di profitto 16. È un ottimo locale per l'intorno flip (capitolo «Algoritmi euristici: definizioni»): i suoi vicini ammissibili hanno profitto 6 e 10. Usiamo $\beta=3$, $T_0=4$, $\alpha=0.8$, $L=2$, $T_{\min}=2.5$ (temperature 4, 3.2, 2.56: 6 iterazioni) e $seed=3$. A ogni iterazione u sceglie l'oggetto $k=1+\lfloor 4u\rfloor$ da invertire, e v serve nel test di accettazione.

it	T	u	k	y	$\sum w_j y_j$	$f_3(y)$	δ	v vs $e^{-\delta/T}$	decisione
1	4	0.0000	1	$(0,0,1,0)$	3	6	10	$0.3946 \geq 0.0821$	rifiuta
2	4	0.2668	2	$(1,1,1,0)$	12	14	2	$0.3760 < 0.6065$	accetta
3	3.2	0.5983	3	$(1,1,0,0)$	9	17	-3	—	accetta
4	3.2	0.1411	1	$(0,1,0,0)$	4	7	10	$0.0366 < 0.0439$	accetta
5	2.56	0.0379	1	$(1,1,0,0)$	9	17	-10	—	accetta
6	2.56	0.1505	1	$(0,1,0,0)$	4	7	10	$0.5582 \geq 0.0201$	rifiuta

All'iterazione 2 la ricerca passa alla soluzione **inammissibile** $(1,1,1,0)$, di peso $12>9$: il suo profitto è 23 e il suo valore penalizzato $23-3\cdot 3=14$, un peggioramento di 2 accettato con probabilità 0.6065. Da lì, togliendo l'oggetto 3 si ottiene la soluzione ammissibile $(1,1,0,0)$ di profitto 17: l'ottimo, $SA(I)=17=z(I)$. L'iterazione 4 mostra che a questa temperatura anche un peggioramento di 10 può essere accettato, con una probabilità piccola; l'iterazione 5 torna all'ottimo.

4.4.4 Un esperimento sull'istanza media del TSP

- Eseguiamo SA con l'intorno 2-opt su TSP20 (Esempio 4.1), dal giro nearest neighbour di lunghezza 326, con $L = 100$ e $T_{\min} = 0.5$. La figura mostra un'esecuzione, con $T_0 = 20$, $\alpha = 0.95$ e $seed = 1$: la lunghezza del giro corrente (grigio) e del giro migliore (blu) lungo le 7200 iterazioni, ogni 10 iterazioni. I dati sono prodotti dal codice `C++` di questo capitolo (`make data`).



- All'inizio la temperatura è alta rispetto al peggioramento tipico di una mossa 2-opt, e il giro corrente diventa molto **peggiore** di quello di partenza (fino a lunghezza 765): la ricerca è quasi una passeggiata casuale. Al diminuire di T, sono accettate meno mosse peggioranti e il giro corrente scende in una valle; alla fine la ricerca è una ricerca locale. Il giro migliore, di lunghezza 298, è trovato all'iterazione 4479: è migliore dell'ottimo locale 2-opt (319), ma non è ottimo.

- La tabella riporta, per 10 esecuzioni con $seed = 1, \ldots, 10$ e diversi parametri, il numero di iterazioni, la lunghezza minima, media e massima del giro migliore, e il numero di esecuzioni che raggiungono l'ottimo 291.

α	T_0	iterazioni	min	media	max	esecuzioni ottime
0.9	1	700	291	310.3	322	1
0.9	5	2200	291	297.3	305	4
0.9	20	3600	291	300.3	323	4
0.9	100	5100	291	297.8	311	3
0.95	1	1400	291	302.3	319	3
0.95	5	4500	291	294.3	298	5
0.95	20	7200	291	294.3	298	5
0.95	100	10400	291	293.2	299	7
0.99	1	6900	291	298.8	305	4
0.99	5	23000	291	291.7	298	9
0.99	20	36800	291	291.0	291	10
0.99	100	52800	291	291.0	291	10

- Con $T_0 = 1$ la temperatura è troppo bassa fin dall'inizio: sono accettate poche mosse peggioranti, e SA si comporta come una ricerca locale randomizzata. Un raffreddamento

più lento ($\alpha = 0.99$) passa più iterazioni in ogni intervallo di temperatura e raggiunge l'ottimo in quasi tutte le esecuzioni, al prezzo di molte più iterazioni. Una singola esecuzione dice poco: il confronto deve guardare la distribuzione su più semi, come ricordato nel capitolo «Algoritmi euristici: definizioni».

4.4.5 Proprietà

- A temperatura fissata T, la successione delle soluzioni correnti di SA è una catena di Markov: la soluzione successiva dipende solo da quella corrente. Se l'intorno è simmetrico ($S' \in N(S)$ se e solo se $S \in N(S')$, con tutti gli intorni della stessa dimensione) e connesso (ogni soluzione può raggiungere ogni altra attraverso mosse), la catena ha un'unica distribuzione stazionaria, la **distribuzione di Boltzmann**:

$$\pi_T(S) = \frac{e^{-f(S)/T}}{\sum_{S'} e^{-f(S')/T}} \qquad \text{per ogni soluzione } S.$$

- Per due soluzioni S_1 e S_2, $\frac{\pi_T(S_1)}{\pi_T(S_2)} = e^{(f(S_2)-f(S_1))/T}$: le soluzioni migliori sono più probabili, e tanto più quanto più bassa è la temperatura. Per $T \to 0$, π_T si concentra sulle soluzioni ottime.

- Hajek (1988) ha dimostrato che con lo schema logaritmico $T_k = \frac{\Gamma}{\ln(k+1)}$, con Γ almeno pari alla profondità della valle più profonda che non contiene una soluzione ottima, la probabilità che la soluzione corrente sia ottima tende a 1. Il risultato è una garanzia *al limite*: il numero di iterazioni necessarie può superare il numero di soluzioni, e in pratica si usa lo schema geometrico, più veloce, senza garanzie.

4.5 Tabu search

La *tabu search* (indicata con TS) esplora completamente l'intorno della soluzione corrente e si sposta sul *miglior* vicino, anche se è peggiore della soluzione corrente. Per evitare di tornare alle soluzioni appena visitate, mantiene una memoria a breve termine, la *lista tabu*: le mosse che annullerebbero le mosse recenti sono vietate (*tabu*) per un certo numero di iterazioni, la *durata tabu* (tenure), a meno che non portino a una soluzione migliore della migliore trovata (*criterio di aspirazione*).

- Lo pseudocodice di TS per $P\,||\,C_{\max}$, con l'intorno move e il confronto lessicografico della Sezione 4.2.1, è il seguente; l'attributo (j, h) vieta di riportare il lavoro j sulla macchina h.

Algoritmo

Algorithm: Tabu search per $P\,||\,C_{\max}$

Input: tempi $t[1..n]$, m macchine, assegnamento di partenza $mach[1..n]$, $tenure$, $maxIter$

Output: migliore assegnamento trovato

```
1  Function TabuSearch(t, m, mach, tenure, maxIter)
2      L := carichi di mach;  tabu[j][h] := 0 per ogni j, h
3      best := mach;  bestC := max_h L[h]
4      LB := max{max_j t[j], ⌈∑_j t[j]/m⌉}            // arresto se raggiunto
5      it := 0
6      while it < maxIter and bestC > LB do
7          it := it + 1;  bj := 0                     // migliore mossa non tabu (bj, bh)
8          for j := 1 to n do
9              for h := 1 to m, h ≠ mach[j] do
10                 L' := i carichi dopo lo spostamento del lavoro j da mach[j] a h
11                 if tabu[j][h] ≥ it and not max_k L'[k] < bestC then continue// tabu
12                 if bj = 0 or L' è lessicograficamente migliore di bL then
                       (bj, bh, bL) := (j, h, L')
13             end
14         end
15         if bj = 0 then break
16         tabu[bj][mach[bj]] := it + tenure           // vietare il ritorno
17         mach[bj] := bh;  L := bL                    // anche se peggiore
18         if max_k L[k] < bestC then
19             bestC := max_k L[k];  best := mach
20         end
21     end
22     return best
```

- La tabu search è stata proposta da Glover (1986); il libro di riferimento è Glover e Laguna, *Tabu Search*, Kluwer, 1997. A differenza di SA, è **deterministica**: esce da un ottimo locale perché si sposta sempre, sul vicino meno cattivo se nessun vicino è migliore.
- Senza memoria, questa regola **cicla**: da un ottimo locale S la ricerca passa al miglior vicino S', che è peggiore, e da S' il miglior vicino è tipicamente di nuovo S. La lista tabu spezza il ciclo vietando la mossa inversa.
- **Attributi.** Memorizzare le soluzioni visitate e confrontare con esse ogni vicino sarebbe troppo costoso. La lista tabu memorizza invece **attributi** delle mosse recenti: qui la coppia (j, h) «il lavoro j ha lasciato la macchina h»; per il TSP, i lati appena tolti dal giro.
- Un attributo vieta più della sola soluzione appena lasciata: ogni soluzione con il lavoro j sulla macchina h è tabu, anche quelle mai visitate. Il **criterio di aspirazione** rimedia a questo eccesso di prudenza: una mossa tabu è ammessa se porta a una soluzione migliore della migliore trovata, che certamente non è stata visitata.

- **Durata tabu.** Se è troppo breve, la ricerca può ancora ciclare su cicli più lunghi; se è troppo lunga, sono vietate troppe mosse e la ricerca viene spinta lontano dalle regioni buone. Valori tipici sono fra 5 e 20, o proporzionali a $\sqrt{n}$; estrarre la durata a caso in un intervallo a ogni iterazione (robust tabu search, Taillard 1991) rende improbabili i cicli.
- **Criterio di arresto.** Un numero massimo di iterazioni, o di iterazioni senza miglioramento della soluzione migliore. Per $P\,||\,C_{\max}$ possiamo anche fermarci quando il makespan raggiunge il lower bound LB: allora la soluzione è ottima, e la ricerca può solo perdere tempo.
- **Memoria a lungo termine.** Oltre alla lista tabu (memoria a breve termine), TS può registrare quanto spesso ogni attributo compare nelle soluzioni visitate. La *diversificazione* penalizza gli attributi frequenti, per spingere la ricerca verso regioni inesplorate; l'*intensificazione* riparte dalle migliori soluzioni trovate, e fissa gli attributi che hanno in comune.

4.5.1 Esempio

Esempio 4.5: TS sull'istanza di riferimento di $P\,||\,C_{\max}$

Partiamo dalla soluzione calcolata da LPT nel capitolo «Euristiche costruttive», $J_1 = \{1,5,7\}$, $J_2 = \{2,6\}$, $J_3 = \{3,4\}$, con carichi $(11,8,8)$, $C_{\max} = 11$ e $\sum_h L_h^2 = 249$. È un ottimo locale per l'intorno move: alla prima iterazione i 14 vicini sono

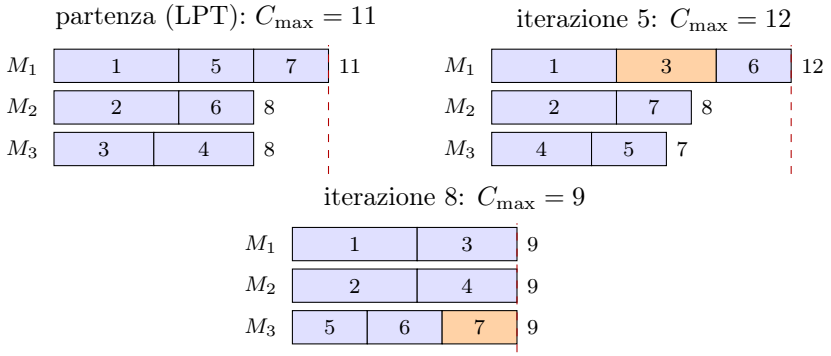
mossa	$1 \to M_2$	$1 \to M_3$	$2 \to M_1$	$2 \to M_3$	$3 \to M_1$	$3 \to M_2$	$4 \to M_1$
$(C_{\max}, \sum_h L_h^2)$	$(13, 269)$	$(13, 269)$	$(16, 329)$	$(13, 299)$	$(15, 305)$	$(12, 281)$	$(15, 305)$
mossa	$4 \to M_2$	$5 \to M_2$	$5 \to M_3$	$6 \to M_1$	$6 \to M_3$	$7 \to M_2$	$7 \to M_3$
$(C_{\max}, \sum_h L_h^2)$	$(12, 281)$	$(11, 249)$	$(11, 249)$	$(14, 285)$	$(11, 267)$	$(11, 249)$	$(11, 249)$

Nessun vicino ha $C_{\max} < 11$, e i quattro migliori hanno $C_{\max} = 11$ e $\sum_h L_h^2 = 249$, come la soluzione corrente: una ricerca locale si ferma qui. TS, con $tenure = 6$, si sposta invece sul miglior vicino non tabu (il primo dei quattro, il lavoro 5 su M_2), e prosegue:

it	mossa	carichi	$C_{\max}$	$\sum_h L_h^2$	$bestC$	nuovo attributo tabu
1	lavoro 5: $M_1 \to M_2$	$(8, 11, 8)$	11	249	11	$(5, M_1)$ fino all'it. 7
2	lavoro 5: $M_2 \to M_3$	$(8, 8, 11)$	11	249	11	$(5, M_2)$ fino all'it. 8
3	lavoro 6: $M_2 \to M_1$	$(11, 5, 11)$	11	267	11	$(6, M_2)$ fino all'it. 9
4	lavoro 7: $M_1 \to M_2$	$(8, 8, 11)$	11	249	11	$(7, M_1)$ fino all'it. 10
5	lavoro 3: $M_3 \to M_1$	$(12, 8, 7)$	12	257	11	$(3, M_3)$ fino all'it. 11
6	lavoro 6: $M_1 \to M_3$	$(9, 8, 10)$	10	245	10	$(6, M_1)$ fino all'it. 12
7	lavoro 4: $M_3 \to M_2$	$(9, 12, 6)$	12	261	10	$(4, M_3)$ fino all'it. 13
8	lavoro 7: $M_2 \to M_3$	$(9, 9, 9)$	9	243	9	$(7, M_2)$ fino all'it. 14

- All'iterazione 2 il miglior vicino riporterebbe il lavoro 5 su M_1, tornando alla soluzione di partenza: la mossa è tabu (attributo $(5, M_1)$ fino all'iterazione 7), e la ricerca sposta il lavoro 5 su M_3.
- All'iterazione 5 i quattro vicini con $C_{\max} = 11$ sono tutti tabu, e nessuno soddisfa il criterio di aspirazione ($C_{\max} < bestC = 11$): la migliore mossa non tabu **peggiora** il makespan a 12. È così che TS lascia il plateau.

- All'iterazione 6 il makespan scende a 10, e all'iterazione 8 a 9: $C_{\max} = \text{LB} = 9$, quindi la soluzione $J_1 = \{1, 3\}$, $J_2 = \{2, 4\}$, $J_3 = \{5, 6, 7\}$ è ottima e la ricerca si ferma: $TS(I) = 9 = z(I)$.



- La figura mostra i diagrammi di Gantt della soluzione di partenza, della soluzione dopo la mossa peggiorante dell'iterazione 5 (il lavoro spostato in arancione), e della soluzione ottima raggiunta all'iterazione 8.

- Con $tenure \leq 5$, TS non raggiunge l'ottimo su questa istanza entro 50 iterazioni: la lista tabu è troppo corta per impedire alla ricerca di tornare sul plateau di makespan 11.

4.5.2 Tabu search per il TSP, e l'effetto della durata tabu

- Per il TSP con l'intorno 2-opt, gli attributi naturali sono i **lati**: quando una mossa toglie i lati (σ_i, σ_{i+1}) e (σ_j, σ_{j+1}), questi due lati non possono essere rimessi nel giro per $tenure$ iterazioni, a meno che la mossa non dia un giro più corto del migliore. Tutto il resto è come nello pseudocodice della Sezione 4.5: a ogni iterazione tutte le $\frac{n(n-3)}{2}$ mosse sono valutate in $O(1)$ ciascuna, e si esegue la migliore non tabu.

- Le tabelle riportano il risultato di TS sulle istanze medie, dal giro nearest neighbour (326) per TSP20 con 200 iterazioni, e dalla soluzione LPT (73) per PCM15 con 300 iterazioni, per diversi valori della durata tabu; per PCM15 la tabella dà anche il numero di iterazioni eseguite (300 salvo che si raggiunga il lower bound).

TSP20: TS con 2-opt, 200 iterazioni ($z(I) = 291$)						
tenure	0	1	2	5	10	20
lunghezza migliore	319	291	291	291	291	291

PCM15: TS con l'intorno move, al più 300 iterazioni ($z(I) = 69$)										
tenure	0	1	2	3	5	7	10	15	20	30
$C_{\max}$ migliore	73	73	73	73	73	72	71	69	70	70
iterazioni	300	300	300	300	300	300	300	231	300	300

- Con $tenure = 0$ non c'è memoria, e TS cicla: su TSP20 non lascia mai il primo ottimo locale 2-opt. Su PCM15 l'intorno move ha un ampio plateau, e TS ha bisogno di una

lista tabu lunga per attraversarlo; ma una lista tabu troppo lunga vieta troppo, e con *tenure* ≥ 20 l'ottimo si perde di nuovo. La durata tabu va calibrata, e il suo valore migliore dipende dal problema e dall'istanza.

4.6 Iterated local search

L'*iterated local search* (indicata con ILS) applica una ricerca locale, e poi ripete: *perturba* l'ottimo locale corrente con una modifica casuale (un *kick*) che la ricerca locale non può annullare facilmente, applica la ricerca locale alla soluzione perturbata, e decide con un *criterio di accettazione* se il nuovo ottimo locale sostituisce quello corrente. È una passeggiata fra gli ottimi locali.

- Lo pseudocodice di ILS per il TSP, con la ricerca locale 2-opt e il kick double bridge, è il seguente:

Algoritmo

Algorithm: Iterated local search per il TSP

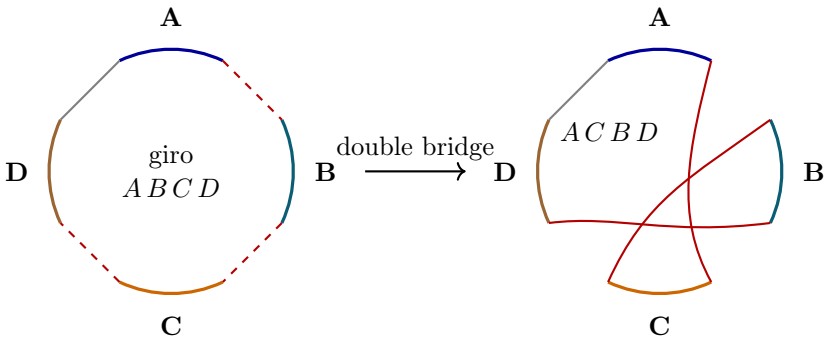
Input: distanze d, giro di partenza $\sigma[1..n]$, numero di iterazioni $maxIter$
Output: l'ultimo ottimo locale accettato

```
1  Function IteratedLocalSearch(d, σ, maxIter)
2      cur := TwoOptLS(σ); lcur := L(cur)              // primo ottimo locale
3      for it := 1 to maxIter do
4          nw := DoubleBridge(cur)                      // perturbazione
5          nw := TwoOptLS(nw); lnew := L(nw)            // ricerca locale
6          if lnew < lcur then
7              cur := nw; lcur := lnew                  // accettazione
8          end
9      end
10     return cur
11 Function DoubleBridge(σ)
12     p1 := 1 + ⌊Rand() · (n − 3)⌋
13     p2 := p1 + 1 + ⌊Rand() · (n − 2 − p1)⌋
14     p3 := p2 + 1 + ⌊Rand() · (n − 1 − p2)⌋
15     A := σ[1..p1]; B := σ[p1+1..p2]; C := σ[p2+1..p3]; D := σ[p3+1..n]
16     return (A, C, B, D)
```

- Lo schema è stato descritto nella sua forma generale da Lourenço, Martin e Stützle (2003), nello *Handbook of Metaheuristics*; per il TSP risale alle «large-step Markov chains» di Martin, Otto e Felten (1991), che hanno introdotto il kick double bridge.
- ILS ha tre componenti, e ciascuna ha un ruolo:
  - la **ricerca locale** intensifica: porta ogni soluzione perturbata al fondo della sua valle;
  - la **perturbazione** diversifica: sposta la ricerca in un'altra valle;

  - il **criterio di accettazione** bilancia le due: accettare solo ottimi locali migliori (come nello pseudocodice) intensifica, accettare ogni nuovo ottimo locale diversifica; regole intermedie accettano un ottimo locale peggiore con una certa probabilità, come in SA.

- L'**intensità** della perturbazione è il parametro chiave. Se è troppo debole, la ricerca locale la annulla e torna allo stesso ottimo locale; se è troppo forte, la soluzione perturbata non ha niente in comune con l'ottimo locale, e ILS diventa un multi-start casuale. Un buon kick cambia la soluzione in un modo che la ricerca locale *non può* annullare, ma ne conserva la maggior parte.



- Il **double bridge** taglia il giro in quattro segmenti A, B, C, D e li ricollega come $A\,C\,B\,D$: toglie tre lati (tratteggiati) e ne aggiunge tre nuovi (rossi), mentre il lato fra la fine di D e l'inizio di A si conserva. Nessun segmento viene invertito, mentre ogni mossa 2-opt inverte un segmento: quando B e C hanno almeno due città ciascuno, una singola mossa 2-opt non può annullare il kick, e la ricerca locale 2-opt, che accetta solo mosse miglioranti, torna raramente allo stesso ottimo locale. (Se B e C sono città singole, il kick cambia solo due lati ed è esso stesso una mossa 2-opt: l'esercizio «Iterated local search e VNS 2» usa questo caso.)

4.6.1 Esempio

Esempio 4.6: ILS sull'istanza media del TSP

Su TSP20, la ricerca locale 2-opt dal giro nearest neighbour (lunghezza 326) si ferma nell'ottimo locale

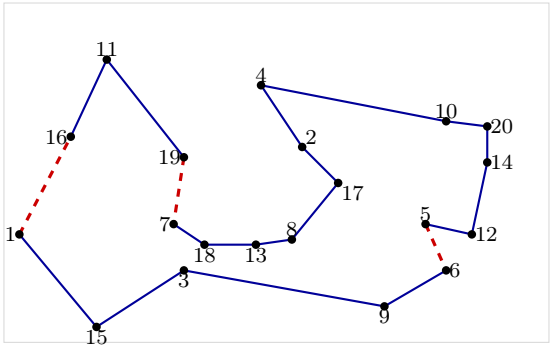
$$cur = (1, 16, 11, 19, 7, 18, 13, 8, 17, 2, 4, 10, 20, 14, 12, 5, 6, 9, 3, 15), \qquad L(cur) = 319.$$

Con $seed = 1$, i tre numeri casuali del primo kick sono 0.0000, 0.1315, 0.7556, quindi

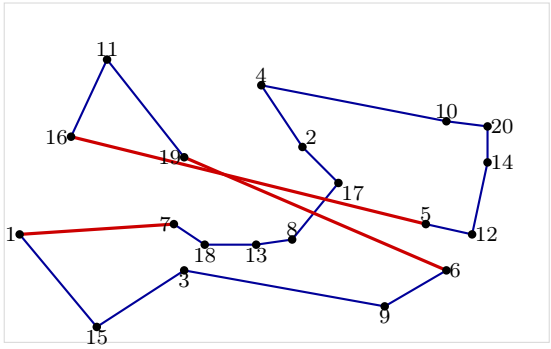
$$p_1 = 1 + \lfloor 0.0000 \cdot 17 \rfloor = 1, \quad p_2 = 1 + 1 + \lfloor 0.1315 \cdot 17 \rfloor = 4, \quad p_3 = 4 + 1 + \lfloor 0.7556 \cdot 15 \rfloor = 16,$$

e i segmenti sono $A = (1)$, $B = (16, 11, 19)$, $C = (7, 18, 13, 8, 17, 2, 4, 10, 20, 14, 12, 5)$, $D = (6, 9, 3, 15)$. Il kick toglie i lati $(1, 16)$, $(19, 7)$, $(5, 6)$ e aggiunge $(1, 7)$, $(5, 16)$, $(19, 6)$: il giro perturbato $A\,C\,B\,D$ ha lunghezza 432, molto peggiore. La ricerca locale 2-opt lo porta a un nuovo ottimo locale di lunghezza $303 < 319$, che viene accettato. Il secondo kick ($p = (8, 14, 16)$) dà, dopo 2-opt, un giro di lunghezza $291 = z(I)$; il terzo dà di nuovo un giro di lunghezza 291, non migliore, e viene rifiutato.

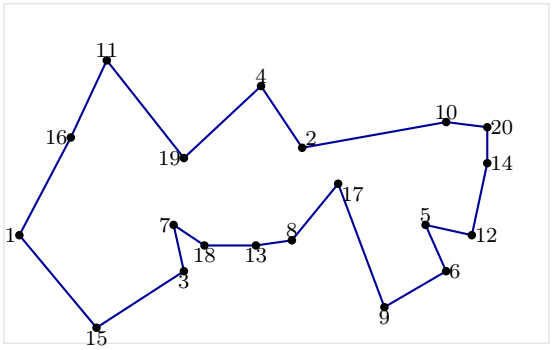
(a) ottimo locale: $L = 319$



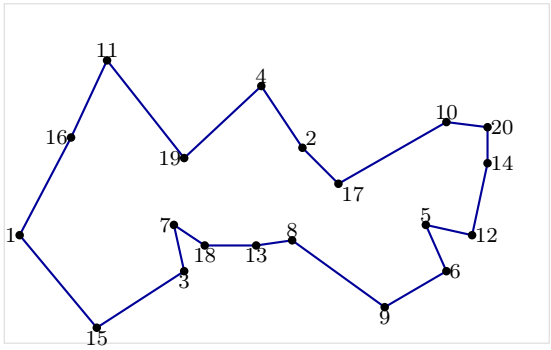
(b) dopo il double bridge: $L = 432$



(c) dopo 2-opt: $L = 303$



(d) dopo la seconda iterazione: $L = 291$



- Il riquadro (a) mostra l'ottimo locale con i tre lati tolti dal kick (tratteggiati), il riquadro (b) il giro perturbato con i tre lati nuovi, i riquadri (c) e (d) gli ottimi locali dopo la prima e la seconda iterazione. Il kick sposta il segmento C prima di B; la ricerca locale 2-opt ripara poi i lati lunghi che esso ha creato, conservando le parti buone del vecchio giro.
- Con 100 iterazioni e $seed = 1, \ldots, 10$, ILS raggiunge l'ottimo 291 in ogni esecuzione; l'iterazione in cui l'ottimo viene raggiunto è

$$2,\ 3,\ 1,\ 7,\ 8,\ 11,\ 15,\ 6,\ 2,\ 3$$

per i dieci semi. Ogni iterazione costa un kick e una ricerca locale 2-opt, che parte da una soluzione vicina a un ottimo locale e richiede poche mosse.

4.7 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}.

- Lo pseudocodice di VNS per $P\,||\,C_{\max}$, dove N_k contiene gli assegnamenti ottenuti con k mosse e la ricerca locale è MoveDescent, è il seguente:

Algoritmo

Algorithm: Variable neighbourhood search per $P\,||\,C_{\max}$

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      LB := max{max_j t[j], ⌈∑_j t[j]/m⌉}
3      mach := MoveDescent(mach); L := carichi di mach; C := max_h L[h]
4      it := 0; k := 1
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 Function Shake(mach, k)
17     for s := 1 to k do
18         j := 1 + ⌊Rand() · n⌋; g := 1 + ⌊Rand() · (m − 1)⌋
19         mach[j] := la g-esima macchina diversa da mach[j]
20     end
21     return mach
```

- 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.

- 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.

4.7.1 Esempio

Esempio 4.7: VNS sull'istanza di riferimento di $P\,||\,C_{\max}$

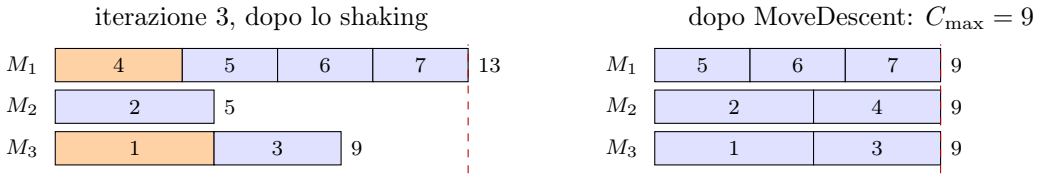
Partiamo dalla soluzione LPT, con carichi $(11, 8, 8)$, e usiamo $k_{\max} = 3$ e $seed = 1$ (numeri casuali 0.0000, 0.1315, 0.7556, 0.4587, 0.5328, 0.2190, 0.0470, 0.6789). MoveDescent non cambia la soluzione LPT, che è un ottimo locale per l'intorno move (Esempio 4.5). Ogni spostamento casuale usa due numeri: $j = 1 + \lfloor 7u \rfloor$ e $g = 1 + \lfloor 2u \rfloor$.

- **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: $k = 2$.

- **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 $C_{\max} = 9 = \text{LB}$ 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.



- 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

- 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 della Sezione 4.5.2 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.

4.8 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.

- Lo pseudocodice di $GRASP$ per il KP, con l'algoritmo greedy randomizzato per efficienza e KPLocalSearch, è il seguente:

Algoritmo

Algorithm: GRASP per il KP

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

Output: migliore soluzione trovata

Function *GRASP*$(p, w, c, \alpha, maxIter)$
  $zbest := -1$
  for $it := 1$ **to** $maxIter$ **do**
    $x :=$ RandomizedGreedy(p, w, c, α) `// costruzione`
    $x :=$ KPLocalSearch(x) `// ricerca locale`
    if $\sum_j p[j]x[j] > zbest$ **then** $best := x$; $zbest := \sum_j p[j]x[j]$
  end
  return *best*

Function *RandomizedGreedy*(p, w, c, α)
  $x := (0, \ldots, 0)$; $rc := c$ `// capacità residua`
  while *qualche oggetto con* $x[j] = 0$ *ha* $w[j] \le rc$ **do**
    $e_{\max}$, $e_{\min} :=$ massimo e minimo di $p[j]/w[j]$ su questi oggetti candidati
    $RCL :=$ candidati con $p[j]/w[j] \ge e_{\max} - \alpha\,(e_{\max} - e_{\min})$, per $p[j]/w[j]$ non crescente
    $j :=$ l'elemento di RCL in posizione $1 + \lfloor$Rand$() \cdot |RCL| \rfloor$
    $x[j] := 1$; $rc := rc - w[j]$
  end
  return x

- 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):

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

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

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

- 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 4.8: *GRASP* sull'istanza di riferimento del KP

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.

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.

$GRASP(I) = 17 = z(I)$. Su questa istanza piccola ogni partenza porta all'ottimo; su istanze più grandi partenze diverse portano a ottimi locali diversi.

4.8.1 GRASP per il TSP, e l'effetto di α

- 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 $d_{\min} + \alpha(d_{\max} - d_{\min})$, e la ricerca locale è TwoOptLS.

- La tabella 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.

	TSP20 ($z(I) = 291$)				KP20 ($z(I) = 521$)			
α	min	media	max	ottime	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.

4.9 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*.

- Lo pseudocodice di LNS per il TSP, che toglie q città casuali e le reinserisce con l'inserimento più economico (cheapest insertion), è il seguente:

Algoritmo

Algorithm: Large neighbourhood search per il TSP

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

- 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 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.
- 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 4.9: *LNS* sull'istanza di riferimento del TSP

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 $(1, 2, 4, 5)$, e 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 $2 + 7 + 9 = 18$. Cheapest insertion della città 3:

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

quindi la città 3 va fra 5 e 1: $(1, 4, 5, 3)$. Cheapest insertion della città 2:

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

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.

4.9.1 Large neighbourhood search per il BPP: riparazione euristica ed esatta

- 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.
- 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.

Algoritmo

Algorithm: Large neighbourhood search per il BPP

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

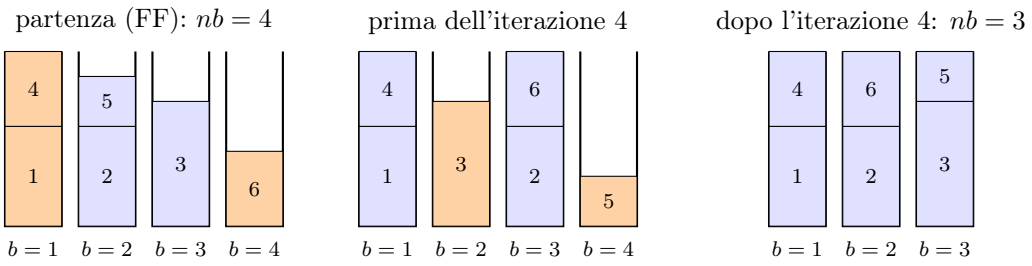
Function *LNS-BPP*$(w, c, bin, nb, q, exact, maxIter)$
  LB $:= \lceil \sum_j w[j]/c \rceil$; $it := 0$
  while $it < maxIter$ ***and*** $nb >$ LB **do**
    $it := it + 1$
    $empty :=$ il contenitore meno carico e $q-1$ altri contenitori a caso `// distruzione`
    $items :=$ gli oggetti posti nei contenitori di $empty$
    $newbin :=$ gli altri contenitori, rinumerati $1, \ldots, nb - q$
    if $exact$ **then** impacchettare $items$ in modo ottimo in k contenitori nuovi **else** impacchettare $items$ con FFD in k contenitori nuovi `// riparazione`
    $nb' := nb - q + k$
    if $nb' < nb$ ***or*** $(nb' = nb$ ***and*** $\sum_b (carico\ di\ b\ in\ newbin)^2 \geq \sum_b (carico\ di\ b\ in\ bin)^2)$ **then**
      $bin := newbin$; $nb := nb'$ `// accettazione`
    end
  end
  return bin, nb

Esempio 4.10: *LNS* **sull'istanza di riferimento del BPP**

Partiamo dall'impacchettamento calcolato da First Fit, $P_1 = \{1, 4\}$, $P_2 = \{2, 5\}$, $P_3 = \{3\}$, $P_4 = \{6\}$, con carichi $(7, 6, 5, 3)$, e 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.

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 $nb =$ LB.



- 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	$9, 9, 9, 9, 9, 9, 9, 9, 9, 9$	$2, 3, 9, 12, 4, 5, 1, 3, 10, 2$

- 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.

4.9.2 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:

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

dove $\rho \in [0, 1]$ è il *fattore di reazione* e 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.

Algoritmo

Algorithm: Adaptive large neighbourhood search per il TSP

Input: distanze d, giro di partenza σ, q, $maxIter$, fattore di reazione ρ
Output: migliore giro trovato

Function $ALNS(d, \sigma, q, maxIter, \rho)$
  $cur := \sigma$; $best := \sigma$; $w_1 := 1$; $w_2 := 1$
  for $it := 1$ **to** $maxIter$ **do**
    if $\textsc{Rand}() \cdot (w_1 + w_2) < w_1$ **then** $o := 1$ **else** $o := 2$ `// roulette wheel`
    $nw := cur$
    if $o = 1$ **then** $removed := \textsc{RandomRemoval}(nw, q)$ **else** $removed := \textsc{WorstRemoval}(nw, q)$
    $\textsc{CheapestInsertionRepair}(nw, removed)$
    $\psi := 3$, 2, 1 o 0 se $L(nw) < L(best)$, $L(nw) < L(cur)$, $L(nw) = L(cur)$, $L(nw) > L(cur)$
    if $L(nw) \leq L(cur)$ **then** $cur := nw$
    if $L(cur) < L(best)$ **then** $best := cur$
    $w_o := (1 - \rho)\, w_o + \rho\, \psi$ `// pesi adattivi`
  end
  return $best$

- 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.

4.10 Metodi basati su popolazioni: 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.

- Lo pseudocodice di un GA steady-state per il KP con codifica binaria è il seguente; con $memetic = \text{true}$ è un algoritmo memetico.

Algoritmo

Algorithm: Algoritmo genetico e memetico per il KP

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
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
13         REPAIRKP(child); if memetic then KPLOCALSEARCH(child)
14         worst := l'ultimo indice s di minimo fit[s]
15         if child ∉ pop and Σ_j p[j] child[j] > fit[worst] then   // sostituzione
16             pop[worst] := child; fit[worst] := Σ_j p[j] child[j]
17         end
18     end
19     return l'individuo di massimo fit
20 Function Tournament()
21     a := 1 + ⌊RAND() · P⌋; b := 1 + ⌊RAND() · P⌋
22     if fit[a] ≥ fit[b] then return a else return b
```

- 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.** 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 nella Sezione 4.4.3, 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.
- 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 4.11: *GA* sull'istanza di riferimento del KP

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):

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

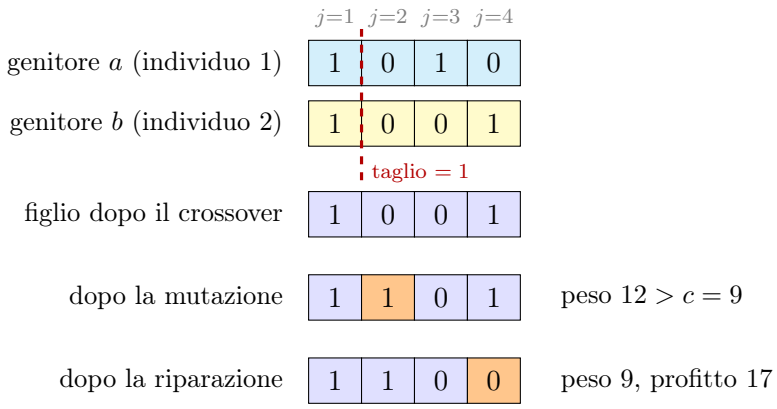
Tutti e quattro gli individui sono ammissibili: $pop[1] = (1,0,1,0)$, $pop[2] = (1,0,0,1)$, $pop[3] = (1,0,0,0)$, $pop[4] = (0,1,0,1)$, con fitness $(16, 14, 10, 11)$.

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.

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.



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

	min	media	max	ottime	profitto migliore per seme
GA	508	516.9	521	4	521, 519, 521, 508, 514, 521, 510, 519, 521, 515
memetico	521	521.0	521	10	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.

4.11 Confronto e conclusioni

4.11.1 I metodi sulle istanze medie

- La tabella raccoglie i risultati degli esperimenti delle sezioni precedenti: 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
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
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

- 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».

4.11.2 Intensificazione e diversificazione

- 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
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

4.11.3 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.

4.11.4 Che cosa questi metodi non usano

- 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.
- 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 (Sezione 4.9.1), 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.

4.12 Errori tipici

Attenzione

- **Riportare una sola esecuzione.** I risultati dipendono dal seme: riportare minimo, media e massimo su più esecuzioni.
- **Calibrare e valutare sulle stesse istanze.** Il quadro è ottimistico; usare istanze di addestramento per i parametri e istanze diverse per la valutazione.
- **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.
- **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.

- **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.

4.13 In sintesi

In sintesi

- 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.

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

4.14 Implementazione in C++

- Tutte le funzioni di questa sezione 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 della Sezione 4.4.4.
- I numeri casuali sono prodotti da `Rand`, il generatore della Sezione 4.2.2. 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 // Park-Miller minimal standard generator: seed = 16807 * seed mod (2^31 - 1),
      u = seed /
2 // (2^31 - 1)
```

```
// if a list of numbers is given, they are returned in order instead

double Rand(Random& r)
{
    double u;
    if (r.given.size() > 0)
    {
        u = r.given[r.used];
    }
    else
    {
        r.seed = (16807 * r.seed) % 2147483647;
        u = (double) r.seed / 2147483647.0;
    }
    r.used = r.used + 1;
    return u;
}
```

4.14.1 Ricerche locali e funzioni ausiliarie

- La ricerca locale 2-opt per il TSP, con la valutazione in $O(1)$ di una mossa:

```
// length L(pi) of the tour pi

int TourLength(const Vec& pi, const Matrix& d)
{
    int n = pi.size();
    int len = 0;
    for (int k = 0; k < n; ++k)
    {
        len = len + d[pi[k]][pi[(k + 1) % n]];
    }
    return len;
}
```

```
// change of length of the 2-opt move (i,j): the edges (pi[i],pi[i+1]) and (pi[j],pi[j+1])
// are replaced by (pi[i],pi[j]) and (pi[i+1],pi[j+1]); computed in O(1)

int TwoOptDelta(const Vec& pi, const Matrix& d, int i, int j)
{
    int n = pi.size();
    int a = pi[i], b = pi[i + 1], c = pi[j], e = pi[(j + 1) % n];
    return d[a][c] + d[b][e] - d[a][b] - d[c][e];
}
```

```
// performs the 2-opt move (i,j): reverses the segment pi[i+1..j]

void ApplyTwoOpt(Vec& pi, int i, int j)
{
    int lo = i + 1, hi = j;
    while (lo < hi)
    {
        int tmp = pi[lo]; pi[lo] = pi[hi]; pi[hi] = tmp;
        lo = lo + 1; hi = hi - 1;
    }
}
```

```cpp
// 2-opt local search with best improvement: stops at a local optimum
// ties between moves are broken in favour of the first move (i,j) in lexicographic order

void TwoOptLS(Vec& pi, const Matrix& d)
{
    int n = pi.size();
    while (true)
    {
        int bdelta = 0, bi = -1, bj = -1;
        for (int i = 0; i < n - 1; ++i)
        {
            for (int j = i + 2; j < n; ++j)
            {
                if (i == 0 && j == n - 1) continue;      // same two edges: no change
                int delta = TwoOptDelta(pi, d, i, j);
                if (delta < bdelta) { bdelta = delta; bi = i; bj = j; }
            }
        }
        // no improving move: local optimum
        if (bi < 0) return;
        ApplyTwoOpt(pi, bi, bj);
    }
}
```

```cpp
// nearest neighbour tour from city 0 (ties: smallest index)

Vec NearestNeighbour(const Matrix& d)
{
    int n = d.size();
    Vec pi(1, 0);
    vector<bool> visited(n, false);
    visited[0] = true;
    for (int k = 1; k < n; ++k)
    {
        int last = pi.back(), next = -1;
        for (int j = 0; j < n; ++j)
        {
            if (!visited[j] && (next < 0 || d[last][j] < d[last][next])) next = j;
        }
        pi.push_back(next);
        visited[next] = true;
    }
    return pi;
}
```

- La discesa move per $P\,||\,C_{\max}$ con il confronto lessicografico, e la soluzione di partenza LPT:

```cpp
// loads L[h] of the machines for the assignment mach

Vec Loads(const Vec& mach, const Vec& t, int m)
{
    Vec L(m, 0);
```

```
    for (int j = 0; j < (int) t.size(); ++j) L[mach[j]] = L[mach[j]] + t[j];
    return L;
}
```

```
// true if the loads L1 are better than L2 in the lexicographic order
// (makespan first, then sum of the squared loads)

bool LexLess(const Vec& L1, const Vec& L2)
{
    int C1 = 0, C2 = 0, S1 = 0, S2 = 0;
    for (int h = 0; h < (int) L1.size(); ++h)
    {
        if (L1[h] > C1) C1 = L1[h];
        if (L2[h] > C2) C2 = L2[h];
        S1 = S1 + L1[h] * L1[h];
        S2 = S2 + L2[h] * L2[h];
    }
    return C1 < C2 || (C1 == C2 && S1 < S2);
}
```

```
// local search for P||Cmax with the move neighbourhood and best improvement,
// with respect to the lexicographic order (makespan, sum of squared loads)

void MoveDescent(const Vec& t, int m, Vec& mach)
{
    int n = t.size();
    Vec L = Loads(mach, t, m);
    while (true)
    {
        int bj = -1, bh = -1;
        Vec bL = L;
        for (int j = 0; j < n; ++j)
        {
            for (int h = 0; h < m; ++h)
            {
                if (h == mach[j]) continue;
                Vec L2 = L;
                L2[mach[j]] = L2[mach[j]] - t[j];
                L2[h] = L2[h] + t[j];
                if (LexLess(L2, bL)) { bj = j; bh = h; bL = L2; }
            }
        }
        if (bj < 0) return;                                  // local optimum
        mach[bj] = bh;
        L = bL;
    }
}
```

```
// longest processing time first: jobs by non-increasing t (ties: smallest
    index),
// each assigned to the least loaded machine (ties: smallest index)

Vec LPT(const Vec& t, int m)
{
    int n = t.size();
    Vec order(n), mach(n), L(m, 0);
    for (int j = 0; j < n; ++j) order[j] = j;
```

```
    for (int a = 1; a < n; ++a)                              // insertion sort, stable
    {
        int j = order[a], b = a;
        while (b > 0 && t[order[b - 1]] < t[j]) { order[b] = order[b - 1]; b = b - 1; }
        order[b] = j;
    }
    for (int a = 0; a < n; ++a)
    {
        int j = order[a], h = 0;
        for (int k = 1; k < m; ++k) if (L[k] < L[h]) h = k;
        mach[j] = h;
        L[h] = L[h] + t[j];
    }
    return mach;
}
```

- La ricerca locale per il KP con flip e swap ammissibili:

```
// total profit of the items selected by x

int Profit(const Vec& p, const Vec& x)
{
    int P = 0;
    for (int i = 0; i < (int) p.size(); ++i) if (x[i] == 1) P = P + p[i];
    return P;
}
```

```
// total weight of the items selected by x

int Weight(const Vec& w, const Vec& x)
{
    int W = 0;
    for (int i = 0; i < (int) w.size(); ++i) if (x[i] == 1) W = W + w[i];
    return W;
}
```

```
// local search for the KP with best improvement on the feasible neighbours of the flip
// neighbourhood (only additions can improve) and of the swap neighbourhood

void KPLocalSearch(const Vec& p, const Vec& w, int c, Vec& x)
{
    int n = p.size();
    while (true)
    {
        int W = Weight(w, x), P = Profit(p, x);
        // bi = -1: add item bk; else swap bi out, bk in
        int bP = P, bi = -1, bk = -1;
        for (int k = 0; k < n; ++k)
            if (x[k] == 0 && W + w[k] <= c && P + p[k] > bP)
            { bP = P + p[k]; bi = -1; bk = k; }
        for (int i = 0; i < n; ++i)
        {
            if (x[i] != 1) continue;
```

```
            for (int k = 0; k < n; ++k)
                if (x[k] == 0 && W - w[i] + w[k] <= c && P - p[i] + p[k] > bP)
                { bP = P - p[i] + p[k]; bi = i; bk = k; }
        }
        if (bk < 0) return;                                  // local optimum
        if (bi >= 0) x[bi] = 0;
        x[bk] = 1;
    }
}
```

4.14.2 Simulated annealing

- Il simulated annealing per il TSP (Sezione 4.4, con l'intorno 2-opt), e per il KP con l'obiettivo penalizzato della Sezione 4.4.3:

```
// simulated annealing for the TSP with the 2-opt neighbourhood and geometric
    cooling
// pi: starting tour on input, best tour found on output; returns its length
// if curve is not NULL, the length of the current tour after each move is
    appended to it

int SimulatedAnnealingTSP(const Matrix& d, Vec& pi, double T0, double alpha,
    int L,
                          double Tmin, Random& r, Vec* curve)
{
    int n = pi.size();
    // the M = n(n-3)/2 moves (i,j) of 2-opt
    Vec mi, mj;
    for (int i = 0; i < n - 1; ++i)
        for (int j = i + 2; j < n; ++j)
            if (!(i == 0 && j == n - 1)) { mi.push_back(i); mj.push_back(j); }
    int M = mi.size();

    Vec cur = pi;                                            // current solution
    int fcur = TourLength(cur, d);
    Vec best = cur;                                          // best solution found
    int fbest = fcur;
    double T = T0;                                           // temperature
    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; }
            }
            if (curve != NULL) curve->push_back(fcur);
        }
        T = alpha * T;                                       // cooling
```

```
    }
    pi = best;
    return fbest;
}
```

```
// simulated annealing for the KP with the flip neighbourhood and the penalized objective
// f(x) = profit(x) - beta * max{0, weight(x) - c}; infeasible solutions may be visited,
// the best solution is updated with feasible solutions only
// x: feasible starting solution on input, best solution found on output; returns its
// profit

int SimulatedAnnealingKP(const Vec& p, const Vec& w, int c, Vec& x, int beta,
                         double T0, double alpha, int L, double Tmin, Random& r)
{
    int n = p.size();
    int excess = Weight(w, x) - c;
    int f = Profit(p, x) - beta * (excess > 0 ? excess : 0);
    Vec best = x;
    int zbest = Profit(p, x);
    double T = T0;
    while (T > Tmin)
    {
        for (int l = 0; l < L; ++l)
        {
            int k = (int) (Rand(r) * n);                    // random flip
            Vec y = x;
            y[k] = 1 - y[k];
            int ey = Weight(w, y) - c;
            int fy = Profit(p, y) - beta * (ey > 0 ? ey : 0);
            // maximization: delta > 0 is a worsening
            int delta = f - fy;
            double v = Rand(r);
            if (delta <= 0 || v < exp(-delta / T))
            {
                x = y; f = fy;
                if (ey <= 0 && Profit(p, x) > zbest) { best = x; zbest = Profit(p, x); }
            }
        }
        T = alpha * T;
    }
    x = best;
    return zbest;
}
```

4.14.3 Tabu search

- La tabu search per $P\,||\,C_{\max}$ (Sezione 4.5) e per il TSP (Sezione 4.5.2):

```
// tabu search for P||Cmax with the move neighbourhood (one job to another
    machine);
// the best non-tabu move is performed even if it is worse (makespan first,
    then sum of
// squares);
// after moving job j away from machine h, the attribute (j,h) is tabu for
    tenure
// iterations;
// a tabu move is allowed if it gives a makespan smaller than the best one (
    aspiration)
// mach: starting assignment on input, best assignment on output; returns its
    makespan

int TabuSearchPCmax(const Vec& t, int m, Vec& mach, int tenure, int maxIter)
{
    int n = t.size();
    Vec L = Loads(mach, t, m);
    // tabu[j][h] >= it: j may not go to h
    Matrix tabu(n, Vec(m, 0));
    Vec best = mach;
    int bestC = 0, sum = 0, LB = 0;
    for (int h = 0; h < m; ++h) if (L[h] > bestC) bestC = L[h];
    for (int j = 0; j < n; ++j) { sum = sum + t[j]; if (t[j] > LB) LB = t[j]; }
    // LB = max{ max t_j, ceil(sum/m) }
    if ((sum + m - 1) / m > LB) LB = (sum + m - 1) / m;
    int it = 0;
    while (it < maxIter && bestC > LB)
    {
        it = it + 1;
        int bj = -1, bh = -1;
        Vec bL;
        for (int j = 0; j < n; ++j)
        {
            for (int h = 0; h < m; ++h)
            {
                if (h == mach[j]) continue;
                Vec L2 = L;
                L2[mach[j]] = L2[mach[j]] - t[j];
                L2[h] = L2[h] + t[j];
                int C2 = 0;
                for (int k = 0; k < m; ++k) if (L2[k] > C2) C2 = L2[k];
                // tabu and no aspiration
                if (tabu[j][h] >= it && !(C2 < bestC)) continue;
                if (bj < 0 || LexLess(L2, bL)) { bj = j; bh = h; bL = L2; }
            }
        }
        if (bj < 0) break;
        // j may not go back for tenure iterations
        tabu[bj][mach[bj]] = it + tenure;
        mach[bj] = bh;
        L = bL;
        int C = 0;
        for (int k = 0; k < m; ++k) if (L[k] > C) C = L[k];
        if (C < bestC) { bestC = C; best = mach; }
```

```
    }
    mach = best;
    return bestC;
}
```

```
// tabu search for the TSP with the 2-opt neighbourhood
// the two edges removed by a move may not be added back for tenure iterations,
// unless the move gives a tour shorter than the best one (aspiration criterion)

int TabuSearchTSP(const Matrix& d, Vec& pi, int tenure, int maxIter)
{
    int n = pi.size();
    Vec cur = pi;
    int lcur = TourLength(cur, d);
    Vec best = cur;
    int lbest = lcur;
    // tabu[a][b] >= it: edge (a,b) is tabu
    Matrix tabu(n, Vec(n, 0));
    for (int it = 1; it <= maxIter; ++it)
    {
        int bdelta = 0, bi = -1, bj = -1;
        for (int i = 0; i < n - 1; ++i)
        {
            for (int j = i + 2; j < n; ++j)
            {
                if (i == 0 && j == n - 1) continue;
                int a = cur[i], b = cur[i + 1], c = cur[j], e = cur[(j + 1) % n];
                int delta = d[a][c] + d[b][e] - d[a][b] - d[c][e];
                bool isTabu = tabu[a][c] >= it || tabu[b][e] >= it;
                if (isTabu && !(lcur + delta < lbest)) continue;
                if (bi < 0 || delta < bdelta) { bdelta = delta; bi = i; bj = j; }
            }
        }
        if (bi < 0) break;                                  // every move is tabu
        int a = cur[bi], b = cur[bi + 1], c = cur[bj], e = cur[(bj + 1) % n];
        tabu[a][b] = tabu[b][a] = it + tenure;              // removed edges become tabu
        tabu[c][e] = tabu[e][c] = it + tenure;
        // best non-tabu move, even if worse
        ApplyTwoOpt(cur, bi, bj);
        lcur = lcur + bdelta;
        if (lcur < lbest) { best = cur; lbest = lcur; }
    }
    pi = best;
    return lbest;
}
```

4.14.4 Iterated local search e variable neighbourhood search

- L'iterated local search per il TSP con il kick double bridge (Sezione 4.6):

```
// double-bridge kick: the tour A B C D, cut at the positions p1 < p2 < p3, becomes A C B
// D

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

```
// iterated local search for the TSP: double-bridge kick + 2-opt local search,
// the new local optimum is accepted if it is better than the current one

int IteratedLocalSearchTSP(const Matrix& d, Vec& pi, int maxIter, Random& r)
{
    Vec cur = pi;
    TwoOptLS(cur, d);                                        // first local optimum
    int lcur = TourLength(cur, d);
    for (int it = 1; it <= maxIter; ++it)
    {
        Vec nw = DoubleBridge(cur, r);                       // perturbation
        TwoOptLS(nw, d);                                     // local search
        int lnew = TourLength(nw, d);
        if (lnew < lcur) { cur = nw; lcur = lnew; }          // acceptance criterion
    }
    pi = cur;
    return lcur;
}
```

- La variable neighbourhood search per $P\,||\,C_{\max}$ (Sezione 4.7):

```
// shaking in the k-th neighbourhood: k random moves, each one moving a random job j
// to a random machine different from mach[j]

Vec Shake(const Vec& t, int m, const Vec& mach, int k, Random& r)
{
    int n = t.size();
    Vec sh = mach;
    for (int s = 0; s < k; ++s)
    {
        int j = (int) (Rand(r) * n);
        int g = (int) (Rand(r) * (m - 1));
```

```
        // the (g+1)-th machine other than sh[j]
        sh[j] = (g < sh[j]) ? g : g + 1;
    }
    return sh;
}
```

```
// basic variable neighbourhood search for P||Cmax: shaking with k random moves,
// local search by MoveDescent; k returns to 1 after an improvement, otherwise it grows
// up to kmax

int VNSPCmax(const Vec& t, int m, Vec& mach, int kmax, int maxIter, Random& r)
{
    int n = t.size(), sum = 0, LB = 0;
    for (int j = 0; j < n; ++j) { sum = sum + t[j]; if (t[j] > LB) LB = t[j]; }
    if ((sum + m - 1) / m > LB) LB = (sum + m - 1) / m;
    MoveDescent(t, m, mach);
    Vec L = Loads(mach, t, m);
    int C = 0;
    for (int h = 0; h < m; ++h) if (L[h] > C) C = L[h];
    int it = 0, k = 1;
    while (it < maxIter && C > LB)
    {
        it = it + 1;
        Vec nw = Shake(t, m, mach, k, r);              // shaking
        MoveDescent(t, m, nw);                         // local search
        Vec Lnew = Loads(nw, t, m);
        if (LexLess(Lnew, L))                          // move or not
        {
            mach = nw; L = Lnew; k = 1;
            C = 0;
            for (int h = 0; h < m; ++h) if (L[h] > C) C = L[h];
        }
        else
        {
            k = (k < kmax) ? k + 1 : 1;
        }
    }
    return C;
}
```

4.14.5 GRASP

- GRASP per il KP (Sezione 4.8) e per il TSP (Sezione 4.8.1):

```
// greedy randomized construction for the KP: at each step the candidates are the items
// that fit,
// RCL = { i candidate : p_i/w_i >= emax - alpha (emax - emin) }, sorted by non-increasing
// efficiency (ties: smallest index), and the item is chosen at random in the RCL

Vec RandomizedGreedyKP(const Vec& p, const Vec& w, int c, double alpha, Random& r)
{
    int n = p.size(), rc = c;                              // rc: residual capacity
    Vec x(n, 0);
    while (true)
    {
        double emax = -1, emin = -1;
        for (int i = 0; i < n; ++i)
        {
            if (x[i] == 1 || w[i] > rc) continue;
            double e = (double) p[i] / w[i];
            if (emax < 0 || e > emax) emax = e;
            if (emin < 0 || e < emin) emin = e;
        }
        if (emax < 0) return x;                            // no item fits
        double thr = emax - alpha * (emax - emin);
        Vec rcl;
        for (int i = 0; i < n; ++i)
        {
            if (x[i] == 1 || w[i] > rc || (double) p[i] / w[i] < thr - 1e-9) continue;
            int pos = rcl.size();                          // insertion in sorted position
            rcl.push_back(i);
            while (pos > 0 && p[rcl[pos - 1]] * w[i] < p[i] * w[rcl[pos - 1]])
            { rcl[pos] = rcl[pos - 1]; pos = pos - 1; }
            rcl[pos] = i;
        }
        int i = rcl[(int) (Rand(r) * rcl.size())];
        x[i] = 1;
        rc = rc - w[i];
    }
}
```

```
// GRASP for the KP: greedy randomized construction + local search, maxIter times

int GraspKP(const Vec& p, const Vec& w, int c, Vec& best, double alpha, int maxIter,
            Random& r)
{
    int zbest = -1;
    for (int it = 1; it <= maxIter; ++it)
```

```
    {
        Vec x = RandomizedGreedyKP(p, w, c, alpha, r);   // construction
        KPLocalSearch(p, w, c, x);                       // local search
        int z = Profit(p, x);
        if (z > zbest) { best = x; zbest = z; }
    }
    return zbest;
}
```

```
// randomized nearest neighbour from city 0: the next city is chosen at random in the
// restricted candidate list RCL = { j unvisited : d[last][j] <= dmin + alpha (dmax -
// dmin) },
// sorted by non-decreasing distance (ties: smallest index)

Vec RandomizedNN(const Matrix& d, double alpha, Random& r)
{
    int n = d.size();
    Vec pi(1, 0);
    vector<bool> visited(n, false);
    visited[0] = true;
    for (int k = 1; k < n; ++k)
    {
        int last = pi.back();
        int dmin = -1, dmax = -1;
        for (int j = 0; j < n; ++j)
        {
            if (visited[j]) continue;
            if (dmin < 0 || d[last][j] < dmin) dmin = d[last][j];
            if (dmax < 0 || d[last][j] > dmax) dmax = d[last][j];
        }
        double thr = dmin + alpha * (dmax - dmin);
        Vec rcl;
        for (int j = 0; j < n; ++j)
        {
            if (visited[j] || d[last][j] > thr + 1e-9) continue;
            int pos = rcl.size();                        // insertion in sorted position
            rcl.push_back(j);
            while (pos > 0 && d[last][rcl[pos - 1]] > d[last][j])
            { rcl[pos] = rcl[pos - 1]; pos = pos - 1; }
            rcl[pos] = j;
        }
        int next = rcl[(int) (Rand(r) * rcl.size())];
        pi.push_back(next);
        visited[next] = true;
    }
    return pi;
}
```

```
// GRASP for the TSP: randomized nearest neighbour + 2-opt local search, maxIter times

int GraspTSP(const Matrix& d, Vec& best, double alpha, int maxIter, Random& r)
{
    int lbest = -1;
    for (int it = 1; it <= maxIter; ++it)
```

```
    {
        Vec pi = RandomizedNN(d, alpha, r);              // greedy randomized
    construction
        TwoOptLS(pi, d);                                 // local search
        int len = TourLength(pi, d);
        if (lbest < 0 || len < lbest) { best = pi; lbest = len; }
    }
    return lbest;
}
```

4.14.6 Large neighbourhood search

- LNS e ALNS per il TSP (Sezioni 4.9 e 4.9.2):

```
// destroy: removes q random cities from the tour pi (never the first city) and
    returns
// them

Vec RandomRemoval(Vec& pi, int q, Random& r)
{
    Vec removed;
    for (int s = 0; s < q; ++s)
    {
        int k = 1 + (int) (Rand(r) * (pi.size() - 1));
        removed.push_back(pi[k]);
        pi.erase(pi.begin() + k);
    }
    return removed;
}
```

```
// destroy: removes, q times, the city whose removal saves the most length (
    never the
// first city)

Vec WorstRemoval(Vec& pi, int q, const Matrix& d)
{
    Vec removed;
    for (int s = 0; s < q; ++s)
    {
        int n = pi.size(), bk = -1, bs = 0;
        for (int k = 1; k < n; ++k)
        {
            int a = pi[k - 1], v = pi[k], b = pi[(k + 1) % n];
            int saving = d[a][v] + d[v][b] - d[a][b];
            if (bk < 0 || saving > bs) { bk = k; bs = saving; }
        }
        removed.push_back(pi[bk]);
        pi.erase(pi.begin() + bk);
    }
    return removed;
}
```

```
// repair: inserts the removed cities, in the order of removal, each in the
    position
// of minimum insertion cost d[a][v] + d[v][b] - d[a][b] (ties: first position)

void CheapestInsertionRepair(Vec& pi, const Vec& removed, const Matrix& d)
```

```
{
    for (int s = 0; s < (int) removed.size(); ++s)
    {
        int v = removed[s], n = pi.size(), bpos = -1, bcost = 0;
        for (int k = 0; k < n; ++k)
        {
            int a = pi[k], b = pi[(k + 1) % n];
            int cost = d[a][v] + d[v][b] - d[a][b];
            if (bpos < 0 || cost < bcost) { bpos = k; bcost = cost; }
        }
        pi.insert(pi.begin() + bpos + 1, v);
    }
}
```

```
// large neighbourhood search for the TSP: random removal of q cities + cheapest
// insertion,
// the new tour is accepted if it is shorter than the current one

int LnsTSP(const Matrix& d, Vec& pi, int q, int maxIter, Random& r)
{
    Vec cur = pi;
    int lcur = TourLength(cur, d);
    for (int it = 1; it <= maxIter; ++it)
    {
        Vec nw = cur;
        Vec removed = RandomRemoval(nw, q, r);          // destroy
        CheapestInsertionRepair(nw, removed, d);        // repair
        int lnew = TourLength(nw, d);
        if (lnew < lcur) { cur = nw; lcur = lnew; }     // acceptance
    }
    pi = cur;
    return lcur;
}
```

```
// adaptive large neighbourhood search for the TSP with two destroy operators
// (o = 0: random removal, o = 1: worst removal) and cheapest insertion as repair;
// the operator is chosen with probability proportional to its weight wgt[o],
// updated as wgt[o] = (1 - rho) wgt[o] + rho psi, with psi = 3, 2, 1, 0 if the new tour
// is a new best, better than the current one, as good as the current one, worse

int AlnsTSP(const Matrix& d, Vec& pi, int q, int maxIter, double rho, Random& r,
            double* wgt)
{
    Vec cur = pi;
    int lcur = TourLength(cur, d);
    Vec best = cur;
    int lbest = lcur;
    wgt[0] = 1.0; wgt[1] = 1.0;
    for (int it = 1; it <= maxIter; ++it)
    {
        int o = (Rand(r) * (wgt[0] + wgt[1]) < wgt[0]) ? 0 : 1;     // roulette wheel
        Vec nw = cur;
```

```
        Vec removed = (o == 0) ? RandomRemoval(nw, q, r) : WorstRemoval(nw, q, d);
        CheapestInsertionRepair(nw, removed, d);
        int lnew = TourLength(nw, d);
        double psi;
        if (lnew < lbest) psi = 3.0;
        else if (lnew < lcur) psi = 2.0;
        else if (lnew == lcur) psi = 1.0;
        else psi = 0.0;
        if (lnew <= lcur) { cur = nw; lcur = lnew; }
        if (lcur < lbest) { best = cur; lbest = lcur; }
        wgt[o] = (1 - rho) * wgt[o] + rho * psi;                    // adaptive weights
    }
    pi = best;
    return lbest;
}
```

- LNS per il BPP (Sezione 4.9.1), con l'impacchettamento di partenza First Fit, il metodo di distruzione e i due metodi di riparazione:

```
// first fit on the items in the given order; returns the number of bins nb, bin[i] = bin
// of item i

int FirstFit(const Vec& w, int c, const Vec& order, Vec& bin)
{
    Vec load;
    bin.assign(w.size(), -1);
    for (int a = 0; a < (int) order.size(); ++a)
    {
        int i = order[a], b = 0;
        while (b < (int) load.size() && load[b] + w[i] > c) b = b + 1;
        if (b == (int) load.size()) load.push_back(0);   // open a new bin
        bin[i] = b;
        load[b] = load[b] + w[i];
    }
    return load.size();
}
```

```
// destroy for the BPP: the least loaded bin (ties: smallest index) and q-1 other bins at
// random

Vec DestroyBins(const Vec& bin, int nb, const Vec& w, int q, Random& r)
{
    Vec load(nb, 0);
    for (int i = 0; i < (int) w.size(); ++i) load[bin[i]] = load[bin[i]] + w[i];
    int e = 0;
    for (int b = 1; b < nb; ++b) if (load[b] < load[e]) e = b;
    Vec empty(1, e);
    while ((int) empty.size() < q && (int) empty.size() < nb)
    {
        Vec others;
        for (int b = 0; b < nb; ++b)
```

```
        {
            bool chosen = false;
            for (int s = 0; s < (int) empty.size(); ++s) if (empty[s] == b)
    chosen = true;
            if (!chosen) others.push_back(b);
        }
        empty.push_back(others[(int) (Rand(r) * others.size())]);
    }
    return empty;
}
```

```
// heuristic repair for the BPP: first fit decreasing on the removed items, in
    new bins;
// asg[s] = bin (0, 1, ...) of items[s]; returns the number of new bins

int RepairFFD(const Vec& items, const Vec& w, int c, Vec& asg)
{
    int k = items.size();
    Vec order(k);
    for (int s = 0; s < k; ++s) order[s] = s;
    for (int a = 1; a < k; ++a)                          // non-increasing
    weight, stable
    {
        int s = order[a], b = a;
        while (b > 0 && w[items[order[b - 1]]] < w[items[s]])
        { order[b] = order[b - 1]; b = b - 1; }
        order[b] = s;
    }
    Vec load;
    asg.assign(k, -1);
    for (int a = 0; a < k; ++a)
    {
        int s = order[a], b = 0;
        while (b < (int) load.size() && load[b] + w[items[s]] > c) b = b + 1;
        if (b == (int) load.size()) load.push_back(0);
        asg[s] = b;
        load[b] = load[b] + w[items[s]];
    }
    return load.size();
}
```

```
// exact repair for the BPP: optimal packing of the removed items in new bins,
    by a
// depth-first branch-and-bound (items by non-increasing weight; each item goes
     in an
// open bin or in a new one)

static void Branch(int a, const Vec& order, const Vec& items, const Vec& w, int
     c,
                   Vec& load, Vec& cur, Vec& asg, int& best)
{
    if ((int) load.size() >= best) return;               // bound: cannot
    improve
    if (a == (int) order.size()) { best = load.size(); asg = cur; return; }
    int s = order[a];
    for (int b = 0; b < (int) load.size(); ++b)
    {
        if (load[b] + w[items[s]] > c) continue;
```

```
        load[b] = load[b] + w[items[s]]; cur[s] = b;
        Branch(a + 1, order, items, w, c, load, cur, asg, best);
        load[b] = load[b] - w[items[s]];
    }
    load.push_back(w[items[s]]); cur[s] = load.size() - 1;
    Branch(a + 1, order, items, w, c, load, cur, asg, best);
    load.pop_back();
}

int RepairExact(const Vec& items, const Vec& w, int c, Vec& asg)
{
    int k = items.size();
    Vec order(k);
    for (int s = 0; s < k; ++s) order[s] = s;
    for (int a = 1; a < k; ++a)
    {
        int s = order[a], b = a;
        while (b > 0 && w[items[order[b - 1]]] < w[items[s]])
        { order[b] = order[b - 1]; b = b - 1; }
        order[b] = s;
    }
    Vec load, cur(k, -1);
    int best = k + 1;
    Branch(0, order, items, w, c, load, cur, asg, best);
    return best;
}
```

```
// large neighbourhood search for the BPP: empty q bins (DestroyBins) and repack their
// items with FFD (exact = false) or optimally (exact = true); the new packing is
// accepted if it uses fewer bins, or the same number of bins and a sum of squared loads
// not smaller
// bin, nb: starting packing on input, final packing on output; returns the number of bins

int LnsBPP(const Vec& w, int c, Vec& bin, int nb, int q, bool exact, int maxIter,
           Random& r)
{
    int n = w.size(), sum = 0;
    for (int i = 0; i < n; ++i) sum = sum + w[i];
    int LB = (sum + c - 1) / c;
    int it = 0;
    while (it < maxIter && nb > LB)
    {
        it = it + 1;
        Vec empty = DestroyBins(bin, nb, w, q, r);      // destroy
        vector<bool> isEmptied(nb, false);
        for (int s = 0; s < (int) empty.size(); ++s) isEmptied[empty[s]] = true;
        Vec items, newbin(n, -1), newIndex(nb, -1);
        int kept = 0;
        for (int b = 0; b < nb; ++b)
            if (!isEmptied[b]) { newIndex[b] = kept; kept = kept + 1; }
        for (int i = 0; i < n; ++i)
        {
```

```
            if (isEmptied[bin[i]]) items.push_back(i);
            else newbin[i] = newIndex[bin[i]];
        }
        Vec asg;                                              // repair
        int k = exact ? RepairExact(items, w, c, asg) : RepairFFD(items, w, c,
    asg);
        for (int s = 0; s < (int) items.size(); ++s) newbin[items[s]] = kept +
    asg[s];
        int nb2 = kept + k;
        Vec load(nb, 0), load2(nb2, 0);
        for (int i = 0; i < n; ++i) { load[bin[i]] += w[i]; load2[newbin[i]] +=
     w[i]; }
        // auxiliary objective: sum of squared loads
        long long aux = 0, aux2 = 0;
        for (int b = 0; b < nb; ++b) aux = aux + load[b] * load[b];
        for (int b = 0; b < nb2; ++b) aux2 = aux2 + load2[b] * load2[b];
        // acceptance
        if (nb2 < nb || (nb2 == nb && aux2 >= aux)) { bin = newbin; nb = nb2; }
    }
    return nb;
}
```

4.14.7 Algoritmo genetico e memetico

- L'algoritmo genetico e memetico per il KP (Sezione 4.10):

```
// repair of an infeasible KP solution: removes the selected item of minimum
    efficiency
// (ties: largest index) until the capacity is respected

void RepairKP(const Vec& p, const Vec& w, int c, Vec& x)
{
    int n = p.size(), W = Weight(w, x);
    while (W > c)
    {
        int s = -1;
        for (int i = 0; i < n; ++i)
            if (x[i] == 1 && (s < 0 || p[i] * w[s] <= p[s] * w[i])) s = i;
        x[s] = 0;
        W = W - w[s];
    }
}
```

```
// binary tournament: two random individuals, the fitter one wins (ties: the
    first drawn)

int Tournament(const Vec& fit, Random& r)
{
    int P = fit.size();
    int a = (int) (Rand(r) * P);
    int b = (int) (Rand(r) * P);
    return (fit[a] >= fit[b]) ? a : b;
}
```

```
// steady-state genetic algorithm for the KP with binary encoding: binary
    tournament
// selection,
```

```cpp
// one-point crossover, bit-flip mutation with probability pm, repair; if memetic is true,
// every new individual is also improved by KPLocalSearch; the child replaces the worst
// individual (ties: last index) if it is better and not already in the population

int GeneticKP(const Vec& p, const Vec& w, int c, Vec& best, int P, double pm, int maxIter,
              bool memetic, Random& r)
{
    int n = p.size();
    Matrix pop(P, Vec(n, 0));
    Vec fit(P, 0);
    for (int s = 0; s < P; ++s)                          // random initial population
    {
        for (int i = 0; i < n; ++i) pop[s][i] = (Rand(r) < 0.5) ? 1 : 0;
        RepairKP(p, w, c, pop[s]);
        if (memetic) KPLocalSearch(p, w, c, pop[s]);
        fit[s] = Profit(p, pop[s]);
    }
    for (int it = 1; it <= maxIter; ++it)
    {
        int a = Tournament(fit, r);                      // selection
        int b = Tournament(fit, r);
        int cut = 1 + (int) (Rand(r) * (n - 1));         // one-point crossover
        Vec child(n);
        for (int i = 0; i < n; ++i) child[i] = (i < cut) ? pop[a][i] : pop[b][i];
        for (int i = 0; i < n; ++i)                      // mutation
            if (Rand(r) < pm) child[i] = 1 - child[i];
        RepairKP(p, w, c, child);                        // repair
        if (memetic) KPLocalSearch(p, w, c, child);      // memetic: local search
        int fc = Profit(p, child);
        int worst = 0;
        for (int s = 1; s < P; ++s) if (fit[s] <= fit[worst]) worst = s;
        bool present = false;
        for (int s = 0; s < P; ++s) if (pop[s] == child) present = true;
        // replacement
        if (!present && fc > fit[worst]) { pop[worst] = child; fit[worst] = fc; }
    }
    int sb = 0;
    for (int s = 1; s < P; ++s) if (fit[s] > fit[sb]) sb = s;
    best = pop[sb];
    return fit[sb];
}
```

Bibliografia

- 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.
- 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 (a cura di), *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.
- 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.

Capitolo 5

Euristiche MIP per i problemi classici

Prerequisiti

- Dal modulo *Programmazione lineare intera*, Capitolo 10 «Euristiche MIP»: i metodi di questo capitolo (arrotondamento, diving, feasibility pump, RINS, local branching, proximity search, relax-and-fix e fix-and-optimize), i loro pseudocodici e le condizioni in cui funzionano.
- Dal modulo *Problemi di ottimizzazione combinatoria*, le formulazioni dei problemi classici: i modelli del problema dello zaino 0/1, del problema del bin packing, del problema del commesso viaggiatore, di $P \,||\, C_{\max}$ e del lot sizing.
- Da questo modulo, i capitoli sugli algoritmi euristici, da «Algoritmi euristici: definizioni» a «Metaeuristiche» («Euristiche costruttive», «Ricerca locale»): le euristiche costruttive (GR, FF, NN, LPT), la ricerca locale e le metaeuristiche, le cui soluzioni sono i punti di partenza di questo capitolo.

Obiettivi

Alla fine di questo capitolo saremo in grado di:

- scrivere i modelli MIP dei problemi classici nella forma richiesta dalle euristiche basate sulla PLI e calcolarne i rilassamenti lineari;
- eseguire a mano arrotondamento, diving, feasibility pump, RINS, local branching e proximity search sul problema dello zaino 0/1, sul bin packing e su $P \,||\, C_{\max}$;
- eseguire relax-and-fix e fix-and-optimize su un'istanza di lot sizing;
- spiegare perché ciascun metodo riesce o fallisce su ciascun problema;
- calcolare il gap di una soluzione euristica quando è disponibile un bound duale.

5.1 Euristiche basate sulla PLI

Definizione 5.1: Euristica basata sulla PLI

Un algoritmo euristico per un problema con un modello MIP è **basato sulla PLI** (una **matheuristica**) se usa il rilassamento lineare del modello e un solver MIP per cercare una soluzione ammissibile.

- Le euristiche basate sulla PLI sono definite nel modulo *Programmazione lineare intera*, Capitolo 10 «Euristiche MIP». In questo capitolo le applichiamo ai problemi classici, con le istanze di riferimento dei capitoli precedenti.

- La notazione è quella di quel capitolo: un MIP con variabili intere $\mathscr{D}$, $\tilde{\boldsymbol{x}}$ è una soluzione ottima del rilassamento lineare, $\bar{\boldsymbol{x}}$ una soluzione ammissibile (la soluzione incumbent), $[\,v\,] = \lfloor v + \frac{1}{2} \rfloor$ l'arrotondamento all'intero più vicino, $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) = \sum_{j \in \mathscr{D}} |x_j - \bar{x}_j|$ la distanza da un punto intero. Un problema di massimo, come il KP, si tratta scambiando min e max, $\leq$ e $\geq$ nei confronti fra valori obiettivo.

- Gli esempi sono risolti con il solver MIP Gurobi, usato da Python tramite `gurobipy`. Quando un LP ha più soluzioni ottime, il testo riporta quella restituita dal solver; i codici sono nell'ultima sezione.

Le euristiche costruttive, la ricerca locale e le metaeuristiche lavorano su una rappresentazione delle soluzioni; un'euristica basata sulla PLI lavora sul modello del problema, attraverso il suo rilassamento lineare e un solver MIP.

5.2 I problemi e i loro modelli MIP

- Le euristiche sono illustrate sui quattro problemi dei capitoli precedenti: il problema dello zaino 0/1 (KP), il problema del bin packing (BPP), il problema del commesso viaggiatore (TSP) e $P \,||\, C_{\max}$. Le loro definizioni e ipotesi sono nelle rispettive pagine.

- Le istanze di riferimento sono richiamate nella tabella seguente, con i valori calcolati nei capitoli precedenti sugli algoritmi euristici.

problema	istanza di riferimento	valori
KP	$p = (10, 7, 6, 4)$, $w = (5, 4, 3, 3)$, $c = 9$	GR: $\{1, 3\}$, profitto 16; $z(I) = 17$, $\{1, 2\}$
BPP	$w = (4, 4, 5, 3, 2, 3)$, $c = 7$	FF: 4 contenitori; $z(I) = 3$
TSP	5 città, le distanze di riferimento	NN dalla città 1: 25; $z(I) = 18$
$P \,||\, C_{\max}$	$t = (5, 5, 4, 4, 3, 3, 3)$, $m = 3$	LPT: $J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$, 11; $z(I) = 9$

- I modelli MIP sono quelli del modulo *Problemi di ottimizzazione combinatoria*: il modello ILP_{KP} del capitolo «Formulazioni dei problemi di zaino», il modello ILP_{BPP} del capitolo «Formulazioni dei problemi di packing», il modello $\text{ILP}_{\text{TSP}}^{\text{DFJ}}$ del capitolo «Formulazioni dei problemi di routing» e il modello $\text{ILP}_{P\|C_{\max}}$ del capitolo «Formulazioni dei problemi di scheduling». Richiamiamo i tre modelli usati negli esempi, con le variabili e la notazione di quei capitoli.

5.2.1 Il problema dello zaino 0/1

- La variabile binaria x_j vale 1 se l'oggetto j è selezionato nello zaino, e 0 altrimenti, per ogni $j \in \mathscr{I}$.

Usando queste variabili, un modello PLI per il problema dello zaino 0/1 (KP) è il seguente:

$$\max \quad \sum_{j \in \mathscr{I}} p_j \, x_j \tag{5.1a}$$

$$\text{soggetto a} \quad \sum_{j \in \mathscr{I}} w_j \, x_j \leq c, \tag{5.1b}$$

$$x_j \in \{0,1\}, \qquad \forall j \in \mathscr{I}. \tag{5.1c}$$

Questo modello PLI si chiama modello ILP_{KP}.

5.2.2 Il problema del bin packing

- Dato un upper bound m sul numero di contenitori usati in una soluzione ottima, e l'insieme $\mathscr{B} = \{1, 2, \ldots, m\}$ dei contenitori disponibili, la variabile binaria x_{jb} vale 1 se l'oggetto j è inserito nel contenitore b, e la variabile binaria y_b vale 1 se il contenitore b è usato, per ogni $j \in \mathscr{I}$ e $b \in \mathscr{B}$.

Usando queste variabili, un modello PLI per il problema del bin packing (BPP) è il seguente:

$$\min \quad \sum_{b \in \mathscr{B}} y_b \tag{5.2a}$$

$$\text{soggetto a} \quad \sum_{b \in \mathscr{B}} x_{jb} = 1, \qquad \forall j \in \mathscr{I}, \tag{5.2b}$$

$$-\sum_{j \in \mathscr{I}} w_j \, x_{jb} + c \, y_b \geq 0, \qquad \forall b \in \mathscr{B}, \tag{5.2c}$$

$$x_{jb} \in \{0,1\}, \qquad \forall j \in \mathscr{I}, \ \forall b \in \mathscr{B}, \tag{5.2d}$$

$$y_b \in \{0,1\}, \qquad \forall b \in \mathscr{B}. \tag{5.2e}$$

Questo modello PLI si chiama modello ILP_{BPP}.

- Il contenuto del contenitore b è lo schema di riempimento $P_b = \{j \in \mathscr{I} : x_{jb} = 1\}$. Il numero n di oggetti è sempre un upper bound m valido; sull'istanza di riferimento usiamo $m = n = 6$.

5.2.3 Il problema del commesso viaggiatore

- Il modello $\text{ILP}_{\text{TSP}}^{\text{DFJ}}$ ha una variabile binaria x_{ij} per ogni arco $(i,j) \in A$, i vincoli sugli archi uscenti da ogni città ed entranti in ogni città, e i vincoli di eliminazione dei sottogiri sui sottoinsiemi $S \subseteq \mathscr{V} \setminus \{1\}$ con $|S| \geq 2$. Sull'istanza di riferimento, con $n = 5$ città, ha 20 variabili e $5 + 5 + 11 = 21$ vincoli.

5.2.4 Il problema di scheduling $P \,||\, C_{\max}$

- Dato l'insieme $\mathscr{H} = \{1, 2, \ldots, m\}$ delle macchine, la variabile binaria x_{jh} vale 1 se il lavoro j è assegnato alla macchina h, per ogni $j \in \mathscr{J}$ e $h \in \mathscr{H}$, e la variabile continua non negativa $C_{\max}$ è il makespan.

Usando queste variabili, un modello PLI per $P\|C_{\max}$ è il seguente:

$$\min \quad C_{\max} \tag{5.3a}$$

$$\text{soggetto a} \quad \sum_{h \in \mathscr{H}} x_{jh} = 1, \qquad \forall j \in \mathscr{J}, \tag{5.3b}$$

$$-\sum_{j \in \mathscr{J}} t_j \, x_{jh} + C_{\max} \geq 0, \qquad \forall h \in \mathscr{H}, \tag{5.3c}$$

$$x_{jh} \in \{0,1\}, \qquad \forall j \in \mathscr{J}, \ \forall h \in \mathscr{H}, \tag{5.3d}$$

$$C_{\max} \geq 0. \tag{5.3e}$$

Questo modello PLI si chiama modello $\text{ILP}_{P\|C_{\max}}$.

- $C_{\max}$ è l'unica variabile continua dei quattro modelli: dato l'assegnamento, il completamento la pone uguale al carico massimo $L_h = \sum_{j \in \mathscr{J}} t_j \, x_{jh}$.

5.2.5 I rilassamenti lineari sulle istanze di riferimento

- La tabella seguente riporta la dimensione dei quattro modelli sulle istanze di riferimento, il valore $z(\text{LP})$ dei loro rilassamenti lineari, calcolato con Gurobi, e il valore ottimo $z(I)$.

problema	variabili	vincoli	$z(\text{LP})$	$z(I)$	soluzione LP intera?
KP	4	1	17.75	17	no
BPP	42	12	3	3	no
TSP	20	21	18	18	sì
$P \,||\, C_{\max}$	22	10	9	9	no

- Su tutte e quattro le istanze il valore LP, arrotondato nel verso del bound (per difetto per il KP, per eccesso per i tre problemi di minimo), è già il valore ottimo. Su queste istanze i rilassamenti sono *forti*: la difficoltà è trovare una soluzione che raggiunga il bound, non dimostrarlo.

- È la situazione tipica in cui le euristiche basate sulla PLI sono utili.

- Sull'istanza di riferimento del TSP la soluzione LP del modello $\text{ILP}_{\text{TSP}}^{\text{DFJ}}$ è essa stessa il giro ottimo $(1, 3, 5, 2, 4)$, con $\tilde{x}_{13} = \tilde{x}_{35} = \tilde{x}_{52} = \tilde{x}_{24} = \tilde{x}_{41} = 1$: ogni metodo di questo capitolo lo trova subito. Il rilassamento lineare del modello $\text{ILP}_{\text{TSP}}^{\text{MTZ}}$ è più debole: il suo valore è 17. Per questo il TSP non è usato negli altri esempi di questo capitolo.

Esempio 5.1: Le soluzioni LP delle istanze di riferimento

KP. La soluzione LP è $\tilde{\boldsymbol{x}} = \left(1, \frac{1}{4}, 1, 0\right)$: gli oggetti sono presi per efficienza non crescente, e l'oggetto critico 2 è preso per la frazione $\frac{9-8}{4} = \frac{1}{4}$ del suo peso, quindi $z(\text{LP}) = 10 + \frac{7}{4} + 6 = 17.75$.

$P\,||\,C_{\max}$. La soluzione LP restituita dal solver è (zeri omessi)

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
$\tilde{x}_{j1}$		$\frac{3}{5}$				1	1
$\tilde{x}_{j2}$			$\frac{1}{2}$	1	1		
$\tilde{x}_{j3}$	1	$\frac{2}{5}$	$\frac{1}{2}$				

con carichi $L_1 = 3 + 5 \cdot \frac{3}{5} + 3 = 9$, $L_2 = 4 \cdot \frac{1}{2} + 4 + 3 = 9$, $L_3 = 5 + 5 \cdot \frac{2}{5} + 4 \cdot \frac{1}{2} = 9$ e $\tilde{C}_{\max} = 9 = \frac{27}{3}$. Il LP ha infinite soluzioni ottime: ogni assegnamento frazionario con tre carichi uguali a 9 è ottimo.

BPP. La soluzione LP restituita dal solver mette l'oggetto 1 nel contenitore 6, l'oggetto 2 nel contenitore 4, l'oggetto 3 per $\frac{4}{5}$ nel contenitore 3 e per $\frac{1}{5}$ nel contenitore 4, l'oggetto 4 nel contenitore 6, l'oggetto 5 nel contenitore 1, l'oggetto 6 nel contenitore 3, e apre i contenitori per le frazioni

$$\tilde{\boldsymbol{y}} = \left(\frac{2}{7}, 0, 1, \frac{5}{7}, 0, 1\right),$$

appena sufficienti a contenerne i carichi: 2, 0, 7, 5, 0, 7. Il suo valore è $\frac{21}{7} = 3$.

5.3 Arrotondamento

- L'*arrotondamento* arrotonda la soluzione LP al punto intero più vicino e completa le variabili continue; fallisce se il punto arrotondato viola un vincolo. L'arrotondamento per difetto è sempre ammissibile quando tutti i coefficienti di A sono non negativi e $b \geq 0$ (l'osservazione sull'arrotondamento per difetto del Capitolo 10 «Euristiche MIP» del modulo *Programmazione lineare intera*); un vincolo di uguaglianza, come i vincoli di assegnamento del BPP e di $P\,||\,C_{\max}$, è quasi sempre violato da arrotondamenti indipendenti.

Esempio 5.2: Arrotondamento sulle istanze di riferimento

KP. La soluzione LP $\tilde{\boldsymbol{x}} = \left(1, \frac{1}{4}, 1, 0\right)$ è arrotondata a $\bar{\boldsymbol{x}} = (1, 0, 1, 0)$: peso $8 \leq 9$, profitto 16. È la soluzione di GR del capitolo «Euristiche costruttive». Arrotondare invece $\tilde{x}_2 = \frac{1}{4}$

per eccesso darebbe $(1, 1, 1, 0)$, di peso $12 > 9$: inammissibile.

$P \,||\, C_{\max}$. Nella soluzione LP dell'Esempio 5.1, $\left[\frac{3}{5}\right] = 1$ e $\left[\frac{2}{5}\right] = 0$ mandano il lavoro 2 sulla macchina 1, ma $\left[\frac{1}{2}\right] = 1$ assegna il lavoro 3 a *entrambe* le macchine 2 e 3:

$$\bar{x}_{31} + \bar{x}_{32} + \bar{x}_{33} = 0 + 1 + 1 = 2 \neq 1.$$

Il vincolo di assegnamento (5.3b) del lavoro 3 è violato: **l'arrotondamento fallisce**.

BPP. Arrotondando la soluzione LP dell'Esempio 5.1 otteniamo $\bar{\boldsymbol{y}} = (0, 0, 1, 1, 0, 1)$ e mettiamo l'oggetto 3 nel contenitore 3. Due vincoli di capacità (5.2c) sono violati: il contenitore 1 contiene l'oggetto 5 ma $\bar{y}_1 = 0$ ($-2 + 7 \cdot 0 \geq 0$ è falso), e il contenitore 3 contiene gli oggetti 3 e 6 ($-(5 + 3) + 7 \cdot 1 \geq 0$ è falso). **L'arrotondamento fallisce**.

- I tre esiti riassumono il metodo: sul KP, con un solo vincolo di packing, l'arrotondamento è sicuro (l'osservazione sull'arrotondamento per difetto del Capitolo 10 «Euristiche MIP» del modulo *Programmazione lineare intera*); sui problemi con vincoli di assegnamento fallisce, perché il LP distribuisce un lavoro o un oggetto su più macchine o contenitori, e l'arrotondamento non può decidere dove vada.

5.4 Diving

- Il *diving* fissa una variabile frazionaria alla volta e risolve di nuovo il LP: il fractional diving fissa la variabile meno frazionaria al suo intero più vicino, il guided diving fissa la variabile più vicina alla soluzione incumbent al valore dell'incumbent; fallisce quando il LP diventa inammissibile. I pareggi si risolvono a favore della variabile che viene prima nell'ordine delle variabili ($x_{11}, x_{12}, \ldots$ per i modelli di assegnamento, poi $C_{\max}$ o $y_1, y_2, \ldots$).

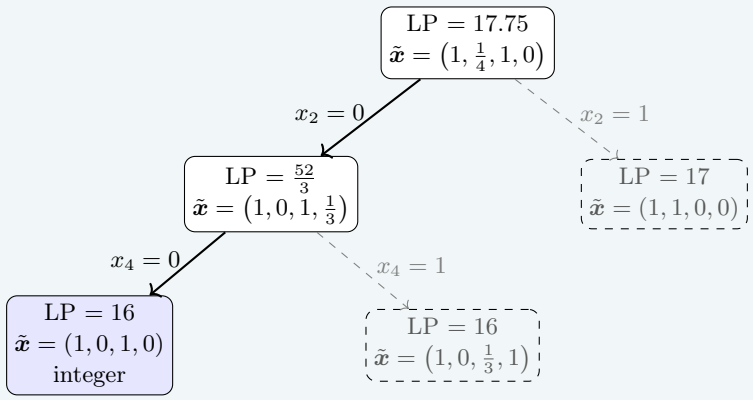
Esempio 5.3: Fractional diving sul KP

Passo 1. La soluzione LP è $\tilde{\boldsymbol{x}} = \left(1, \frac{1}{4}, 1, 0\right)$, di valore 17.75. L'unica variabile frazionaria è x_2, arrotondata a $\left[\frac{1}{4}\right] = 0$: fissare $x_2 := 0$.

Passo 2. Senza l'oggetto 2, il LP prende gli oggetti 1 e 3 (peso 8) e la frazione $\frac{1}{3}$ dell'oggetto 4: $\tilde{\boldsymbol{x}} = \left(1, 0, 1, \frac{1}{3}\right)$, di valore $10 + 6 + \frac{4}{3} = \frac{52}{3} \approx 17.33$. Fissare $x_4 := \left[\frac{1}{3}\right] = 0$.

Passo 3. La soluzione LP $\tilde{\boldsymbol{x}} = (1, 0, 1, 0)$, di valore 16, è intera: il diving restituisce $\{1, 3\}$, di profitto 16.

Il diving è il cammino disegnato a linee continue nell'albero di branch-and-bound qui sotto; i nodi tratteggiati sono i rami che non visita mai, con il valore dei loro rilassamenti lineari.



Il ramo $x_2 = 1$, abbandonato al primo passo, contiene la soluzione ottima $\{1, 2\}$ di profitto 17, che è una soluzione ottima del suo rilassamento lineare. La prima decisione del diving, $\left[\frac{1}{4}\right] = 0$, era quella sbagliata.

Esempio 5.4: Fractional diving su $P\,||\,C_{\max}$

Il diving parte dalla soluzione LP dell'Esempio 5.1. La tabella riporta, per ogni passo, il valore del LP, le sue variabili frazionarie e il fissaggio; a parità di frazionarietà $|\tilde{x}_j - [\tilde{x}_j]|$ fra due variabili decide l'ordine $x_{11}, x_{12}, x_{13}, x_{21}, \ldots$

passo	valore LP	variabili frazionarie	fissaggio
1	9	$x_{21} = \frac{3}{5}$, $x_{23} = \frac{2}{5}$, $x_{32} = \frac{1}{2}$, $x_{33} = \frac{1}{2}$	$x_{21} := 1$
2	9	$x_{11} = \frac{1}{5}$, $x_{13} = \frac{4}{5}$, $x_{32} = \frac{3}{4}$, $x_{33} = \frac{1}{4}$	$x_{11} := 0$
3	9	$x_{32} = \frac{3}{4}$, $x_{33} = \frac{1}{4}$, $x_{41} = \frac{1}{4}$, $x_{43} = \frac{3}{4}$	$x_{32} := 1$
4	9	$x_{41} = \frac{1}{4}$, $x_{43} = \frac{3}{4}$, $x_{52} = \frac{2}{3}$, $x_{53} = \frac{1}{3}$	$x_{41} := 0$
5	9	$x_{61} = \frac{1}{3}$, $x_{62} = \frac{2}{3}$	$x_{61} := 0$
6	9	$x_{12} = \frac{3}{5}$, $x_{13} = \frac{2}{5}$, $x_{71} = \frac{1}{3}$, $x_{72} = \frac{2}{3}$	$x_{71} := 0$
7	9.5	$x_{42} = \frac{1}{8}$, $x_{43} = \frac{7}{8}$	$x_{42} := 0$
8	9.5	$x_{62} = \frac{1}{6}$, $x_{63} = \frac{5}{6}$	$x_{62} := 0$
9	9.5	$x_{12} = \frac{1}{2}$, $x_{13} = \frac{1}{2}$	$x_{12} := 1$
10	9.5	$x_{72} = \frac{1}{6}$, $x_{73} = \frac{5}{6}$	$x_{72} := 0$
11	10	nessuna	—

L'ultima soluzione LP è intera: $J_1 = \{2, 5\}$, $J_2 = \{1, 3\}$, $J_3 = \{4, 6, 7\}$, con carichi 8, 9, 10 e makespan 10. Il diving migliora l'arrotondamento, che era fallito, e *LPT* (11), ma non raggiunge l'ottimo 9.

- Per sei passi il valore LP resta 9: il LP di $P\,||\,C_{\max}$ ha un insieme enorme di soluzioni ottime, e fissare una variabile sposta solo il LP su un'altra di esse. Il valore LP non dice nulla sulla qualità delle decisioni finché il carico di una macchina non è forzato sopra 9, al passo 7, e allora è troppo tardi per annullare le decisioni che lo hanno causato. Gli obiettivi min-max sono piatti per il LP quanto lo sono per la ricerca locale.

Esempio 5.5: Guided diving su $P\,||\,C_{\max}$

La soluzione incumbent è la soluzione di LPT, $J_1 = \{1,5,7\}$, $J_2 = \{2,6\}$, $J_3 = \{3,4\}$, di makespan 11, ossia $\bar{x}_{11} = \bar{x}_{51} = \bar{x}_{71} = \bar{x}_{22} = \bar{x}_{62} = \bar{x}_{33} = \bar{x}_{43} = 1$ e tutte le altre $\bar{x}_{jh} = 0$.

Passo 1. Nella soluzione LP dell'Esempio 5.1 le variabili frazionarie e le loro distanze dall'incumbent sono

$$|\tilde{x}_{21} - \bar{x}_{21}| = \tfrac{3}{5}, \quad |\tilde{x}_{23} - \bar{x}_{23}| = \tfrac{2}{5}, \quad |\tilde{x}_{32} - \bar{x}_{32}| = \tfrac{1}{2}, \quad |\tilde{x}_{33} - \bar{x}_{33}| = \tfrac{1}{2}.$$

La più vicina è x_{23}: fissare $x_{23} := \bar{x}_{23} = 0$.

Passo 2. La nuova soluzione LP, di valore 9, ha le variabili frazionarie $x_{12} = \frac{2}{5}$, $x_{13} = \frac{3}{5}$, $x_{31} = \frac{1}{2}$, $x_{32} = \frac{1}{2}$, a distanze $\frac{2}{5}$, $\frac{3}{5}$, $\frac{1}{2}$, $\frac{1}{2}$ dall'incumbent. Fissare $x_{12} := \bar{x}_{12} = 0$.

Passo 3. La soluzione LP è intera, di valore 9: $J_1 = \{5,6,7\}$, $J_2 = \{2,4\}$, $J_3 = \{1,3\}$, tutti i carichi uguali a 9. Il guided diving trova una soluzione **ottima** dopo due LP.

- Guidato da una soluzione di makespan 11, il diving termina in una soluzione di makespan 9: l'incumbent decide solo i *valori* delle due variabili fissate, entrambi uguali a 0, mentre il LP ricostruisce il resto dell'assegnamento.

- Sull'istanza di riferimento del BPP, il fractional diving esegue 8 fissaggi, tutti con valore LP 3; il nono, $x_{12} := 1$, mette l'oggetto 1 nel contenitore 2, che contiene già l'oggetto 2 per un fissaggio precedente, e $4 + 4 > 7$ rende il LP inammissibile: **il diving fallisce**. I fissaggi $x_{jb} = 1$ assegnano gli oggetti ai contenitori uno alla volta, e il LP del BPP è così debole (3 a ogni passo) da non dare alcun segnale prima del vicolo cieco.

5.5 La feasibility pump

- La *feasibility pump* alterna la proiezione di un punto intero sulla regione ammissibile del LP (un LP che minimizza Δ) e l'arrotondamento della proiezione, con un flip quando l'arrotondamento restituisce lo stesso punto intero e un riavvio casuale quando la pump cicla, finché la distanza è 0. I numeri casuali sono quelli del generatore *minimal standard* di Park e Miller con i semi $1, 2, \ldots$, come nel capitolo «Metaeuristiche».

Esempio 5.6: La feasibility pump su $P\,||\,C_{\max}$

Iterazione 0. La soluzione LP dell'Esempio 5.1 è arrotondata come nell'Esempio 5.2: il lavoro 3 è assegnato alle macchine 2 e 3, e $\bar{\boldsymbol{x}}^0$ è inammissibile.

Iterazione 1. La proiezione minimizza $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^0)$, che è lineare perché tutte le variabili intere sono binarie. Il solver restituisce il punto di $\mathcal{L}$ che mantiene tutti gli assegnamenti di $\bar{\boldsymbol{x}}^0$ e toglie il lavoro 3 dalla macchina 2, ossia porta x_{32} da 1 a 0: $\Delta = 1$ (togliere il lavoro 3 dalla macchina 3 sarebbe altrettanto vicino). Il risultato è intero:

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

con carichi 11, 7, 9, e il suo arrotondamento è il nuovo punto intero $\bar{\boldsymbol{x}}^1$.

Iterazione 2. $\bar{\boldsymbol{x}}^1 \in \mathcal{L}$, la distanza è 0, e la pump restituisce una soluzione di makespan 11, dopo due proiezioni.

- La soluzione ha lo stesso makespan di LPT: la proiezione si occupa dei vincoli di assegnamento e ignora i carichi, poiché $C_{\max}$ è continua e non entra in Δ. La pump risponde alla domanda «esiste una soluzione ammissibile?», che per $P\,||\,C_{\max}$ non era una domanda difficile.

Esempio 5.7: La feasibility pump sul BPP

Sull'istanza di riferimento del BPP, dove l'arrotondamento e il diving falliscono, la pump è eseguita con i semi $1, 2, \ldots, 10$ dei numeri casuali. La tabella riporta il numero di proiezioni, di flip e di riavvii, il numero di contenitori aperti dalla soluzione trovata ($\sum_b \bar{y}_b$, il valore della soluzione) e il numero di contenitori non vuoti.

seme	1	2	3	4	5	6	7	8	9	10
proiezioni	15	12	9	9	6	6	19	5	10	20
flip	8	5	3	4	3	3	8	2	5	8
riavvii	4	3	3	2	1	1	5	1	2	4
contenitori aperti	4	4	4	5	5	4	3	4	4	5
contenitori non vuoti	4	4	4	3	5	4	3	4	4	4

La pump trova una soluzione ammissibile con ogni seme, ma sempre dopo qualche riavvio: senza di essi ciclerebbe. Solo con il seme 7 la soluzione è ottima: $\{3,5\}$, $\{2,4\}$, $\{1,6\}$, di valore 3. Con il seme 4 la pump apre 5 contenitori, due dei quali restano vuoti: il riempimento $\{3,5\}$, $\{1,6\}$, $\{2,4\}$ è ottimo, ma la soluzione ha valore 5, perché nulla nella funzione distanza chiede di chiudere i contenitori vuoti.

- La pump è un metodo per trovare una *prima* soluzione ammissibile, e lo fa bene. La qualità della soluzione è compito dei metodi di miglioramento, che partono da una soluzione incumbent.

5.6 RINS, local branching e proximity search

- I tre metodi partono da una soluzione incumbent $\bar{\boldsymbol{x}}$ e risolvono un sotto-MIP con il vincolo di cutoff $f(\boldsymbol{x}) \le f(\bar{\boldsymbol{x}}) - \theta$, che qui ha $\theta = 1$ poiché i valori obiettivo sono interi: RINS fissa le variabili intere su cui la soluzione LP e l'incumbent concordano; il local branching aggiunge $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) \le k$ e aumenta k quando non trova nulla; la proximity search minimizza $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}})$.

5.6.1 RINS

Esempio 5.8: RINS sul KP

La soluzione incumbent è la soluzione $\bar{\boldsymbol{x}} = (1, 0, 1, 0)$ di GR, di profitto 16, e la soluzione LP è $\tilde{\boldsymbol{x}} = \left(1, \frac{1}{4}, 1, 0\right)$. Concordano su $x_1 = 1$, $x_3 = 1$, $x_4 = 0$, che vengono fissate. Il sotto-MIP, con il vincolo di capacità (5.1b) e il vincolo di cutoff, è

$$\begin{aligned} \max \quad & 16 + 7\,x_2 & & \\ \text{soggetto a} \quad & 8 + 4\,x_2 & \le & \quad 9 \\ & 16 + 7\,x_2 & \ge & \quad 17 \\ & x_2 & \in & \quad \{0, 1\} \end{aligned}$$

ed è inammissibile: la capacità richiede $x_2 = 0$, il cutoff $x_2 = 1$. RINS restituisce l'incumbent. La soluzione ottima $\{1, 2\}$ ha $x_3 = 0$, ma x_3 è fissata a 1 perché il LP e l'incumbent concordano su di essa.

Esempio 5.9: RINS su $P\,||\,C_{\max}$

La soluzione incumbent è la soluzione di LPT ($J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$, makespan 11), e $\tilde{\boldsymbol{x}}$ è la soluzione LP dell'Esempio 5.1. La tabella confronta, per ogni lavoro, la macchina dell'incumbent con la soluzione LP.

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
macchina in $\bar{\boldsymbol{x}}$	1	2	3	3	1	2	1
macchine in $\tilde{\boldsymbol{x}}$	3	1 ($\frac{3}{5}$), 3 ($\frac{2}{5}$)	2 ($\frac{1}{2}$), 3 ($\frac{1}{2}$)	2	2	1	1
variabili fissate	$x_{12} = 0$	—	$x_{31} = 0$	$x_{41} = 0$	$x_{53} = 0$	$x_{63} = 0$	$x_{71} = 1$, $x_{72} = x_{73} = 0$

Solo il lavoro 7 è sulla stessa macchina in entrambe le soluzioni; per gli altri lavori, le due soluzioni concordano solo su alcune macchine che il lavoro *non* usa. Otto delle 21 variabili binarie sono fissate. Il sotto-MIP con il cutoff $C_{\max} \le 10$ è risolto da Gurobi, che restituisce

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

con tutti i carichi uguali a 9: **RINS trova una soluzione ottima**.

- Sul KP la soluzione incumbent e la soluzione LP sono troppo simili, e l'intorno di RINS ha una sola variabile libera; su $P\,||\,C_{\max}$ sono molto diverse, e l'intorno è abbastanza grande da contenere l'ottimo.

5.6.2 Local branching

- **Local branching e ricerca locale.** Con $k = 1$ l'intorno è l'intorno flip del KP del capitolo «Algoritmi euristici: definizioni»; con $k = 2$ contiene anche l'intorno swap. Per le variabili di assegnamento del BPP e di $P\,||\,C_{\max}$ la distanza ha un significato semplice.

Osservazione 5.2

Siano $\boldsymbol{x}$ e $\bar{\boldsymbol{x}}$ due assegnamenti ammissibili di $P\,||\,C_{\max}$. Allora $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) = 2\,r$, dove r è il numero di lavori assegnati a macchine diverse in $\boldsymbol{x}$ e in $\bar{\boldsymbol{x}}$.

Dimostrazione. Un lavoro sulla stessa macchina in $\boldsymbol{x}$ e $\bar{\boldsymbol{x}}$ contribuisce 0 a Δ. Un lavoro sulla macchina h in $\bar{\boldsymbol{x}}$ e sulla macchina $h' \neq h$ in $\boldsymbol{x}$ contribuisce $1 - x_{jh} = 1$ e $x_{jh'} = 1$, e 0 per le altre macchine. □

- Quindi, per $P\,||\,C_{\max}$, l'intorno $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) \leq 2$ è l'intorno **move** del capitolo «Ricerca locale» (un lavoro cambia macchina), e $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) \leq 4$ contiene l'intorno **swap** (due lavori cambiano macchina). Lo stesso vale per le variabili x_{jb} del BPP.

Esempio 5.10: Local branching sul KP

Partiamo da $\bar{\boldsymbol{x}}^1 = (1, 0, 1, 0)$, di profitto 16, con $k = 1$, $k_{\max} = 3$, $k_{step} = 1$ e $\theta = 1$.

$k = 1$. I vicini a distanza 1 sono i flip di $\bar{\boldsymbol{x}}^1$; con il cutoff $\sum_j p_j x_j \geq 17$ nessuno è ammissibile. Aggiungere $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^1) \geq 2$ e porre $k = 2$.

$k = 2$. Il sotto-MIP con $2 \leq \Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^1) \leq 2$ trova lo swap $(1, 1, 0, 0)$, di profitto 17: la nuova soluzione incumbent $\bar{\boldsymbol{x}}^2$. Aggiungere $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^1) \geq 3$.

$k = 2$, attorno a $\bar{\boldsymbol{x}}^2$. Nessuna soluzione di profitto almeno 18: aggiungere $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^2) \geq 3$ e porre $k = 3$. **$k = 3$.** Nemmeno ora c'è una soluzione di profitto almeno 18; k diventa $4 > k_{\max}$ e il metodo termina con $\{1, 2\}$, di profitto 17: l'ottimo.

Esempio 5.11: Local branching su $P\,||\,C_{\max}$

Partiamo dalla soluzione $\bar{\boldsymbol{x}}^1$ di LPT, di makespan 11, con $k = 2$, $k_{\max} = 6$, $k_{step} = 2$ e $\theta = 1$, in modo che il cutoff sia $C_{\max} \leq 10$, poi 9, e così via.

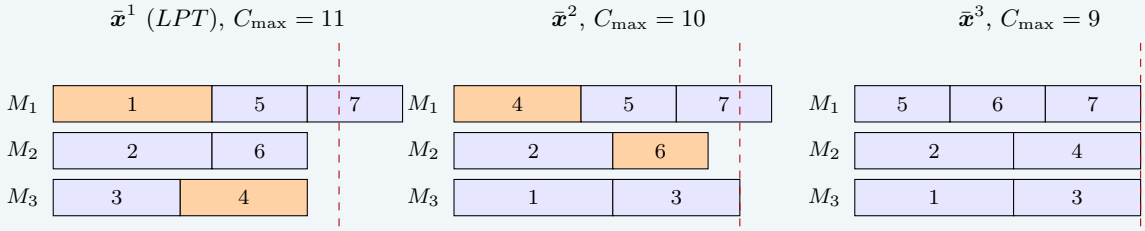
$k = 2$. Per l'Osservazione 5.2 l'intorno è l'intorno move, e LPT è un ottimo locale per esso (capitolo «Ricerca locale»): il sotto-MIP è inammissibile. Aggiungere $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^1) \geq 3$ e porre $k = 4$.

$k = 4$. La soluzione migliore con due lavori riassegnati ha makespan 10; il solver restituisce lo swap dei lavori 1 e 4, $\bar{\boldsymbol{x}}^2$ qui sotto. Aggiungere $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^1) \geq 5$.

$k = 4$, attorno a $\bar{\boldsymbol{x}}^2$. Il solver restituisce lo swap dei lavori 4 e 6, di makespan 9: $\bar{\boldsymbol{x}}^3$.

Aggiungere $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^2) \geq 5$.

$k = 4$ e $k = 6$, attorno a $\bar{\boldsymbol{x}}^3$. Nessuna soluzione di makespan 8: k diventa $8 > k_{\max}$, e il metodo termina con makespan 9, l'ottimo.



La linea tratteggiata è il lower bound 9; i lavori scambiati per ottenere la soluzione successiva sono colorati.

- Nel linguaggio del capitolo «Ricerca locale», il local branching esegue una ricerca locale con un intorno la cui dimensione è controllata da k, esplorato in modo esatto dal solver, e allarga l'intorno quando si blocca: è esattamente l'idea della variable neighbourhood search, con il solver che esegue l'esplorazione.

5.6.3 Proximity search

- Un sotto-MIP di proximity inammissibile con $\theta = 1$ dimostra che la soluzione incumbent è ottima, quando l'obiettivo assume valori interi (la proposizione del Capitolo 10 «Euristiche MIP» del modulo *Programmazione lineare intera*).

Esempio 5.12: Proximity search sul KP

Da $\bar{\boldsymbol{x}} = (1, 0, 1, 0)$, di profitto 16: la soluzione più vicina di profitto almeno 17 è $(1, 1, 0, 0)$, a distanza 2 (esce l'oggetto 3, entra l'oggetto 2), di profitto 17. Attorno a essa nessuna soluzione ha profitto almeno 18: per la condizione di ottimalità della proximity search (Capitolo 10 «Euristiche MIP» del modulo *Programmazione lineare intera*), $\{1, 2\}$ è **ottima**.

Esempio 5.13: Proximity search su $P \,||\, C_{\max}$

Dalla soluzione di *LPT*, di makespan 11:

iterazione	cutoff	$\min \Delta$	soluzione restituita	$C_{\max}$
1	$C_{\max} \leq 10$	4	$J_1 = \{5, 6, 7\}$, $J_2 = \{1, 2\}$, $J_3 = \{3, 4\}$	10
2	$C_{\max} \leq 9$	4	$J_1 = \{5, 6, 7\}$, $J_2 = \{1, 4\}$, $J_3 = \{2, 3\}$	9
3	$C_{\max} \leq 8$	—	inammissibile	—

Per l'Osservazione 5.2, distanza 4 significa due lavori riassegnati: i lavori 1 e 6 alla prima iterazione, i lavori 2 e 4 alla seconda. Nessuna soluzione a distanza 2 migliora il makespan, come per il local branching con $k = 2$. Alla terza iterazione il sotto-MIP è inammissibile, e la condizione di ottimalità della proximity search (Capitolo 10 «Euristiche MIP» del modulo *Programmazione lineare intera*) dimostra che il makespan 9 è ottimo.

- La dimostrazione di ottimalità viene dal solver, che dimostra l'inammissibilità dell'ultimo sotto-MIP con i propri bound duali: la proximity search, usata con $\theta = 1$, è un metodo esatto che migliora la soluzione incumbent strada facendo.

5.7 Relax-and-fix e fix-and-optimize sul lot sizing

- Relax-and-fix e fix-and-optimize decompongono il MIP in blocchi di variabili intere. Il lot sizing è un'applicazione naturale, perché le sue variabili sono indicizzate dai periodi $t \in T = \{1, 2, \dots, m\}$ di un orizzonte di pianificazione, e le decisioni di un periodo interagiscono soprattutto con quelle dei periodi vicini. I blocchi sono le variabili di setup di periodi consecutivi.

- Il modello è il modello ILP_{CLSP} del capitolo «Formulazioni dei problemi di lot sizing», con il valore forte di big-M $M_t = \min\{cap_t, D_{t,m}\}$, dove $D_{t,m} = \sum_{k=t}^{m} d_k$; per un'istanza senza capacità, $cap_t = D_{t,m}$. Le sue variabili sono la produzione x_t, la scorta s_t alla fine del periodo t e il setup y_t, per ogni periodo $t \in T$.

Usando queste variabili, un modello PLI per il problema di lot sizing capacitato (CLS) è il seguente:

$$\min \quad \sum_{t \in T} (sc_t\, y_t + vc_t\, x_t + hc_t\, s_t) \tag{5.4a}$$

$$\text{soggetto a} \quad s_{t-1} + x_t = d_t + s_t, \qquad \forall t \in T, \tag{5.4b}$$

$$x_t \le M_t\, y_t, \qquad \forall t \in T, \tag{5.4c}$$

$$x_t \le cap_t, \qquad \forall t \in T, \tag{5.4d}$$

$$s_0 = s_m = 0, \tag{5.4e}$$

$$x_t \in \mathbb{Z}_{\ge 0}, \qquad \forall t \in T, \tag{5.4f}$$

$$s_t \in \mathbb{Z}_{\ge 0}, \qquad \forall t \in T, \tag{5.4g}$$

$$y_t \in \{0, 1\}, \qquad \forall t \in T. \tag{5.4h}$$

Questo modello PLI si chiama modello ILP_{CLSP}.

- Fissati i setup y, resta un LP con dati interi e una matrice di rete, che ha una soluzione ottima intera. Le decisioni difficili sono le m variabili binarie: come nei codici, le euristiche trattano x_t e s_t come variabili continue, con $x_t \ge 0$ e $s_t \ge 0$ al posto di (5.4f) e (5.4g).

- Il rilassamento lineare è debole, perché una frazione $y_t = x_t/M_t$ di setup basta per produrre x_t.

Esempio 5.14: Relax-and-fix su un'istanza di lot sizing senza capacità

L'istanza del modulo *Modellazione MIP*, Capitolo 4 «Euristiche costruttive», ha $m = 5$ periodi, $d = (20, 10, 30, 40, 10)$, $sc_t = 50$, $vc_t = 0$ e $hc_t = 1$ per ogni $t \in T$, e nessuna capacità. Il suo rilassamento lineare ha valore $\frac{36985}{396} \approx 93.40$, l'euristica costruttiva del costo unitario minimo di quel capitolo dà 200, e il valore ottimo è 170. Relax-and-fix con un periodo per blocco, $B_t = \{y_t\}$, trova una soluzione di valore 170: **ottima**.

Esempio 5.15: Relax-and-fix su un'istanza capacitata

Consideriamo $m = 6$ periodi, $d = (50, 30, 10, 10, 40, 10)$, $sc_t = 40$, $vc_t = 0$, $hc_t = 1$ e $cap_t = 80$ per ogni $t \in T$. Il rilassamento lineare ha valore $\frac{2150}{21} \approx 102.38$ e il valore ottimo è 160, con setup nei periodi 1, 2, 5. Relax-and-fix con $B_t = \{y_t\}$:

passo r	valore del MIP del passo r	y della soluzione	fissaggio
1	$\frac{2465}{21} \approx 117.38$	$(1,\ 0.375,\ 0.143,\ 0.167,\ 1,\ 0)$	$y_1 := 1$
2	$\frac{2780}{21} \approx 132.38$	$(1,\ 0,\ 0.143,\ 0.167,\ 1,\ 0)$	$y_2 := 0$
3	$\frac{500}{3} \approx 166.67$	$(1,\ 0,\ 1,\ 0.167,\ 1,\ 0)$	$y_3 := 1$
4	170	$(1, 0, 1, 0, 1, 0)$	$y_4 := 0$
5	170	$(1, 0, 1, 0, 1, 0)$	$y_5 := 1$
6	170	$(1, 0, 1, 0, 1, 0)$	$y_6 := 0$

(i valori frazionari sono arrotondati a tre decimali). Il risultato ha setup nei periodi 1, 3, 5: produzione 80, 20, 50, scorte 30, 10, 10 alla fine dei periodi 1, 3, 5, costo $3 \cdot 40 + 50 = 170$. La decisione sbagliata è presa al passo 2: con il futuro rilassato, $y_2 = 0$ dà 132.38 e $y_2 = 1$ dà 142.38, quindi relax-and-fix risparmia il setup del periodo 2; ma poi la capacità impone un setup nel periodo 3, che il rilassamento aveva pagato solo per una frazione 0.143.

Esempio 5.16: Fix-and-optimize sull'istanza capacitata

Partiamo dalla soluzione di relax-and-fix dell'Esempio 5.15, $\bar{\boldsymbol{y}} = (1, 0, 1, 0, 1, 0)$ di costo 170, con le finestre di due periodi consecutivi $W_t = \{y_t, y_{t+1}\}$, $t = 1, 2, \ldots, 5$.

finestra	incumbent dopo la finestra	
$\{y_1, y_2\}$	170	nessun miglioramento
$\{y_2, y_3\}$	160	$\bar{\boldsymbol{y}} = (1, 1, 0, 0, 1, 0)$
$\{y_3, y_4\}$, $\{y_4, y_5\}$, $\{y_5, y_6\}$	160	nessun miglioramento
secondo passaggio, tutte le finestre	160	nessun miglioramento: stop

La finestra $\{y_2, y_3\}$ corregge la decisione sbagliata di relax-and-fix: il setup passa dal periodo 3 al periodo 2. Il piano produce 50, 50, 50 nei periodi 1, 2, 5, con scorte 20 e 10 alla fine dei periodi 2, 3 e 10 alla fine del periodo 5: costo $3 \cdot 40 + 40 = 160$, l'**ottimo**.

5.8 Le euristiche dentro il solver

- Le euristiche di Gurobi, i loro parametri e il MIP start sono descritti nel Capitolo 10 «Euristiche MIP» e nel Capitolo 9 «I solver MIP: che cosa fanno» del modulo *Programmazione lineare intera*. Una soluzione euristica calcolata fuori dal solver (con FFD, LPT, una ricerca locale) si può passare al solver come MIP start (l'attributo `Start` delle variabili).

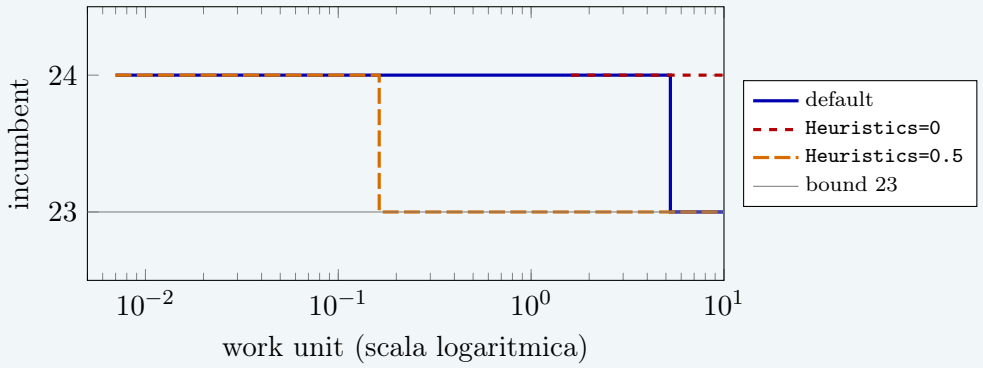
Esempio 5.17: L'effetto delle euristiche di Gurobi su un'istanza media del BPP

L'istanza ha $n = 60$ oggetti, con pesi $w_j = 20 + (s_j \bmod 41)$, dove $s_1, s_2, \ldots$ è la sequenza di Park–Miller dal seme 1, e capacità $c = 100$:

58	29	22	22	55	45	38	40	34	32
43	57	20	57	53	32	40	57	50	33
29	49	37	20	27	22	55	40	48	40
52	24	21	48	31	36	38	46	30	20
31	58	40	28	54	50	26	31	27	54
26	26	22	30	33	55	27	50	47	26

Il peso totale è 2271, quindi $LB = \lceil 2271/100 \rceil = 23$; FFD usa 24 contenitori, e il modello $\mathrm{ILP}_{\mathrm{BPP}}$ è costruito con $m = 24$ contenitori. Gurobi 13.0.3 è eseguito su un thread con un limite di 10 *work unit* (una misura deterministica dello sforzo di calcolo, all'incirca un secondo ciascuna), con parametri diversi.

parametri	incumbent	bound	nodi	work	incumbent trovate (work: valore)
default	23	23	2533	5.27	0.007: 24; 5.273: 23
`Heuristics=0`	24	23	4809	10.00	1.610: 24
`Heuristics=0.5`	23	23	1	0.16	0.007: 24; 0.163: 23
`MIPFocus=1`	23	23	1	0.16	0.007: 24; 0.160: 23
`NoRelHeurWork=1`	23	23	2533	6.48	0.007: 24; 6.477: 23
`RINS=0`	23	23	2959	5.86	0.007: 24; 5.861: 23



Con i parametri di default, una soluzione con 24 contenitori è trovata subito, e quella ottima solo dopo 2533 nodi. Dedicando più sforzo alle euristiche (`Heuristics=0.5` o `MIPFocus=1`) si trova 23 alla radice, circa 30 volte più velocemente; senza euristiche la prima soluzione arriva dopo 1.61 work unit, e l'ottimo non è trovato entro il limite. Poiché il bound vale 23 già alla radice, tutta la difficoltà di questa istanza è primale: trovare la soluzione, non dimostrarne l'ottimalità.

- I numeri dipendono dall'istanza, dalla versione del solver e dalla macchina; una sola istanza non dimostra nulla (capitolo «Algoritmi euristici: definizioni»). La lezione è qualitativa: sulle istanze il cui bound è forte, come il BPP con $LB = z(I)$, il tempo di soluzione è il tempo per trovare una buona soluzione, e lo decidono le euristiche del solver, o un buon MIP start.

5.9 Che cosa danno le euristiche basate sulla PLI

- Il capitolo «Algoritmi euristici: definizioni» si chiudeva con un avvertimento: un'euristica dà solo un lato del gap, e senza un bound duale non si può dire nulla sulla qualità della sua soluzione. Le euristiche dei capitoli precedenti danno *solo* bound primali.

- Un'euristica basata sulla PLI lavora con un modello MIP, e il modello porta con sé il suo rilassamento lineare; eseguita dentro un solver MIP, porta con sé anche il bound duale del branch-and-bound. Per la prima volta, l'euristica **sa quanto è lontana dall'ottimo** mentre è in esecuzione: il gap del capitolo «Algoritmi euristici: definizioni» si calcola in ogni momento, e quando vale 0 la soluzione è *dimostrata* ottima.

	euristiche dei capitoli precedenti			euristiche basate sulla PLI		
istanza	soluzione	bound duale	gap	soluzione	bound duale	gap
KP	GR: 16	LP: 17	6.25%	LB, PS: 17	PS: 17	0%
BPP	FF: 4	$\lceil 21/7 \rceil = 3$	33.3%	pump (seme 7): 3	LP: 3	0%
TSP	NN: 25	due lati: 17	47.1%	arrotondamento: 18	LP: 18	0%
$P \,\|\|\, C_{\max}$	LPT: 11	9	22.2%	RINS, LB, PS: 9	LP: 9	0%
lot sizing, capacitato	RF: 170	LP: 102.38	66.0%	FO: 160	solver: 160	0%
BPP medio	FFD: 24	23	4.3%	Gurobi: 23	23	0%

- Sul KP l'arrotondamento intero del valore LP è $\lfloor 17.75 \rfloor = 17$, e la proximity search dimostra da sola l'ottimalità di 17, per la condizione di ottimalità della proximity search (Capitolo 10 «Euristiche MIP» del modulo *Programmazione lineare intera*). Sull'istanza di lot sizing capacitato il bound LP è debole, e il gap è chiuso dal branch-and-bound del solver, che dimostra che 160 è ottimo.

Un'euristica basata sulla PLI non si limita a trovare soluzioni: dentro un solver MIP lavora con un bound duale, quindi il gap della sua soluzione è sempre noto, e un gap uguale a 0 certifica l'ottimalità.

5.10 Errori tipici

Attenzione

- **Arrotondare un problema con vincoli di uguaglianza.** Gli arrotondamenti indipendenti di un vincolo di assegnamento non sommano quasi mai a 1: sul BPP e su $P\,||\,C_{\max}$ l'arrotondamento fallisce.
- **Credere che un diving possa annullare una decisione.** Un diving è un solo cammino dell'albero, senza backtracking: il primo fissaggio sbagliato (sul KP, $x_2 := 0$) non viene mai rivisto.
- **Usare la feasibility pump per ottenere una buona soluzione.** Dopo il primo LP la pump ignora l'obiettivo: restituisce una soluzione ammissibile, eventualmente con contenitori vuoti lasciati aperti.
- **Fidarsi del fissaggio rigido.** RINS fissa le variabili su cui il LP e l'incumbent concordano, quindi non può trovare una soluzione migliore che ne cambi una.
- **Dimenticare il vincolo di cutoff.** Senza $f(\boldsymbol{x}) \leq f(\bar{\boldsymbol{x}}) - \theta$ il sotto-MIP restituisce di nuovo la soluzione incumbent; con esso, un sotto-MIP inammissibile dimostra un ottimo locale.
- **Dichiarare l'ottimalità senza le condizioni.** La condizione di ottimalità della proximity search (Capitolo 10 «Euristiche MIP» del modulo *Programmazione lineare intera*) richiede $\theta = 1$ e valori obiettivo interi su tutte le soluzioni ammissibili.
- **Ignorare la miopia di relax-and-fix.** Il futuro rilassato è ottimistico: una decisione che sembra economica può imporre un setup più avanti.
- **Generalizzare da una sola istanza o da una sola esecuzione del solver.** Cammini e tempi dipendono dalla versione del solver e dalle regole di pareggio, quindi le lezioni sono qualitative.

5.11 In sintesi

In sintesi

- Le euristiche basate sulla PLI del Capitolo 10 «Euristiche MIP» del modulo *Programmazione lineare intera* si applicano ai problemi classici attraverso i loro modelli MIP.
- L'arrotondamento e il diving costruiscono una soluzione a partire dal LP; la feasibility pump cerca solo l'ammissibilità; RINS, local branching e proximity search migliorano una soluzione incumbent con un sotto-MIP.
- Relax-and-fix costruisce una soluzione lungo il tempo, fix-and-optimize la migliora finestra per finestra.
- Dentro un solver MIP il bound duale è disponibile: il gap della soluzione euristica è noto, e un gap uguale a 0 certifica l'ottimalità.

Quadro riassuntivo delle istanze di riferimento. Migliore soluzione trovata dai metodi di questo capitolo contro il bound duale (max per il KP, min per gli altri):

istanza	soluzione	bound duale	gap
KP	17	17	0%
BPP	3	3	0%
$P \mid\mid C_{\max}$	9	9	0%
lot sizing, capacitato	160	160	0%
BPP medio	23	23	0%

5.12 Implementazione in Python

- I codici dei metodi (arrotondamento, diving, feasibility pump, RINS, local branching, proximity search, relax-and-fix e fix-and-optimize) sono quelli del Capitolo 10 «Euristiche MIP» del modulo *Programmazione lineare intera*. Qui diamo i modelli dei problemi classici (`models.py`) e l'esperimento con i parametri di Gurobi (`gurobi_heuristics.py`); la cartella `MIP_HEUR_CODE` contiene tutte le funzioni e `instances.py`, con le istanze del capitolo e degli esercizi. Il comando `python3 run_tests.py` esegue ogni esempio del capitolo e ogni istanza degli esercizi, confrontando ogni valore con il valore atteso, scritto nel file.

```
import math
import gurobipy as gp
from gurobipy import GRB

EPS = 1e-6                                  # integrality and
    comparison tolerance

ENV = gp.Env(empty=True)                    # one silent, single-
    thread (deterministic) environment
ENV.setParam("OutputFlag", 0)
ENV.setParam("Threads", 1)
ENV.start()


def int_index(m):
    """positions j (in m.getVars()) of the integer and binary variables
       : the set I"""
    return [j for j, v in enumerate(m.getVars()) if v.VType != GRB.
       CONTINUOUS]


def nearest(v):
    """[v] = floor(v + 1/2): rounding to the nearest integer, 1/2
       rounded up"""
    return math.floor(v + 0.5 + EPS)


def is_integral(v):
```

```
24      return abs(v - round(v)) <= EPS
25
26
27  def solve_lp(m):
28      """optimal solution of the LP relaxation of m (with its current
           bounds), None if infeasible"""
29      m.update()
30      r = m.relax()
31      r.optimize()
32      if r.Status != GRB.OPTIMAL:
33          return None
34      return [v.X for v in r.getVars()]
35
36
37  def obj_of(m, x):
38      """value of the objective function of m at the point x"""
39      e = m.getObjective()
40      return e.getConstant() + sum(e.getCoeff(k) * x[e.getVar(k).index]
           for k in range(e.size()))
41
42
43  def round_point(x_tilde, I):
44      """x_bar = [x_tilde]: round the integer components, copy the
           continuous ones"""
45      x_bar = list(x_tilde)
46      for j in I:
47          x_bar[j] = nearest(x_tilde[j])
48      return x_bar
49
50
51  def evaluate(m, x_int):
52      """fix the integer variables to x_int and optimize the continuous
           ones:
53          returns (value, x), or None if the point cannot be completed to
               a feasible solution"""
54      f = m.copy()
55      V = f.getVars()
56      for j in int_index(f):
57          V[j].LB = V[j].UB = x_int[j]
58      f.optimize()
59      if f.Status != GRB.OPTIMAL:
60          return None
61      return f.ObjVal, [v.X for v in V]
62
63
64  def solve_mip(m):
65      """optimal value and solution of the MIP m, None if infeasible"""
66      m.optimize()
67      if m.SolCount == 0:
68          return None
69      return m.ObjVal, [v.X for v in m.getVars()]
70
71
```

```
def distance(m, x_bar, I):
    """the linear expression Delta(x, x_bar) = sum_{j in I} |x_j - x_bar_j| in the variables of m:
       x_j - l_j if x_bar_j is at its lower bound, u_j - x_j if at its upper bound,
       otherwise an auxiliary variable d_j >= |x_j - x_bar_j|"""
    V = m.getVars()
    expr = gp.LinExpr()
    for j in I:
        if x_bar[j] == V[j].LB:
            expr += V[j] - x_bar[j]
        elif x_bar[j] == V[j].UB:
            expr += x_bar[j] - V[j]
        else:
            d = m.addVar(name=f"d[{j}]")
            m.addConstr(d >= V[j] - x_bar[j])
            m.addConstr(d >= x_bar[j] - V[j])
            expr += d
    return expr


def delta(x, x_bar, I):
    """value of Delta(x, x_bar)"""
    return sum(abs(x[j] - x_bar[j]) for j in I)


def add_cutoff(m, z, theta):
    """require a solution better than z by at least theta"""
    obj = m.getObjective()
    if m.ModelSense == GRB.MINIMIZE:
        m.addConstr(obj <= z - theta, name="cutoff")
    else:
        m.addConstr(obj >= z + theta, name="cutoff")


def note(trace, *items):
    """record a step of the algorithm (used to print the examples of the chapter)"""
    if trace is not None:
        trace.append(items)
```

```
from itertools import combinations


def build_kp(p, w, c):
    m = gp.Model("KP", env=ENV)
    n = len(p)
    x = m.addVars(n, vtype=GRB.BINARY, name="x")
    m.setObjective(gp.quicksum(p[j] * x[j] for j in range(n)), GRB.MAXIMIZE)
    m.addConstr(gp.quicksum(w[j] * x[j] for j in range(n)) <= c, name="capacity")
    m.update()
```

```
    return m


def build_bpp(w, c, nb):
    m = gp.Model("BPP", env=ENV)
    n = len(w)
    x = m.addVars(n, nb, vtype=GRB.BINARY, name="x")
    y = m.addVars(nb, vtype=GRB.BINARY, name="y")
    m.setObjective(y.sum(), GRB.MINIMIZE)
    m.addConstrs((x.sum(j, "*") == 1 for j in range(n)), name="assign")
    m.addConstrs((-gp.quicksum(w[j] * x[j, b] for j in range(n)) + c *
        y[b] >= 0
                  for b in range(nb)), name="capacity")
    m.update()
    return m


def build_tsp(d):
    """DFJ model: one binary variable per arc (i,j), out- and in-degree
        , subtour elimination constraints"""
    m = gp.Model("TSP", env=ENV)
    n = len(d)
    A = [(i, j) for i in range(n) for j in range(n) if i != j]
    x = m.addVars(A, vtype=GRB.BINARY, name="x")
    m.setObjective(gp.quicksum(d[i][j] * x[i, j] for i, j in A), GRB.
        MINIMIZE)
    m.addConstrs((x.sum(i, "*") == 1 for i in range(n)), name="out")
    m.addConstrs((x.sum("*", j) == 1 for j in range(n)), name="in")
    for k in range(2, n):                          # S subset of the cities
         except the first one, |S| >= 2
        for S in combinations(range(1, n), k):
            m.addConstr(gp.quicksum(x[i, j] for i in S for j in S if i
                != j) <= len(S) - 1)
    m.update()
    return m


def build_pcm(t, m):
    model = gp.Model("PCmax", env=ENV)
    n = len(t)
    x = model.addVars(n, m, vtype=GRB.BINARY, name="x")
    Cmax = model.addVar(name="Cmax")
    model.setObjective(Cmax, GRB.MINIMIZE)
    model.addConstrs((x.sum(j, "*") == 1 for j in range(n)), name="
        assign")
    model.addConstrs((-gp.quicksum(t[j] * x[j, h] for j in range(n)) +
        Cmax >= 0
                      for h in range(m)), name="load")
    model.update()
    return model


def build_lot_sizing(d, sc, vc, hc, cap=None):
```

```
    """big-M model of lot sizing: production x_t, inventory s_t, setup
       y_t, M_t = min{cap_t, D_tm};
       cap = None: uncapacitated; x_t and s_t continuous (with y fixed,
           the LP has an integer optimum)"""
    m = gp.Model("LS", env=ENV)
    T = len(d)
    y = m.addVars(T, vtype=GRB.BINARY, name="y")
    x = m.addVars(T, name="x")
    s = m.addVars(T, name="s")
    m.setObjective(gp.quicksum(sc[t] * y[t] + vc[t] * x[t] + hc[t] * s[
       t] for t in range(T)), GRB.MINIMIZE)
    for t in range(T):
        m.addConstr((s[t - 1] if t > 0 else 0) + x[t] == d[t] + s[t],
           name=f"balance[{t}]")
        M = sum(d[t:]) if cap is None else min(cap[t], sum(d[t:]))
        m.addConstr(x[t] <= M * y[t], name=f"setup[{t}]")
        if cap is not None:
            m.addConstr(x[t] <= cap[t], name=f"capacity[{t}]")
    m.addConstr(s[T - 1] == 0, name="end")      # s_0 = s_m = 0
    m.update()
    return m


def build_ip2(A, b, c, U):
    """max c1 x1 + c2 x2 subject to A x <= b, 0 <= x <= U, x integer"""
    m = gp.Model("IP2", env=ENV)
    x = m.addVars(2, lb=0, ub=U, vtype=GRB.INTEGER, name="x")
    m.setObjective(c[0] * x[0] + c[1] * x[1], GRB.MAXIMIZE)
    for k in range(len(A)):
        m.addConstr(A[k][0] * x[0] + A[k][1] * x[1] <= b[k])
    m.update()
    return m


def pcm_solution(m, sets):
    """the solution of the P||Cmax model with job sets J_1, ..., J_m
       (1-based jobs)"""
    x = [0] * m.NumVars
    index = {v.VarName: k for k, v in enumerate(m.getVars())}
    for h, J in enumerate(sets):
        for j in J:
            x[index[f"x[{j - 1},{h}]"]] = 1
    return evaluate(m, x)[1]
```

```
def run_with_parameters(m, params, work_limit):
    """solve m with the given Gurobi parameters for at most work_limit
       work units;
       returns incumbent, dual bound, nodes, work and the list of the
          incumbents found (work, value)"""
    s = m.copy()
    s.Params.WorkLimit = work_limit
    for name, value in params.items():
        s.setParam(name, value)
```

```
    found = []
    def callback(model, where):                    # called by Gurobi at every new incumbent
        if where == GRB.Callback.MIPSOL:
            found.append((model.cbGet(GRB.Callback.WORK), model.cbGet(GRB.Callback.MIPSOL_OBJ)))
    s.optimize(callback)
    z = s.ObjVal if s.SolCount > 0 else None
    return z, s.ObjBound, int(s.NodeCount), s.Work, found
```