Euristiche costruttive (2/2)

Indice (1/2)

Il problema del commesso viaggiatore: accodare o inserire?
- Nearest neighbour
- Nearest neighbour multi-start
- Euristiche di inserimento
- Un confronto su un'istanza più grande

Colorazione dei vertici: greedy e DSATUR
- La colorazione greedy
- DSATUR
- Che cosa garantisce la colorazione greedy
- DSATUR come euristica

Costruzione randomizzata

Indice (2/2)

Look-ahead e regret
- Inserimento con regret
- Look-ahead: il metodo pilota

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

Errori tipici

Implementazione in C++

In sintesi

Dove eravamo

- Un'**euristica costruttiva** costruisce una soluzione ammissibile, un elemento alla volta; un algoritmo **greedy** aggiunge l'elemento che sembra migliore adesso, e una scelta, una volta fatta, non si rivede mai.

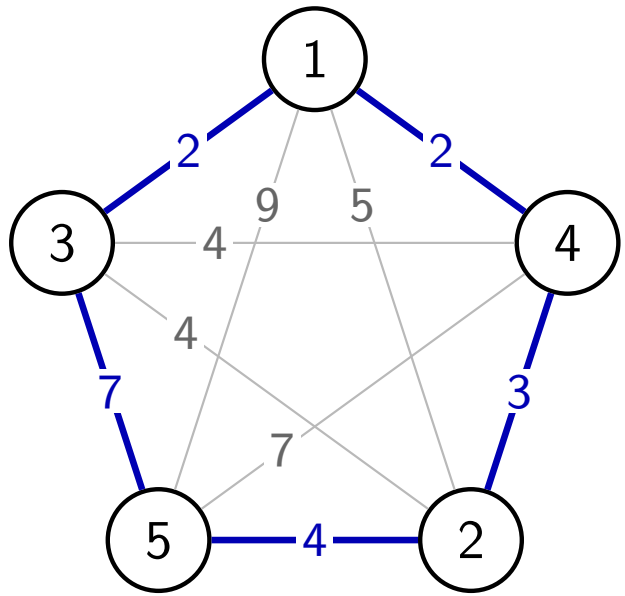
- Nella prima parte l'ordine degli elementi era la principale scelta di progetto: efficienza per il KP, pesi decrescenti per il BPP (FFD, BFD), prima i lavori più lunghi per $P \,||\, C_{\max}$ (LPT); nessun ordinamento è il migliore su ogni istanza.

- In questa seconda parte i criteri sono dinamici: giri per il TSP, colorazioni per il VCP; poi la costruzione randomizzata, il regret e il look-ahead.

Dove eravamo: l'istanza di riferimento del TSP

Le cinque città sono disegnate su un pentagono, nell'ordine $(1, 3, 5, 2, 4)$ del giro ottimo ($z(I) = 18$):

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	—



Il problema del commesso viaggiatore: accodare o inserire?

Introduzione: Il problema del commesso viaggiatore: accodare o inserire?

- In questa sezione costruiamo giri del TSP: ogni permutazione delle città è ammissibile, quindi tutto il progetto sta nel criterio di selezione
- Studiamo il nearest neighbour, che accoda la nuova città alla fine di un cammino, la sua versione multi-start, e le euristiche di inserimento, che la inseriscono in un sottogiro chiuso
- Le eseguiamo sull'istanza di riferimento e le confrontiamo su un'istanza più grande con 6 città

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.

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.

Algorithm: Nearest Neighbour

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

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

```
1  function NearestNeighbour(n, d, s)
2      σ[1] ← s                          // il cammino parte da s
3      visited := {s}
4      for len := 2 to n do
5          last := σ[len − 1]    // ultima città del cammino
6          k := la città j ∉ visited con d[last][j] minimo (indice
             minimo)
7          σ[len] ← k                    // accodare k alla fine
8          visited ← visited ∪ {k}
9      end
10     return σ // il giro si chiude con l'arco (σ[n], s)
11 end
```

Il criterio di *NN* e la sua debolezza

- 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;
  - l'arco di chiusura verso s non viene scelto affatto, è forzato.

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

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.

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

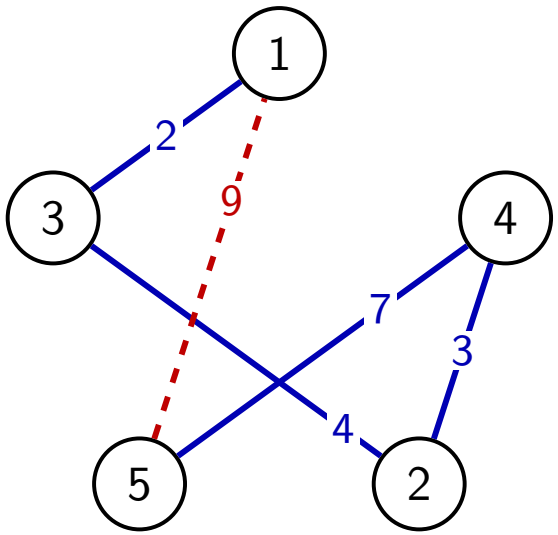
Quindi $\sigma = (1, 3, 2, 4, 5)$ e

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

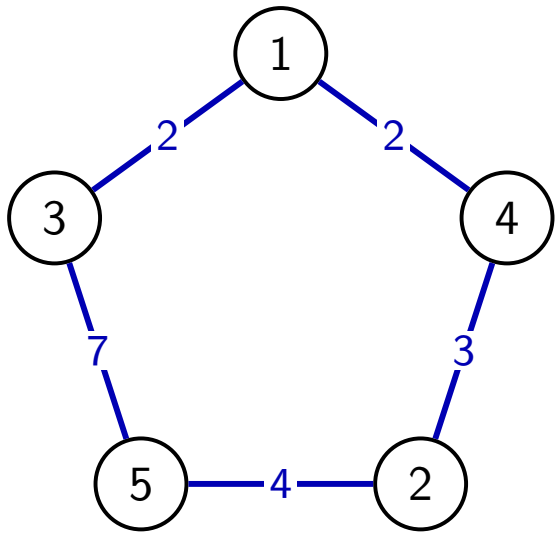
il 39% in più di $z(I) = 18$.

- 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.

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



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



giro ottimo: $L = 18$

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 $s \in \mathscr{V}$ e restituisce il più corto degli n giri ottenuti (in caso di pareggio, il primo trovato).

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
10 end
```

Da una scelta arbitraria a una ricerca

- 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: *NN* multi-start sull'istanza di riferimento del TSP (1/3)

Le cinque esecuzioni di *NN* danno

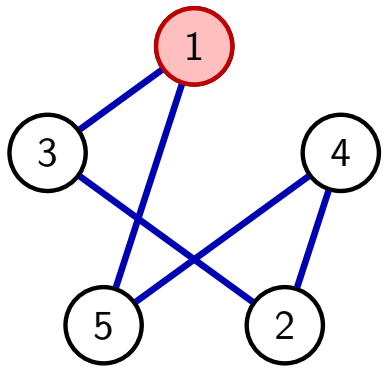
partenza s	1	2	3
giro	$(1,3,2,4,5)$	$(2,4,1,3,5)$	$(3,1,4,2,5)$
L	25	18	18

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

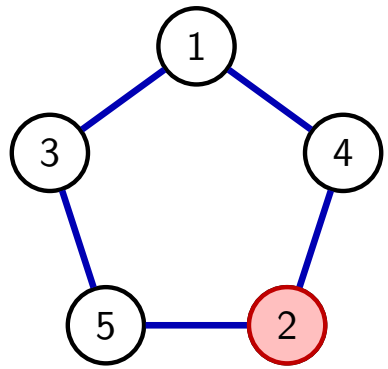
Esempio: *NN* multi-start sull'istanza di riferimento del TSP (2/3)

- 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 $18 = z(I)$.

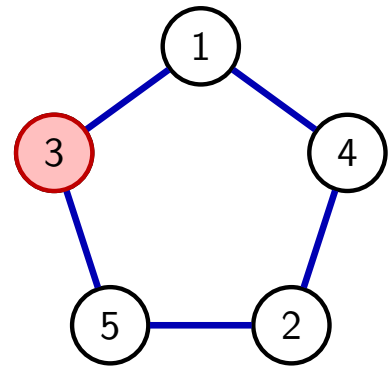
Esempio: *NN* multi-start sull'istanza di riferimento del TSP (3/3)



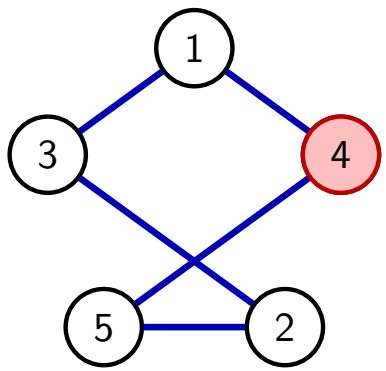
partenza 1: $L = 25$



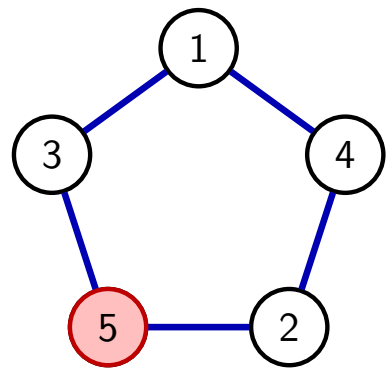
partenza 2: $L = 18$



partenza 3: $L = 18$



partenza 4: $L = 19$



partenza 5: $L = 18$

Il costo 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**

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

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

Quale città, e dove

- Ogni passo prende due decisioni:
  - **quale** città si inserisce (la regola di selezione);
  - **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

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 $\min_{(i,j)\in T} \Delta(i, q, j)$, il costo del suo inserimento

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

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
9 end
```

Inserimento: il primo passo, e accodare o inserire

- **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*, e i tempi di esecuzione

- **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: *NI* e *CI* sull'istanza di riferimento del TSP (1/4)

Partiamo da $T = (1)$. A ogni passo elenchiamo, per ogni città $q \notin T$, la sua distanza $d(q, T)$ (usata da *NI*) e la città inserita:

sottogiro T	$d(q, T)$	inserita
(1)	$2\colon 5,\ 3\colon 2,\ 4\colon 2,\ 5\colon 9$	3 su $(1, 1)$, costo 4
$(1, 3)$	$2\colon 4,\ 4\colon 2,\ 5\colon 7$	4 su $(1, 3)$, costo 4
$(1, 4, 3)$	$2\colon 3,\ 5\colon 7$	2 su $(4, 3)$, costo 3
$(1, 4, 2, 3)$	$5\colon 4$	5 su $(2, 3)$, costo 7

Esempio: *NI* e *CI* sull'istanza di riferimento del TSP (2/4)

A ogni passo elenchiamo, per ogni città $q \notin T$, i suoi costi di inserimento sugli archi di T, nell'ordine di T (usati da *CI*), e la città inserita:

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

Esempio: *NI* e *CI* sull'istanza di riferimento del TSP (3/4)

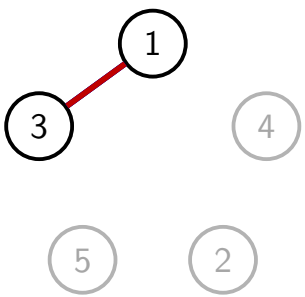
- *NI* seleziona la città con la $d(q, T)$ minima, *CI* la città con il costo di inserimento minimo.
- Su questa istanza le due regole selezionano le stesse città a ogni passo (la città 3 al primo passo per il pareggio con la città 4).
- Entrambe restituiscono $\sigma = (1, 4, 2, 5, 3)$, e

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

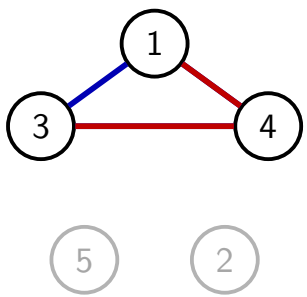
il giro ottimo $(1, 3, 5, 2, 4)$ percorso nel verso opposto.

Esempio: *NI* e *CI* sull'istanza di riferimento del TSP (4/4)

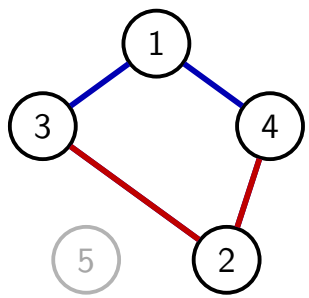
I sottogiri crescono come segue (nuovi archi in rosso):



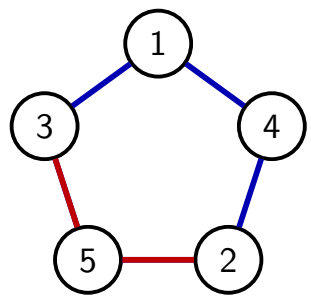
$L = 4$



$L = 8$



$L = 11$



$L = 18$

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

Partiamo da $T = (1)$:

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

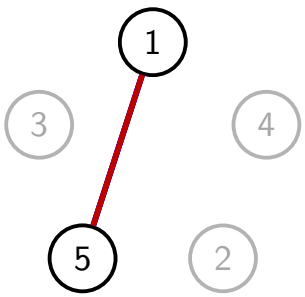
Esempio: *FI* sull'istanza di riferimento del TSP (2/3)

- 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

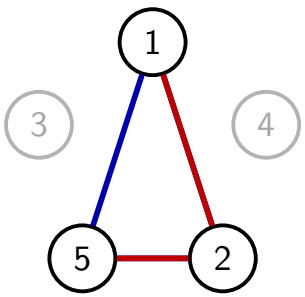
$$d_{12} + d_{25} = d_{15}, \qquad d_{53} + d_{31} = d_{51}, \qquad d_{14} + d_{42} = d_{12}.$$

- 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.

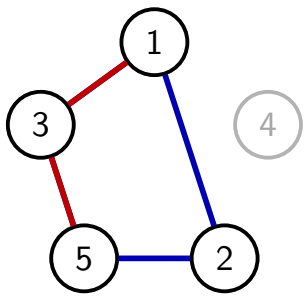
Esempio: *FI* sull'istanza di riferimento del TSP (3/3)



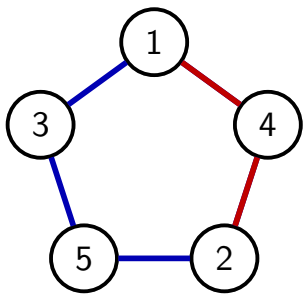
$L = 18$



$L = 18$



$L = 18$



$L = 18$

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.
- 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.

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: un'istanza del TSP con 6 città (1/4)

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	—

Esempio: un'istanza del TSP con 6 città (2/4)

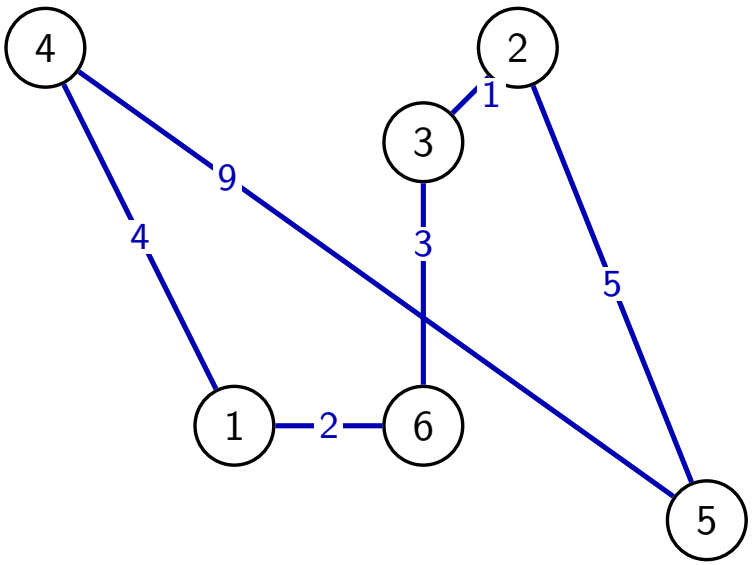
- L'istanza ha $\frac{5!}{2} = 60$ giri distinti; enumerandoli, l'unico giro ottimo è $(1, 4, 3, 2, 5, 6)$, con $z(I) = 19$, e solo 7 giri hanno lunghezza minore di 24.
- Le euristiche di questa sezione danno:

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

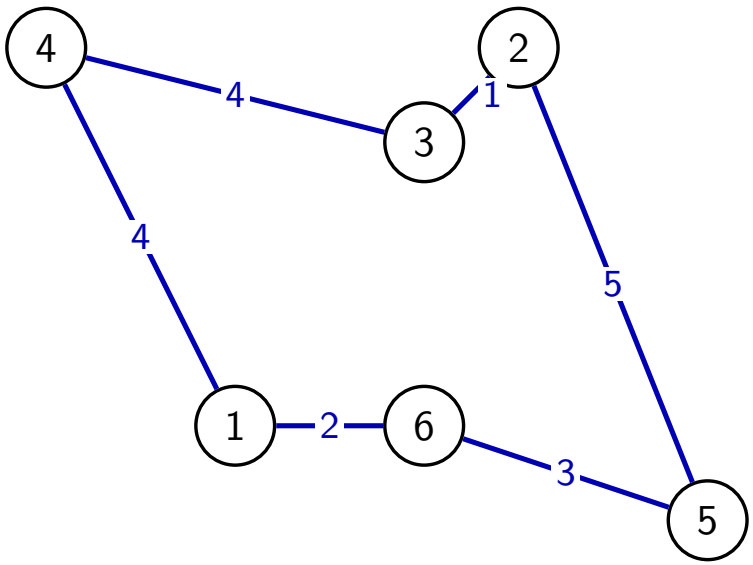
dalla città 1	L
NI	23
FI	19
CI	20

Esempio: un'istanza del TSP con 6 città (3/4)

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: $L = 19 = z(I)$

Esempio: un'istanza del TSP con 6 città (4/4)

- 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.

Riepilogo

Accodare: nearest neighbour:

- *NN* accoda la città non visitata più vicina; l'arco di chiusura è forzato: $NN(I) = 25$ dalla città 1
- il *NN* multi-start esegue *NN* da ogni città e tiene il giro migliore: $18 = z(I)$

Inserire: euristiche di inserimento:

- un sottogiro chiuso, la città inserita nell'arco di $\Delta(i,k,j) = d_{ik} + d_{kj} - d_{ij}$ minimo
- *NI*, *FI*, *CI* differiscono solo per la città selezionata; tutte danno $18 = z(I)$ sull'istanza di riferimento
- sull'istanza con 6 città: *NN* multi-start 24, *NI* 23, *CI* 20, *FI* $19 = z(I)$

Colorazione dei vertici: greedy e DSATUR

Introduzione: Colorazione dei vertici: greedy e DSATUR

- In questa sezione coloriamo i vertici di un grafo uno alla volta: la colorazione greedy segue una lista data, *DSATUR* costruisce la lista al volo
- Le eseguiamo entrambe sul grafo di Petersen, e vediamo che l'ordine della lista può fare una grande differenza
- Dimostriamo che cosa garantisce la colorazione greedy, e che cosa non garantisce

Il problema della colorazione dei vertici

- 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 $\chi(G)$.

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.

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

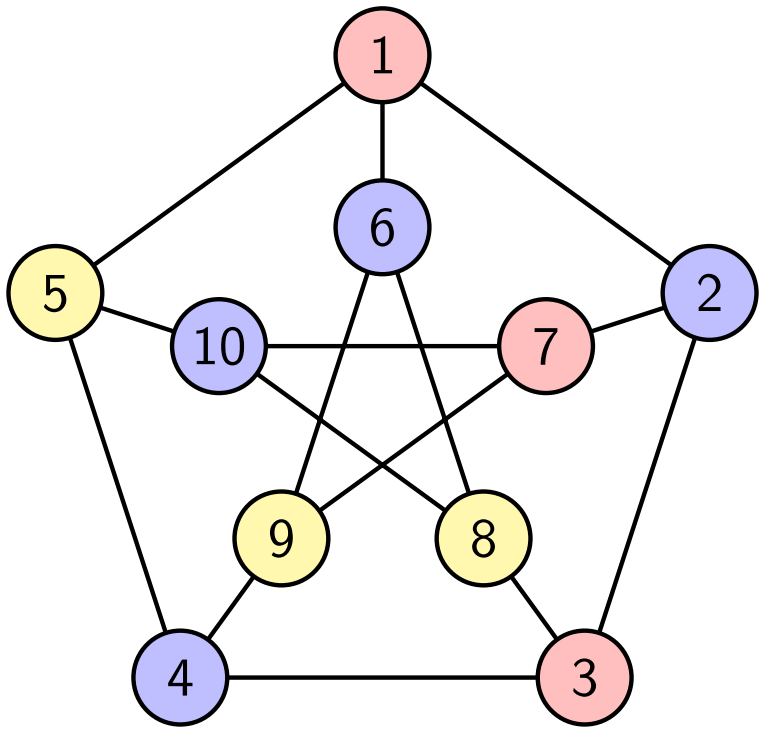
```
1 function GreedyColoring(n, E, λ)
2     col := nuovo vettore[1..n], con tutte le componenti 0
       // 0 = non ancora colorato
3     for k := 1 to n do
4         v := λ[k]
5         U := { col[u] : u ∈ N(v), col[u] ≠ 0 }      // colori
           dei vicini colorati
6         col[v] ← min{ c ∈ Z_{≥1} : c ∉ U }  // colore libero
           minimo
7     end
8     return col, nc = max_v col[v]
9 end
```

La colorazione greedy è ammissibile, e dipende dalla lista

- 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: *GC* sul grafo di Petersen, due ordini (1/3)

Il grafo di Petersen, con i colori dati da *GC* con la lista $\lambda = (1, 2, \ldots, 10)$:



Esempio: *GC* sul grafo di Petersen, due ordini (2/3)

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.

Esempio: *GC* sul grafo di Petersen, due ordini (3/3)

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

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

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.

Algorithm: DSATUR (1/2)

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

```
1 function DSATUR(n, E)
2 |   col := nuovo vettore[1..n], con tutte le componenti 0
3 |   for k := 1 to n do
4 |   |   for ogni u non colorato do
5 |   |   |   sat[u] ← |{ col[w] : w ∈ N(u), col[w] ≠ 0 }|
  |   |   |     // grado di saturazione
6 |   |   |   du[u] ← |{ w ∈ N(u) : col[w] = 0 }|
  |   |   |     // grado fra i non colorati
7 |   |   end
```

Algorithm: DSATUR (2/2)

function *DSATUR*(n, E) (continua)
    for $k := 1$ **to** n **do** (continua)
8       $v :=$ il vertice non colorato che massimizza $(sat[v], du[v])$ in ordine lessicografico (indice minimo)
9       $col[v] \leftarrow \min\{\, c \in \mathbb{Z}_{\geq 1} : c \neq col[w]\ \forall w \in N(v) \,\}$
10   **end**
11   **return** col, $nc = \max_v col[v]$
12 **end**

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: *DSATUR* sul grafo di Petersen (1/2)

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

Esempio: *DSATUR* sul grafo di Petersen (2/2)

- 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 $(1, 2, \ldots, 10)$.

Un upper bound sui colori di GC

Quanti colori può usare GC, qualunque sia la lista?

Proposizione 1

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

$$|U| \leq \deg(v) \leq \Delta(G)$$

elementi, quindi uno dei colori $1, 2, \ldots, \Delta(G) + 1$ non è in U, e il colore libero minimo è al più $\Delta(G) + 1$. □

Che cosa dice il bound (1/2)

- Il bound vale anche per *DSATUR*, che è *GC* con una lista particolare.
- 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.

Che cosa dice il bound (2/2)

- 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.

Una cattiva lista

Una cattiva lista può imporre molti colori a un grafo che ne richiede solo due:

Proposizione 2

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 (1/2)

- 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

$$\lambda = (u_1, v_1, u_2, v_2, \ldots, u_k, v_k).$$

Dimostrazione (2/2)

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 $GC(I_k) = k$. □

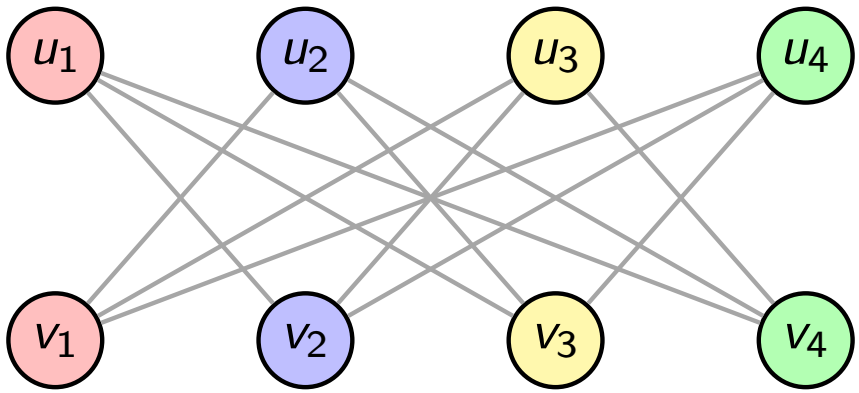
Nessuna garanzia costante per GC

- 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: *GC* e *DSATUR* sul grafo corona G_4 (1/2)

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.

Esempio: *GC* e *DSATUR* sul grafo corona G_4 (2/2)

- *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.

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*.

Riepilogo

Colorazione greedy:

- ogni vertice della lista riceve il colore minimo non usato dai suoi vicini colorati
- sempre ammissibile, tempo $O(n + |E|)$; $GC(I) \leq \Delta(G) + 1$ per ogni lista
- una cattiva lista può imporre $\frac{n}{2}$ colori a un grafo bipartito (grafi corona); qualche lista è sempre ottima

DSATUR:

- colora prima il vertice con più colori diversi intorno: prima il più vincolato
- $3 = \chi(G)$ colori sul grafo di Petersen, 2 colori sui grafi bipartiti
- nessuna garanzia di prestazione, ma una delle euristiche più usate per il VCP

Costruzione randomizzata

Introduzione: Costruzione randomizzata

- In questa sezione otteniamo da un'euristica costruttiva molte soluzioni diverse, e comunque buone, randomizzando le sue scelte
- Definiamo la lista ristretta dei candidati (RCL) e un generatore riproducibile di numeri casuali
- Eseguiamo quattro volte il nearest neighbour randomizzato sull'istanza di riferimento del TSP

Randomizzare la scelta

- 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*.

La lista ristretta dei candidati

- 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

$$RCL = \left\{ q : g(q) \leq g^{\min} + \alpha \left(g^{\max} - g^{\min} \right) \right\}.$$

- 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.

RCL basata sulla cardinalità e GRASP

- 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:

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

- 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.

Estrarre un candidato dalla RCL

- 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 $|RCL| \geq 2$.

Nearest neighbour randomizzato

- 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.

Algorithm: Nearest Neighbour randomizzato (1/2)

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

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

```
1 function RandomizedNN(n, d, s, α, seed)
2 |   σ[1] ← s, visited := {s}
3 |   for len := 2 to n do
4 |   |   last := σ[len − 1]
5 |   |   d^min := min_{j∉visited} d[last][j],
  |   |     d^max := max_{j∉visited} d[last][j]
6 |   |   RCL := { j ∉ visited : d[last][j] ≤
  |   |     d^min + α (d^max − d^min) }          // per indice
```

Algorithm: Nearest Neighbour randomizzato (2/2)

function *RandomizedNN*$(n, d, s, \alpha, seed)$ (continua)

    for $len := 2$ **to** n **do** (continua)
7   **if** $|RCL| \geq 2$ **then**
8           $u :=$ ParkMiller($seed$) `// numero casuale in` $(0, 1)$
9           $k :=$ l'elemento in posizione $\lfloor u \cdot |RCL| \rfloor + 1$ di RCL
10   **else**
11          $k :=$ l'unico elemento di RCL
12   **end**
13      $\sigma[len] \leftarrow k$, $visited \leftarrow visited \cup \{k\}$
14 **end**
15 **return** σ
16 **end**

Qualità e varietà

- 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: $RNN_{0.3}$ sull'istanza di riferimento del TSP (1/5)

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
0.0000	0.1315	0.7556	0.4587	0.5328

u_6	u_7	u_8	u_9
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 $RCL = \{3, 4\}$.

Esempio: $RNN_{0.3}$ sull'istanza di riferimento del TSP (2/5)

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.

Esempio: $RNN_{0.3}$ sull'istanza di riferimento del TSP (3/5)

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 $(1, 4, 2, 5, 3)$, $L = 18 = z(I)$.

Esempio: $RNN_{0.3}$ sull'istanza di riferimento del TSP (4/5)

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 $(1, 4, 3, 2, 5)$, $L = 2 + 4 + 4 + 4 + 9 = 23$.

Esempio: $RNN_{0.3}$ sull'istanza di riferimento del TSP (5/5)

- 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.

Riepilogo

Costruzione randomizzata:

- a ogni passo, un candidato estratto a caso dalla lista ristretta dei candidati (RCL)
- RCL basata sul valore: $g(q) \leq g^{\min} + \alpha\,(g^{\max} - g^{\min})$; $\alpha = 0$ greedy, $\alpha = 1$ puramente casuale
- il primo ingrediente di GRASP

Numeri casuali riproducibili:

- il generatore di Park e Miller: lo stesso seme dà la stessa successione ovunque
- $RNN_{0.3}$ sull'istanza di riferimento: $25, 18, 25, 23$, e l'ottimo 18 viene trovato

Look-ahead e regret

Introduzione: Look-ahead e regret

- In questa sezione riduciamo la miopia del criterio greedy, continuando a prendere ogni scelta una volta per tutte
- L'inserimento con regret serve prima la città che perderebbe di più se fosse rimandata
- Il metodo pilota valuta ogni candidato completando la soluzione con un'euristica veloce

Due domande diverse

- 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.

Il regret di una città

- 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.

Inserimento regret-2

- 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* nel suo arco di costo di inserimento minimo.
- In caso di pareggio, inserisce la città con Δ_q^1 minimo, e poi quella con l'indice minimo.

Algorithm: Inserimento regret-2 (1/2)

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

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

```
1 function Regret2Insertion(n, d, s)
2 |   σ := (s)
3 |   for len := 1 to n − 1 do
4 |   |   foreach città q ∉ σ do
5 |   |   |   Δ¹_q, Δ²_q := i due valori minimi di
  |   |   |    Δ(σ[a], q, σ[a + 1]), a = 1, . . . , len
6 |   |   |   reg_q := Δ²_q − Δ¹_q          // regret di q
7 |   |   end
```

Algorithm: Inserimento regret-2 (2/2)

function *Regret2Insertion*(n, d, s) (continua)

    for $len := 1$ **to** $n - 1$ **do** (continua)
8       $k :=$ la città con reg_q massimo (pareggi: Δ_q^1 minimo, poi indice minimo)
9   inserire k nel suo arco di costo di inserimento minimo
10 **end**
11 **return** σ
12 **end**

Euristiche regret-k

- 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: REG_2 contro CI sull'istanza con 6 città (1/4)

Sull'istanza con 6 città della sezione precedente, 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);
- 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

Esempio: REG_2 contro CI sull'istanza con 6 città (2/4)

- 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.

Esempio: REG_2 contro CI sull'istanza con 6 città (3/4)

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	$4+5+2+4+5=20$	$4+5+4+5+1=19$

Esempio: REG_2 contro CI sull'istanza con 6 città (4/4)

- 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 $CI(I) = 20$.

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-NN*), i candidati sono le città successive del giro.

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
11 end
```

Il costo di guardare avanti

- 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: *PILOT-NN* sull'istanza di riferimento del TSP (1/2)

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

Esempio: *PILOT-NN* sull'istanza di riferimento del TSP (2/2)

- *PILOT-NN* va prima nella città 5, la *più lontana* dalla città 1, cosa che *NN* non farebbe mai.
- 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 della sezione precedente, *PILOT-NN* dalla città 1 dà 20, contro 25 per *NN*.

Riepilogo

Regret:

- $reg_q = \Delta_q^2 - \Delta_q^1$: che cosa si perde se l'inserimento di q è rimandato
- REG_2 inserisce prima la città con il regret massimo; sull'istanza con 6 città $REG_2(I) = 19 = z(I)$, $CI(I) = 20$

Look-ahead:

- il metodo pilota completa ogni candidato con un'euristica veloce e tiene il migliore
- *PILOT-NN* non è mai peggiore di *NN*: 22 contro 25 sull'istanza di riferimento
- entrambe le idee costano più tempo per passo, e le scelte restano definitive

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

Introduzione: Che cosa danno le euristiche costruttive, e che cosa non danno

- In questa sezione raccogliamo i risultati sulle quattro istanze di riferimento e li confrontiamo con i bound duali
- Traiamo le lezioni del capitolo: un bound primale, scelte definitive, l'ordine delle scelte
- Vediamo perché le euristiche costruttive si usano raramente da sole

Risultati sulle istanze di riferimento

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	$\mathrm{UB} = 17$	0%	17
BPP	FF, FFD, BFD	4, 3, 3	$\mathrm{LB} = 3$	0%	3
$P \mid\mid C_{\max}$	*LIST* (lista inversa), *LPT*	12, 11	$\mathrm{LB} = 9$	22.2%	9
TSP	*NN*, multi-start *NN*, *NI*, *FI*, *CI*	25, 18, 18, 18, 18	$\mathrm{LB} = 17$	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 questo capitolo 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 questo capitolo 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 (i tre ordinamenti sull'istanza di riferimento del KP, e l'esempio in cui l'ordinamento peggiora FFD).
- 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.

Riepilogo

Che cosa danno:

- una soluzione ammissibile, cioè un bound primale: l'ottimalità è dimostrata solo se un bound duale lo raggiunge
- soluzioni veloci, usate come punti di partenza dei metodi di miglioramento

Che cosa non danno:

- un modo per tornare indietro: ogni scelta è definitiva
- un ordine che sia il migliore su ogni istanza: più ordini, partenze o costruzioni randomizzate proteggono da questo

Errori tipici

Introduzione: Errori tipici

- Raccogliamo gli errori che si commettono più spesso con le euristiche costruttive
- Due riguardano gli ordinamenti, due l'esecuzione a mano, uno il significato del valore
- Controllarli prima di eseguire a mano un'euristica

Errori tipici (1/2)

Osservazione 1: Aspettarsi che il greedy per efficienza sia ottimo

Può essere arbitrariamente cattivo, e così ciascuno degli altri due ordinamenti: il capitolo contiene una famiglia di istanze per ognuno di essi.

Osservazione 2: Credere che ordinare aiuti sempre

FFD può essere peggiore di FF (l'esempio in cui l'ordinamento peggiora).

Errori tipici (2/2)

Osservazione 3: 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.

Osservazione 4: 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.

Osservazione 5: Leggere $\mathscr{A}(I)$ come un bound su $z(I)$

È solo un bound primale.

Riepilogo

Gli ordinamenti:

- nessun ordinamento degli oggetti del KP è sempre ottimo; ordinare non aiuta sempre

L'esecuzione a mano:

- risolvere i pareggi con la regola fissata dell'euristica
- contare l'arco di chiusura, e provare ogni posizione di un inserimento

Il valore:

- $\mathscr{A}(I)$ è solo un bound primale

Implementazione in C++

Introduzione: Implementazione in C++

- In questa sezione mostriamo alcuni brevi estratti dell'implementazione in C++ delle euristiche del capitolo
- Il codice completo è nella cartella del capitolo e nei notebook
- Il codice segue gli pseudocodici passo per passo, con gli stessi nomi

Il codice del capitolo

- La cartella `CONSTR_HEUR_CODE` contiene un'implementazione in C++ delle euristiche del capitolo, 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 delle 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 (estratto)

L'header `Constr_heuristics.h` dichiara tutte le funzioni; per il TSP:

```
// TSP: d is the n x n distance matrix;
// every function returns the length of the tour
int TourLength(int n, int** d, int* tour);
int NearestNeighbour(int n, int** d, int s,
                     int* tour);
int MultiStartNN(int n, int** d, int* tour);
int Insertion(int n, int** d, int s, int rule,
              int* tour);  // 0 NI, 1 FI, 2 CI
int Regret2Insertion(int n, int** d, int s,
                     int* tour);
int PilotNN(int n, int** d, int s, int* tour);
```

Ordinamento (estratto)

- Gli ordinamenti degli oggetti del KP 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 static bool Before(int i, int j, int* p, int* w,
2                    int criterion)
3 {
4     if (criterion == 0) return p[i] > p[j];
5     if (criterion == 1) return w[i] < w[j];
6     // p_i/w_i > p_j/w_j without divisions
7     return p[i] * w[j] > p[j] * w[i];
8 }
```

Nearest neighbour (estratto)

Il ciclo principale di `NearestNeighbour`:

```
for (int len = 1; len < n; len++)
{
    int last = tour[len - 1];  // last city
    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;
}
```

Inserimento (estratto)

- 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 key = d[q][tour[0]];    // distance from the subtour
2 for (int a = 1; a < len; a++)
3 {
4     if (d[q][tour[a]] < key) key = d[q][tour[a]];
5 }
6 if (rule == 1) key = -key;  // farthest: maximize
```

Costruzione randomizzata (estratto)

- 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.

```
1 // d <= dmin + alpha (dmax - dmin),
2 // in integer arithmetic
3 if (!visited[j] && alpha_den * d[last][j]
4         <= alpha_den * dmin
5             + alpha_num * (dmax - dmin))
6 {
7     RCL[size] = j;
8     size++;
9 }
```

Riepilogo

Il codice:

- una funzione `C++` per euristica, con i nomi e i passi degli pseudocodici
- indici da 0, pareggi risolti a favore dell'indice minimo
- `make test`: 202 controlli contro un'implementazione indipendente in Python, 0 errori

Dettagli di implementazione:

- l'insertion sort mantiene l'ordine degli indici fra chiavi uguali; efficienze confrontate senza divisioni
- un solo ciclo per *NI*, *FI* e *CI*; la soglia della RCL in aritmetica intera

In sintesi

Introduzione: In sintesi

- Raccogliamo prima le idee chiave di questa seconda parte, poi quelle dell'intero capitolo sulle euristiche costruttive
- Il quadro riassume i migliori valori costruttivi sulle quattro istanze di riferimento, con i loro bound duali

Riepilogo della seconda parte (1/2)

Il problema del commesso viaggiatore:

- accodare: *NN* e la sua versione multi-start; inserire: *NI*, *FI*, *CI*, con il costo di inserimento $\Delta(i, k, j)$
- sull'istanza con 6 città *FI* trova l'ottimo 19, il *NN* multi-start solo 24

Colorazione dei vertici:

- *GC* segue una lista, $GC(I) \leq \Delta(G) + 1$, ma una cattiva lista può essere arbitrariamente cattiva (grafi corona)
- *DSATUR* colora prima il vertice più vincolato

Riepilogo della seconda parte (2/2)

Costruzione randomizzata:

- una scelta casuale nella lista ristretta dei candidati trasforma una costruzione in una piccola ricerca

Regret e look-ahead:

- REG_2 inserisce prima la città che perderebbe di più se fosse rimandata
- il metodo pilota completa ogni candidato con un'euristica veloce, e non è mai peggiore del suo pilota

Che cosa danno le euristiche costruttive:

- un bound primale, veloce, con scelte definitive: il punto di partenza dei metodi di miglioramento

Riepilogo (1/2)

- 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.

Riepilogo (2/2)

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
$P \mid\mid C_{\max}$ (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