Algoritmi Euristici

Corso di Laurea in Informatica e Corso di Laurea in Matematica

Roberto Cordone

DI - Università degli Studi di Milano

Lezioni:	Lunedì 13.30 - 15.30 in Aula G30
	Giovedì 13.30 - 15.30 in Aula G30
Ricevimento:	su appuntamento
Tel.:	02 503 16235
E-mail:	roberto.cordone@unimi.it
Web page:	http://homes.di.unimi.it/~cordone/courses/2018-ae/2018-ae.html

Gli algoritmi costruttivi

In Ottimizzazione Combinatoria ogni soluzione x è un sottoinsieme di B

Un'euristica costruttiva aggiorna passo per passo un sottoinsieme $x^{(t)}$

1. parte da un sottoinsieme vuoto: $x^{(0)} = \emptyset$
   (*è ovvio che sia sottoinsieme di una soluzione ottima*)
2. si ferma se vale una condizione di fine opportunamente definita
   (*i sottoinsiemi successivi non possono essere soluzioni ottime*)
3. ad ogni passo t, sceglie l'elemento $i^{(t)} \in B$ "migliore" fra quelli "ammissibili" in base a un opportuno criterio di scelta
   (*si cerca di tenere $x^{(t)}$ dentro una soluzione ammissibile e ottima*)
4. aggiunge $i^{(t)}$ al sottoinsieme corrente $x^{(t)}$: $x^{(t+1)} := x^{(t)} \cup \{i^{(t)}\}$
   (*non si torna più indietro nella scelta!*)
5. torna al punto 2

Lo spazio di ricerca

Lo spazio di ricerca $\mathcal{F}_A$ di un algoritmo costruttivo A è la collezione dei sottoinsiemi che l'algoritmo considera validi e include

- il sottoinsieme vuoto: $\emptyset \in \mathcal{F}_A$ (*per poter cominciare*)
- alcune soluzioni parziali (sottoinsiemi di soluzioni ammissibili) (*per attraversarle e raggiungere le soluzioni ammissibili*)
- le soluzioni ammissibili promettenti (*quelle che non sono ovviamente dominate*)

Esempi:

- *KP*: sottoinsiemi di peso ammissibile, cioè soluzioni ammissibili ($\mathcal{F}_A = X$)
- *MDP*: sottoinsiemi di al più k punti, cioè soluzioni ammissibili e parziali ($\mathcal{F}_A = \bigcup_{x \in X} 2^x$)
- *SCP*: sottoinsiemi di colonne non ridondanti, cioè soluzioni parziali e soluzioni ammissibili non banalmente dominate (*definizione non del tutto banale*)
- *TSP*: sottoinsiemi di archi non contenenti sottocicli, ramificazioni, ... cioè soluzioni parziali, ammissibili e altri sottoinsiemi (*definizione insufficiente a garantire l'ammissibilità*)

Il grafo di costruzione

Il grafo di costruzione di un algoritmo costruttivo A ammette

- come nodi i sottoinsiemi validi

$$x \in \mathcal{F}_A$$

- come archi le coppie di sottoinsiemi validi in cui il secondo ha un elemento in più del primo

$$(x, x \cup \{i\}) : x \in \mathcal{F}_A, i \in B \setminus x \text{ e } x \cup \{i\} \in \mathcal{F}_A$$

L'algoritmo A indica un insieme di estensioni ammissibili

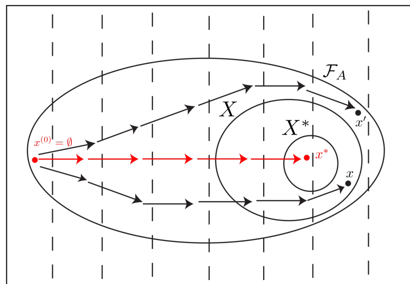
$$\mathrm{Ext}_A(x) = \{i \in B \setminus x : x \cup \{i\} \in \mathcal{F}_A\} \text{ per ogni } x \in \mathcal{F}_A$$

Il grafo di costruzione è ovviamente aciclico

Ogni cammino massimale descrive una possibile esecuzione di A

- parte dal sottoinsieme vuoto $\emptyset$
- termina in un sottoinsieme privo di estensioni ammissibili, che spesso è una soluzione ammissibile (non sempre!)

Un grafico

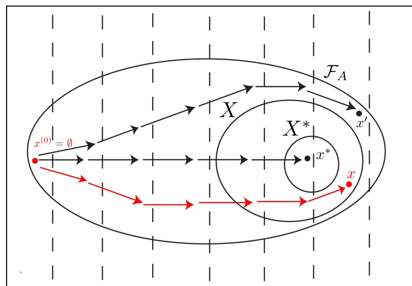


L'algoritmo visita una catena di sottoinsiemi $\emptyset = x^{(0)} \subset \ldots \subset x^{(k)}$ e termina

- in una soluzione ottima $x^* \in X^*$
- in una soluzione ammissibile non ottima $x \in X$
- in un sottoinsieme non ammissibile x'

Esempio: *MSTP* (sia con $\mathcal{F}_{\text{Prim}}$ sia con $\mathcal{F}_{\text{Kruskal}}$)

Un grafico

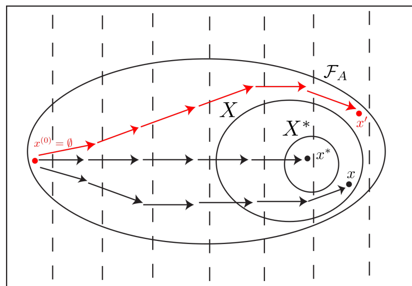


L'algoritmo visita una catena di sottoinsiemi $\emptyset = x^{(0)} \subset \ldots \subset x^{(k)}$ e terminae termina

- in una soluzione ottima $x^* \in X^*$
- in una soluzione ammissibile non ottima $x \in X$
- in un sottoinsieme non ammissibile x'

Esempio: *KP*, *MDP*, ecc...

Un grafico



L'algoritmo visita una catena di sottoinsiemi $\emptyset = x^{(0)} \subset \ldots \subset x^{(k)}$ e termina

- in una soluzione ottima $x^* \in X^*$
- in una soluzione ammissibile non ottima $x \in X$
- in un sottoinsieme non ammissibile x'

Esempio: *SPP* e *TSP* su grafo non completo

Test di appartenenza

Occorre definire $\mathcal{F}_A$ in modo che

1. il test di appartenenza $x^{(t)} \in \mathcal{F}_A$ sia efficiente

   Va ripetuto ad ogni passo dell'algoritmo

2. $\mathcal{F}_A$ includa (per quanto possibile) solo sottoinsiemi di soluzioni ammissibili, e magari ottime

   Così l'algoritmo A terminerebbe in esse, senza sviarsi

Purtroppo, può essere difficile stabilire per un problema dato

- se esistono soluzioni ammissibili
- se un sottoinsieme $x^{(t)}$ fa parte di una soluzione ammissibile $x \in X$
- se un sottoinsieme $x^{(t)}$ fa parte di una soluzione ottima $x^* \in X^*$

Condizione di termine

Un'euristica costruttiva A termina quando aggiungere qualsiasi elemento i al sottoinsieme corrente $x^{(t)}$ lo fa uscire dallo spazio di ricerca $\mathcal{F}_A$

$$x^{(t)} \cup \{i\} \notin \mathcal{F}_A \text{ per ogni } i \in B \setminus x^{(t)} \Rightarrow \text{STOP}$$

Dato che $\mathrm{Ext}_A(x) = \{i \in B \setminus x : x \cup \{i\} \in \mathcal{F}_A\}$, la condizione diventa

$$\mathrm{Ext}_A\left(x^{(t)}\right) = \emptyset \Rightarrow \text{STOP}$$

I possibili andamenti di $x^{(t)}$ sono molto vari

- a volte tutti i sottoinsiemi visitati sono ammissibili (per es., *KP*)
- spesso solo l'ultimo sottoinsieme è una soluzione ammissibile
- in generale, $x^{(t)}$ può entrare e uscire più volte da X e da X^*

La soluzione restituita è la migliore visitata durante l'esecuzione

(*spesso è l'ultima*)

Pseudocodice

Un'euristica costruttiva (problema di minimo) si può descrivere come

Algorithm Greedy(I)
$x := \emptyset$;
$x^* := \emptyset$; $f^* := +\infty$; { Miglior soluzione trovata sinora }
While $\mathrm{Ext}_A(x) \neq \emptyset$ *do*
  $i^* := \arg \min_{i \in \mathrm{Ext}_A(x)} \varphi_A(i, x)$;
  $x := x \cup \{i^*\}$;
  If $x \in X$ and $f(x) < f^*$ *then* $x^* := x$; $f^* := f(x)$;
Return (x^*, f^*);

La sequenza dei sottoinsiemi visitati dall'algoritmo dipende da

- l'insieme $\mathrm{Ext}_A(x)$, cioè equivalentemente lo spazio di ricerca $\mathcal{F}_A$
- il criterio di scelta $\varphi_A : B \times \mathcal{F} \to \mathbb{R}$ usato per scegliere l'elemento i con cui estendere il sottoinsieme corrente $x^{(t)}$ generando $x^{(t+1)}$

Efficacia ed efficienza

Un algoritmo costruttivo A trova l'ottimo quando ad ogni passo t il sottoinsieme corrente $x^{(t)}$ è contenuto in almeno una soluzione ottima

L'algoritmo non si taglia mai tutte le strade verso l'ottimo

Questa proprietà vale in $x^{(0)} = \emptyset$, ma di solito si perde in qualche passo t

Un'euristica costruttiva esegue al massimo $n = |B|$ passi

La complessità di ogni passo è data da

1. la costruzione di $\mathrm{Ext}_A(x)$, in tempo $T_{\mathrm{Ext}_A}(n)$
2. la valutazione di $\varphi_A(i, x)$ per ogni $i \in \mathrm{Ext}_A(x)$, in tempo $T_{\varphi_A}(n)$
3. l'estrazione del valore minimo e del corrispondente i
4. l'aggiornamento di x (ed eventuali altre strutture dati)

In generale, è una complessità polinomiale di ordine piuttosto basso, in cui prevalgono le prime due componenti

$$T_A(n) \in O\left(n\left(T_{\mathrm{Ext}_A}(n) + T_{\varphi_A}(n)\right)\right)$$

Caratteristiche generali

Le euristiche costruttive

1. sono intuitive
2. sono semplici da progettare, analizzare, realizzare
3. sono molto efficienti
4. hanno un'efficacia molto variabile
   - su alcuni problemi garantiscono una soluzione ottima
   - su altri problemi forniscono una garanzia di approssimazione
   - sulla maggior parte dei problemi forniscono soluzioni di qualità estremamente variabile, spesso scarsa
   - su alcuni problemi possono non garantire una soluzione ammissibile

Quindi, è fondamentale studiare il problema prima dell'algoritmo

Quando si usano?

Le euristiche costruttive si usano

1. quando forniscono la soluzione ottima
2. quando i tempi di esecuzione devono essere molto ridotti
(ad es., per i problemi on-line: schedulatori, servizi a chiamata, ...)
3. quando il problema ha dimensioni colossali o richiede calcoli complessi (ad esempio, alcuni dati sono noti per simulazione)
4. come componenti di altri algoritmi, per esempio come
   - fase iniziale per algoritmi di scambio
   - procedura di base per algoritmi di ricombinazione

Un caso notevole

Un caso di particolare importanza è quello in cui

- si estende la funzione obiettivo da X a $\mathcal{F}_A$
- si sceglie l'elemento ammissibile che produce il sottoinsieme migliore

$$\varphi_A(i,x) = f(x \cup \{i\})$$

Algorithm Greedy(I)
$x := \emptyset$;
$x^* := \emptyset$; $f^* := +\infty$; { Miglior soluzione trovata sinora }
While $\mathrm{Ext}_A(x) \neq \emptyset$ *do*
  $i := \arg\min_{i \in \mathrm{Ext}_A(x)} f(x \cup \{i\})$;
  $x := x \cup \{i\}$;
  If $x \in X$ and $f(x) < f^*$ *then* $x^* := x$; $f^* := f(x)$;
Return (x^*, f^*);

Il problema dello zaino unitario

Si vuole scegliere da un insieme di oggetti di pari volume un sottoinsieme di valore massimo che possa stare in uno zaino di capacità limitata

In questo caso speciale del *KP* il vincolo di volume diventa di cardinalità: sono ammissibili tutte e sole le soluzioni con $|x| \leq \lfloor V/v \rfloor$

Algorithm GreedyUKP(I)
$x := \emptyset$;
$x^* := \emptyset$; $f^* := 0$; { Miglior soluzione trovata sinora }
While $|x| < \lfloor V/v \rfloor$ *do*
  $i := \arg \max_{i \in B \setminus x} \phi_i$;
  $x := x \cup \{i\}$;
  If $x \in X$ and $f(x) > f^*$ *then* $x^* := x$; $f^* := f(x)$;
Return (x^*, f^*);

- Il sottoinsieme x è estendibile finché $|x| < \lfloor V/v \rfloor$
- Qualsiasi elemento di $B \setminus x$ estende x in modo ammissibile
- La funzione obiettivo è additiva, e quindi

$$f(x \cup \{i\}) = f(x) + \phi_i \Rightarrow \arg \max_{i \in B \setminus x} f(x \cup \{i\}) = \arg \max_{i \in B \setminus x} \phi_i$$

- Ogni $x^{(t)}$ è migliore della precedente: si restituisce l'ultima

Esempio: il problema dello zaino unitario

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1

$$v_i = 1 \text{ per ogni } i \in B$$

$$V = 4$$

L'algoritmo esegue i seguenti passi:

1. $x := \emptyset$;
2. poiché $|x| = 0 < 4$, valuta $i := a$ e aggiorna $x := \{a\}$;
3. poiché $|x| = 1 < 4$, valuta $i := d$ e aggiorna $x := \{a, d\}$;
4. poiché $|x| = 2 < 4$, valuta $i := c$ e aggiorna $x := \{a, c, d\}$;
5. poiché $|x| = 3 < 4$, valuta $i := e$ e aggiorna $x := \{a, c, d, e\}$;
6. poiché $|x| = 4 \not< 4$, termina

Questo algoritmo trova sempre la soluzione ottima

Ma perché?

Il problema dello zaino

Si vuole scegliere da un insieme di oggetti di vario volume un sottoinsieme di valore massimo che possa stare in uno zaino di capacità limitata

Algorithm GreedyKP(I)
$x := \emptyset$;
$x^* := \emptyset$; $f^* := 0$; { Miglior soluzione trovata sinora }
While $\mathrm{Ext}_A(x) \neq \emptyset$ *do*
  $i := \arg \max_{i \in \mathrm{Ext}_A(x)} \phi_i$;
  $x := x \cup \{i\}$;
Return $(x, f(x))$;

- Solo alcuni elementi di $B \setminus x$ estendono x in modo ammissibile

$$\mathrm{Ext}_A(x) = \{i \in B \setminus x : \sum_{j \in x} v_j + v_i \leq V\}$$

- La funzione obiettivo è additiva, e quindi

$$f(x \cup \{i\}) = f(x) + \phi_i \Rightarrow \arg \max_{i \in \mathrm{Ext}_A(x)} f(x \cup \{i\}) = \arg \max_{i \in \mathrm{Ext}_A(x)} \phi_i$$

- Ogni $x^{(t)}$ è migliore della precedente: si restituisce l'ultima

Esempio: il problema dello zaino

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1
v	5	3	2	3	1	1

$$V = 8$$

L'algoritmo esegue i seguenti passi:

1. $x := \emptyset$;
2. poiché $\mathrm{Ext}_A(x) \neq \emptyset$, sceglie $i := a$ e aggiorna $x := \{a\}$;
3. poiché $\mathrm{Ext}_A(x) \neq \emptyset$, sceglie $i := d$ e aggiorna $x := \{a, d\}$;
4. poiché $\mathrm{Ext}_A(x) = \emptyset$, termina

Questo algoritmo non ha trovato la soluzione ottima $x^* = \{a, c, e\}$

Ma perché?

Il Travelling Salesman Problem

Dato un grafo orientato e una funzione costo definita sugli archi,
si cerca il ciclo di costo minimo che ricopra tutti i nodi del grafo

L'algoritmo costruttivo aggiunge ogni volta l'arco meno costoso fra quelli
che non chiudono sottocicli e mantengono grado ≤ 1 in tutti i nodi

Algorithm GreedyTSP(I)

$x := \emptyset$;

$x^* := \emptyset$; $f^* := 0$; { Miglior soluzione trovata sinora }

While $\text{Ext}_A(x) \neq \emptyset$ *do*

$\quad i := \arg \min_{i \in \text{Ext}_A(x)} c_i$;

$\quad x := x \cup \{i\}$;

If $x \in X$ *then* $x^* := x$; $f^* := f(x)$;

Return (x^*, f^*);

L'unica soluzione ammissibile è l'ultima (ma solo se se ne trova una!)

Esempio: il Travelling Salesman Problem

Per semplicità consideriamo un grafo simmetrico

L'algoritmo esegue i seguenti passi:

1. $x := \emptyset$;
2. poiché $\mathrm{Ext}_A(x) \neq \emptyset$, sceglie $i := (3,5)$ e aggiorna x;
3. poiché $\mathrm{Ext}_A(x) \neq \emptyset$, sceglie $i := (2,4)$ e aggiorna x (è $(5,3) \notin \mathrm{Ext}_A(x)$);
4. poiché $\mathrm{Ext}_A(x) \neq \emptyset$, sceglie $i := (5,2)$ e aggiorna x (è $(4,2) \notin \mathrm{Ext}_A(x)$);
5. poiché $\mathrm{Ext}_A(x) \neq \emptyset$, sceglie $i := (4,1)$ e aggiorna x:
   si noti che $(2,5)$, $(4,5)$, $(5,4)$, $(3,4)$ e $(4,3) \notin \mathrm{Ext}_A(x)$!
6. poiché $\mathrm{Ext}_A(x) = \emptyset$, termina

L'algoritmo non trova soluzione ammissibile

Aggiungendo l'arco $(1,3)$ con costo 100, trova soluzione, ma non ottima

Ma perché?

Esempio: il *MDP*

Si vuole scegliere da un insieme di punti un sottoinsieme di k punti con la massima somma delle distanze reciproche

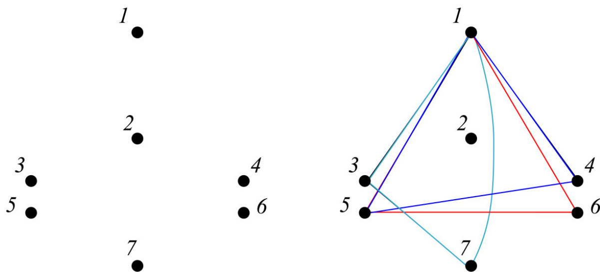
Algorithm GreedyMDP(I)
$x := \emptyset$;
$x^* := \emptyset$; $f^* := 0$; { Miglior soluzione trovata sinora }
While $|x| < k$ *do*
$\quad i := \arg\max_{i \in B \setminus x} \sum_{j \in x} d_{ij}$;
$\quad x := x \cup \{i\}$;
$\quad x^* := x$; $f^* := f(x)$;
Return (x^*, f^*);

- Il sottoinsieme x è estendibile finché $|x| < k$
- Qualsiasi elemento di $B \setminus x$ estende x in modo ammissibile
- La funzione obiettivo è quadratica, e quindi

$$f(x \cup \{i\}) = f(x) + 2\sum_{j \in x} d_{ij} + d_{ii} \Rightarrow \arg\max_{i \in B \setminus x} f(x \cup \{i\}) = \arg\max_{i \in B \setminus x} \sum_{j \in x} d_{ij}$$

- Ogni $x^{(t)}$ è migliore della precedente: si restituisce l'ultima

Esempio: il Maximum Diversity Problem



L'algoritmo ha diversi problemi

1. alla prima iterazione, tutti gli elementi si equivalgono $(f(\{i\}) = 0)$
2. il risultato finale non è ottimo anche se alla prima iterazione
   - si sceglie la coppia di elementi più lontani (cioè $(1, 7)$)
   - si ripete l'algoritmo scegliendo ciascun elemento per primo (ad es., 5)

Ma perché?

Elementi rilevanti

Quali elementi decidono se un algoritmo costruttivo trova l'ottimo o no?

- Forse che lo spazio di ricerca coincida con le soluzioni ($\mathcal{F} = X$)?
  (*No, perché vale per lo zaino sia unitario sia generico*)
- Forse che il vincolo sia di semplice cardinalità?
  (*Spiega il fallimento sullo zaino generico, ma non su MDP e TSP*)
- Forse che la funzione obiettivo sia additiva?
  (*Non spiega il fallimento sul TSP*)

Non esiste una caratterizzazione generale dei problemi risolubili con l'algoritmo costruttivo

Però esistono delle caratterizzazioni parziali

Un caso ben caratterizzato: funzioni additive su basi

Supponiamo che

1. la funzione obiettivo sia additiva

$$\exists \phi : B \rightarrow \mathbb{R} : f(x) = \sum_{i \in x} \phi_i$$

2. le soluzioni siano le basi, cioè i sottoinsiemi massimali, dello spazio di ricerca

$$X = \mathcal{B}_{\mathcal{F}_A} = \{Y \in \mathcal{F} : \nexists Y' \in \mathcal{F} : Y \subset Y'\}$$

È un caso molto frequente (*KP*, *MDP*, *MAX-SAT*, *TSP*, non *SCP*)

In questo caso, l'algoritmo costruttivo A trova sempre la soluzione ottima se e solo se $(B, \mathcal{F}_A)$ è un *matroid embedding*

La definizione di *matroid embedding* è piuttosto complessa

Ci limiteremo a vedere due casi particolari (condizioni sufficienti)

1. matroidi
2. greedoidi dotati di scambio forte

Matroide è un sistema di insiemi $(B, \mathcal{F})$ con $\mathcal{F} \subseteq 2^B$ tale che

- assioma banale: $\emptyset \in \mathcal{F}$
- assioma di ereditarietà: se $x \in \mathcal{F}$ e $y \subset x$ allora $y \in \mathcal{F}$
  Ogni sottoinsieme valido si può costruire aggiungendo gli elementi in ordine qualsiasi
- assioma di scambio: per ogni $x, y \in \mathcal{F}$ con $|x| = |y| + 1$,
  $\exists i \in x \setminus y$ tale che $y \cup \{i\} \in \mathcal{F}$
  Ogni sottoinsieme valido si può estendere con un opportuno elemento di qualsiasi altro sottoinsieme di cardinalità superiore

Queste condizioni

- valgono nel *KP* unitario, nel *MST* (Kruskal, ma non Prim)...
- non valgono nel *KP* generico, nel *TSP*...
- varrebbero nel *MDP*, ma la funzione obiettivo non è additiva

Matroide uniforme e zaino unitario

$$\mathcal{F} = \{x \subseteq B : |x| \leq \lfloor V/v \rfloor\}$$

- Assioma banale: l'insieme vuoto rispetta il vincolo di cardinalità
- Assioma di ereditarietà: se x rispetta il vincolo di cardinalità, tutti i suoi sottoinsiemi lo rispettano
- Assioma di scambio: se x e y rispettano il vincolo di cardinalità e $|x| = |y| + 1$, si può sempre aggiungere un opportuno elemento di x a y senza violare la cardinalità (*in effetti, qualsiasi elemento di* x)

Per il *KP* generico valgono i primi due assiomi, ma non il terzo

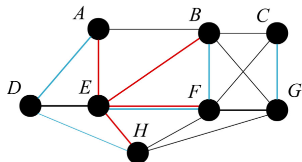
Esempio:

se $V = 6$ e $v = [\,3\;3\;2\;2\;1\,]$, i sottoinsiemi $x = \{3, 4, 5\}$ e $y = \{1, 2\}$ sono in $\mathcal{F}$, ma nessun elemento di x può essere aggiunto a y

Matroide grafico e albero minimo

$$\mathcal{F} = \{x \subseteq B : x \text{ non forma cicli } \}$$

- Assioma banale: l'insieme vuoto rispetta il vincolo di cardinalità
- Assioma di ereditarietà: se x è aciclico, tutti i suoi sottoinsiemi lo sono
- Assioma di scambio: se x e y sono aciclici e $|x| = |y| + 1$, si può sempre aggiungere un opportuno lato di x a y senza chiudere cicli (*non tutti i lati di x vanno bene*)



$$x = \{(A, D), (D, H), (E, F), (B, F), (C, G)\}$$

$$y = \{(A, E), (B, E), (E, F), (E, H)\}$$

Per il *TSP* valgono i primi due assiomi, ma non il terzo

Greedoidi

Greedoide è un sistema di insiemi $(B, \mathcal{F})$ con $\mathcal{F} \subseteq 2^B$ tale che

- assioma banale: $\emptyset \in \mathcal{F}$
- assioma di accessibilità: (*versione debole dell'ereditarietà*)
  se $x \in \mathcal{F}$ e $x \neq \emptyset$ allora $\exists i \in x : x \setminus \{i\} \in \mathcal{F}$
  Ogni sottoinsieme valido si può costruire aggiungendo gli elementi in un ordine opportuno
- assioma di scambio: per ogni $x, y \in \mathcal{F}$ con $|x| = |y| + 1$,
  $\exists i \in x \setminus y$ tale che $y \cup \{i\} \in \mathcal{F}$

In generale l'algoritmo costruttivo non funziona sui greedoidi

Però funziona in questo caso (algoritmo di Prim):

- B = insieme dei lati di un grafo
- $\mathcal{F}$ = collezione degli alberi contenenti un dato vertice v_1

Greedoidi con assioma di scambio forte

Nel caso dell'albero ricoprente, l'algoritmo funziona perché vale un

- assioma di scambio forte:

$$\begin{cases} x \in \mathcal{F}, y \in \mathcal{B}_{\mathcal{F}} \text{ tali che } x \subseteq y \\ i \in B \setminus y \text{ tale che } x \cup \{i\} \in \mathcal{F} \end{cases} \Rightarrow \exists j \in y \setminus x : \begin{cases} x \cup \{j\} \in \mathcal{F} \\ y \cup \{i\} \setminus \{j\} \in \mathcal{F} \end{cases}$$

  Data una base e un suo sottoinsieme (da cui la base è accessibile),
  se esiste un elemento che "svia" il sottoinsieme dalla base,
  deve esisterne un altro che lo mantiene sulla via giusta
  e i due elementi devono potersi scambiare nella base

Si noti che l'ottimalità di un algoritmo costruttivo A dipende

- non solo dalle proprietà del problema
  (funzione obiettivo additiva, basi come soluzioni ammissibili)
- ma anche dallo spazio di ricerca $\mathcal{F}_A$ associato all'algoritmo

Algoritmi costruttivi non esatti: il *KP*

Se il problema non ammette uno spazio di ricerca con proprietà adatte, bisogna tener conto dei vincoli del problema adottando

1. non solo una definizione corretta di $\mathcal{F}_A$
2. ma anche una definizione sofisticata del criterio di scelta $\varphi_A(i, x)$

Questo consente risultati efficaci, pur se non dimostrabilmente ottimi

Nel caso dello zaino, il problema è dato dal volume degli oggetti: vogliamo oggetti di valore alto, ma di volume basso

- usiamo come funzione di scelta il valore unitario: $\varphi_A(i, x) = \dfrac{\phi_i}{v_i}$

L'algoritmo risultante

- può funzionare molto male
- con una piccola modifica è 2-approssimato

Esempio: il *KP*

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1
v	5	3	2	3	1	1
ϕ/v	1.40	0.67	2.00	1.67	4	1

$$V = 8$$

L'algoritmo esegue i seguenti passi:

1. $x := \emptyset$;
2. sceglie $i := e$ e aggiorna $x := \{e\}$;
3. sceglie $i := c$ e aggiorna $x := \{c, e\}$;
4. sceglie $i := d$ e aggiorna $x := \{c, d, e\}$;
5. sceglie $i := f$ e aggiorna $x := \{c, d, e, f\}$; (*l'oggetto a non ci sta*)
6. poiché $\mathrm{Ext}_A(x) = \emptyset$, termina

La soluzione trovata vale 14, quella ottima è $x^* = \{a, c, e\}$ e vale 15

Esempio: il *KP*

Ci sono però casi critici

B	a	b
ϕ	10	90
v	1	10
ϕ/v	10	9

$$V = 10$$

L'algoritmo esegue i seguenti passi:

1. $x := \emptyset$;
2. sceglie $i := a$ e aggiorna $x := \{b\}$;
3. poiché $\mathrm{Ext}_A(x) = \emptyset$, termina

La soluzione trovata vale 10, quella ottima vale 90:
esistono istanze con errore grande a piacere

La causa dell'errore è

- il primo oggetto scartato
- quando ha volume grande, e quindi valore grande

Esempio: il *KP*

Algoritmo approssimato per il *KP*

1. Si parte con un sottoinsieme vuoto: $x^{(0)} = \emptyset$
2. Si trova l'oggetto i di valore unitario massimo in $B \setminus x$
3. Se non eccede il volume, si mette in soluzione e si torna al punto 2

$$x^{(t-1)} = \{i_1, i_2, \ldots, i_{t-1}\} \to x^{(t)} = \{i_1, i_2, \ldots, i_t\}$$

4. Altrimenti, si costruisce una soluzione col solo oggetto

$$x' = \{i_t\}$$

5. Si restituisce la soluzione migliore fra x e x': $f_A = \max\left[f(x), f(x')\right]$

È facile dimostrare che

- la somma delle due soluzioni è una stima per eccesso dell'ottimo

$$f(x) + f(x') = \sum_{\tau=1}^{t} \phi_{i_\tau} \geq f^*$$

- la migliore delle due soluzioni è almeno metà della somma

$$f_A = \max\left[f(x), f(x')\right] \geq \frac{f(x) + f(x')}{2} \geq \frac{1}{2} f^*$$

Algoritmi costruttivi puri e adattivi

Un algoritmo costruttivo A si definisce

- puro se la funzione di scelta $\varphi_A(\cdot)$ dipende solo dal nuovo elemento i
- adattivo se $\varphi_A(\cdot)$ dipende anche dalla soluzione corrente x

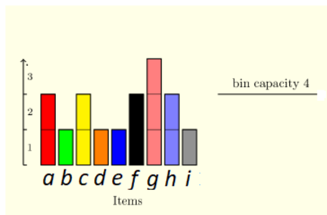
Sinora abbiamo considerato solo algoritmi puri

Consideriamo il *BPP*: si vuole dividere un insieme O di oggetti voluminosi nel minimo numero di contenitori di capacità data tratti da un insieme C

$B = O \times C$ contiene gli assegnamenti oggetto-contenitore (i, j)

- con uno e un solo contenitore per oggetto
- con volume non superiore alla capacità per ogni contenitore

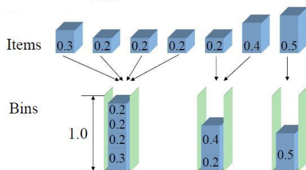
Lo spazio di ricerca $\mathcal{F}_A$ coincide con l'insieme delle soluzioni parziali



Algoritmo First-Fit

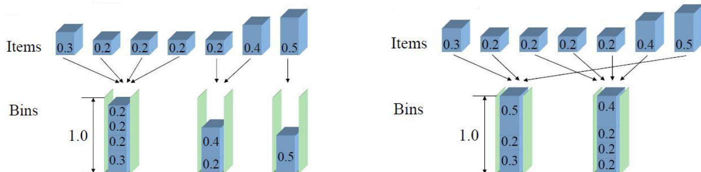
- Si parte con un sottoinsieme vuoto: $x^{(0)} = \emptyset$
- Si prende un oggetto i qualsiasi
- Si sceglie il contenitore j in modo da minimizzare il numero di contenitori usati: (*la scelta dipende da* x, *non solo da* i*!*)
    - si prende il primo contenitore usato con capacità residua sufficiente
    - se nessuno ha capacità residua sufficiente, se ne usa uno nuovo
- Si aggiunge alla soluzione il nuovo assegnamento

$$x^{(t)} \leftarrow x^{(t-1)} \cup \{(i,j)\}$$



Algoritmo First-Fit

La soluzione non è ottima



Però è approssimata:

- occorrono almeno $f^* \geq \sum_{i \in O} v_i / V$ contenitori
- i contenitori usati, tranne al più l'ultimo, hanno contenuto $> V/2$
  (*gli oggetti del secondo semivuoto sarebbero finiti nel primo*)
- il volume totale supera quello degli $f_A - 1$ contenitori "pieni"

$$\sum_{i \in O} v_i > (f_A - 1)\, V/2$$

- da cui $(f_A - 1) \leq 2 \frac{\sum_{i \in O} v_i}{V} \leq 2f^* \Rightarrow f_A \leq 2f^* + 1$

Algoritmo First-Fit decreasing

Il fattore $\alpha = 2$ vale prendendo gli oggetti in qualsiasi ordine

L'intuizione direbbe che sia meglio prendere prima gli oggetti piccoli, perché questo tiene più bassa la funzione obiettivo $f(x, i)$

Ma così si dimentica che gli oggetti vanno comunque presi tutti

Al contrario, è bene prenderli in ordine di volume decrescente perché

- ogni oggetto nel contenitore j ha volume strettamente maggiore della capacità residua di tutti i contenitori precedenti
  (*altrimenti, sarebbe stato messo in uno di loro*)
- tenendo gli oggetti piccoli in fondo, garantiamo che molti contenitori abbiano capacità residua piccola

In questo modo, il fattore migliora: $f_A \leq \frac{11}{9} f^* + 1$

Set Covering

Data una matrice binaria e un vettore di costi associati alle colonne, si cerca il sottoinsieme di colonne di costo minimo che copra tutte le righe

La funzione obiettivo è additiva, ma le soluzioni non sono sottoinsiemi massimali (anzi, conviene che siano piccoli, purché ammissibili)

Una funzione di scelta pura $\varphi_A(i)$ porta a scegliere colonne che coprono inutilmente le stesse righe, purché siano poco costose: serve $\varphi_A(i,x)$

Le idee che appaiono più promettenti sono di considerare

- la funzione obiettivo: scegliere colonne di costo basso
- la soluzione parziale x: scegliere colonne che coprono righe nuove
- i vincoli: scegliere colonne che coprono molte righe

Di conseguenza

- includere in $\mathrm{Ext}_A(x)$ solo colonne che coprono righe aggiuntive
- usare come funzione di scelta $\varphi_A(i,x) = \dfrac{c_i}{a_i(x)}$

  dove $a_i(x)$ è il numero di righe coperte da i, ma non da x

Set Covering: un esempio positivo

c	3	5	6	2	1	7	1	8

A								
	1	0	1	0	0	0	0	1
	0	1	0	0	0	1	0	0
	0	0	0	1	0	0	0	0
	1	0	1	0	0	1	0	0
	0	0	0	0	1	0	1	1
	1	0	0	1	0	1	0	0

L'algoritmo esegue i seguenti passi:

1. $x := \emptyset$;
2. sceglie $i := 1$ e aggiorna $x := \{1\}$;
3. sceglie $i := 5$ e aggiorna $x := \{1, 5\}$;
4. sceglie $i := 4$ e aggiorna $x := \{1, 4, 5\}$;
5. sceglie $i := 2$ e aggiorna $x := \{1, 2, 4, 5\}$;
6. tutte le righe sono coperte, quindi $\mathrm{Ext}_A(x) = \emptyset$ e si termina

La soluzione trovata vale 11 ed è ottima

Set Covering: un esempio negativo

Però può anche andare male

c	25	6	8	24	12

	1	1	0	0	0
	1	1	0	0	0
A	1	1	1	0	0
	1	0	1	1	0
	1	0	0	1	0
	1	0	0	0	1

L'algoritmo esegue i seguenti passi:

1. $x := \emptyset$;
2. essendo $c/a_i(x) = [\, 4.1\overline{6} \quad 2 \quad 4 \quad 12 \quad 12 \,]$, sceglie $i := 2$;
3. essendo $c/a_i(x) = [\, 8.\overline{3} \quad - \quad 8 \quad 12 \quad 12 \,]$, sceglie $i := 3$;
4. essendo $c/a_i(x) = [\, 12.5 \quad - \quad - \quad 24 \quad 12 \,]$, sceglie $i := 5$;
5. essendo $c/a_i(x) = [\, 25 \quad - \quad - \quad 24 \quad - \,]$, sceglie $i := 4$;
6. tutte le righe sono coperte, quindi $\mathrm{Ext}_A(x) = \emptyset$ e si termina

La soluzione trovata è $x = \{2, 3, 4, 5\}$ e vale 50,
mentre la soluzione ottima $x^* = \{1\}$ vale $f^* = 25$

Anche questo algoritmo è approssimato, ma il fattore non è costante

- ad ogni passo t, si valuta ogni colonna i col criterio di scelta

$$\varphi_A\left(i, x^{(t-1)}\right) = \frac{c_i}{a_i\left(x^{(t-1)}\right)}$$

- sia t_j il passo e i_j la colonna in cui la riga j viene coperta
- inizialmente, assegniamo a ogni riga j un peso $\theta_j = 0$
- al passo t_j, diamo alla riga j il peso

$$\theta_j = \frac{c_{i_j}}{a_{i_j}\left(x^{(t_j-1)}\right)}$$

- alla fine, il costo della soluzione x coincide con la somma dei pesi θ_j

$$f_A(x) = \sum_{i \in x} c_i = \sum_{j \in R} \theta_j$$

- siccome ad ogni passo si sceglie la colonna col minimo $\varphi_A\left(i, x^{(t-1)}\right)$ e gli a_i calano, i pesi delle righe via via coperte sono non decrescenti

Approssimabilità del *SCP*

- al passo t, ci sono $|R^{(t)}|$ righe scoperte e le colonne della soluzione ottima potrebbero coprirle tutte con costo f^*
  $\Rightarrow$ almeno una di tali colonne ha costo unitario $\leq f^*/|R^{(t)}|$
- la colonna i scelta ha costo unitario minimo, dunque non superiore e le righe coperte acquistano valori

$$\theta_j \leq \frac{f^*}{|R^{(t_j)}|} \Rightarrow \sum_{j \in R} \theta_j \leq \sum_{j \in R} \frac{f^*}{|R^{(t_j)}|}$$

Per coprire ogni riga, si spende un costo non superiore all'ottimo diviso per il numero di righe scoperte al momento della copertura

- questo numero $|R^{(t)}|$ scende a scatti ad ogni nuova colonna
- si maggiora la sommatoria facendo scendere $|R^{(t)}|$ un'unità per volta
- L'algoritmo costruttivo per il *SCP* ha approssimazione logaritmica

$$f_A = \sum_{j \in R} \theta_j \leq \sum_{j \in R} \frac{f^*}{|R^{(t_j)}|} \leq \sum_{r=|R|}^{1} \frac{f^*}{r} \leq (\ln |R| + 1)\ f^*$$

c	25	6	8	24	12

A	1	1	0	0	0
	1	1	0	0	0
	1	1	1	0	0
	1	0	1	1	0
	1	0	0	1	0
	1	0	0	0	1

1. essendo $\varphi_A(i,x) = \begin{bmatrix} 4.1\bar{6} & 2 & 4 & 12 & 12 \end{bmatrix}$, si sceglie $i := 2$ e si pone $\theta_1 = \theta_2 = \theta_3 = 2$, che è $\leq f^*/|R^{(0)}| = 25/6 = 4.1\bar{6}$;
   È $\theta_1 \leq 25/6$, e a maggior ragione $\theta_2 \leq 25/5$ e $\theta_3 \leq 25/4$
2. essendo $\varphi_A(i,x) = \begin{bmatrix} 8.\bar{3} & - & 8 & 12 & 12 \end{bmatrix}$, si sceglie $i := 3$ e si pone $\theta_4 = 8$, che è $\leq f^*/|R^{(1)}| = 8.\bar{3}$
3. essendo $\varphi_A(i,x) = [\ 12.5 \quad - \quad - \quad 24 \quad 12\ ]$, si sceglie $i := 5$ e si pone $\theta_6 = 12$, che è $\leq f^*/|R^{(2)}| = 12.5$
4. essendo $\varphi_A(i,x) = [\ 25 \quad - \quad - \quad 24 \quad -\ ]$, si sceglie $i := 2$ e si pone $\theta_5 = 24$, che è $\leq f^*/|R^{(3)}| = 25$
5. tutte le righe sono coperte, quindi $\mathrm{Ext}_A(x) = \emptyset$ e l'algoritmo termina

Ora $f_A = \sum_{j \in R} \theta_j = 50$ e vale l'approssimazione $f_A \leq (\ln|R| + 1)\, f^* \approx 2.79\, f^*$

L'algoritmo *Nearest Neighbour* per il *TSP*

Consideriamo il *TSP* su grafo completo $G = (N, A)$ col consueto spazio di ricerca (sottoinsiemi di archi che non hanno sottocicli e hanno grado ≤ 1 per tutti i nodi)

- l'algoritmo costruttivo trova sempre una soluzione ammissibile
- la soluzione può essere cattiva a piacere (*Perché?*)

Cambiamo spazio di ricerca: $\mathcal{F}_A$ include i cammini uscenti dal nodo 1

- vale sempre l'assioma banale
- non vale l'ereditarietà: non tutti i sottoinsiemi sono cammini
- vale l'accessibilità: se si toglie l'ultimo arco, rimane un cammino uscente dal nodo 1 (*quindi è un greedoide*)
- e l'assioma di scambio forte?
  (*se valesse, avremmo un algoritmo polinomiale per il TSP*)

Non vale né l'assioma di scambio forte, né l'assioma di scambio

L'algoritmo costruttivo che ne deriva è detto *Nearest Neighbour*

L'algoritmo *Nearest Neighbour* per il *TSP*

$\text{Ext}_A(x)$ contiene gli archi uscenti dall'ultimo nodo del cammino x
e che non chiudono sottocicli

$$\text{Ext}(\text{x}) = \{(h,k) \in A : h = \text{Last}(x), k \notin N_x \text{ or } k = 1 \text{ and } N_x = N\}$$

dove N_x è l'insieme dei nodi visitati da x e Last(x) l'ultimo

- Si parte con un insieme di archi vuoto: $x^{(0)} = \emptyset$
  che rappresenta un cammino degenere uscente dal nodo 1
  (*la soluzione ottima passa sicuramente dal nodo* 1)
- Si cerca l'arco di costo minimo uscente dall'ultimo nodo di x

$$(i,j) = \arg \min_{(h,k) \in \text{Ext}(\text{x})} c_{hk}$$

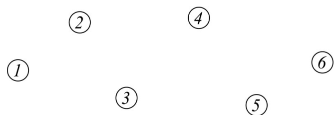
(*è un algoritmo costruttivo puro*)

- Se $j \neq 1$, torna al punto 2; altrimenti, termina
  ($\text{Ext}(x)$ *impone di tornare a* 1 *solo all'ultimo passo*)

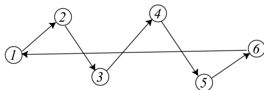
L'algoritmo è molto intuitivo e ha complessità $\Theta(n^2)$

Non è esatto, ma $\log n$-approssimato (*sotto la disuguaglianza triangolare*)

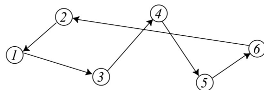
Si consideri un grafo completo (per semplicità non riportiamo gli archi)



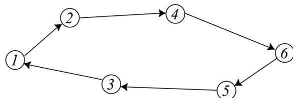
Partendo dal nodo 1



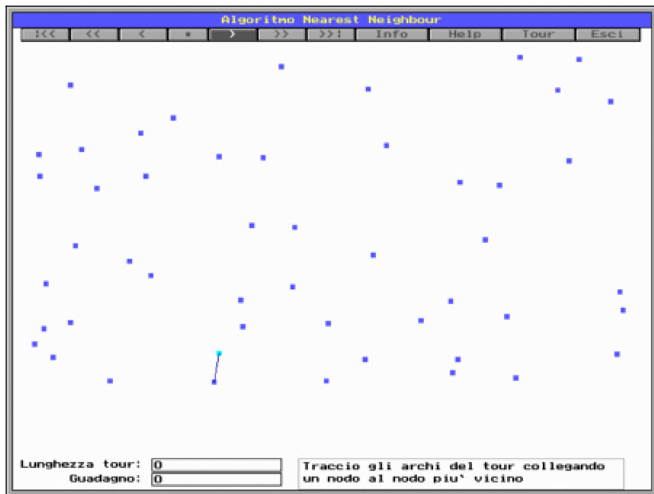
Partendo dal nodo 2



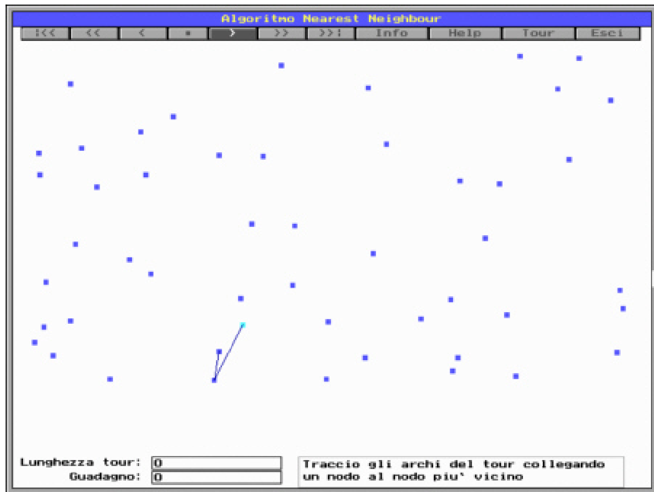
La soluzione ottima non si trova partendo da nessun nodo



Un esempio più grande



Un esempio più grande



Un esempio più grande

Algoritmo Nearest Neighbour

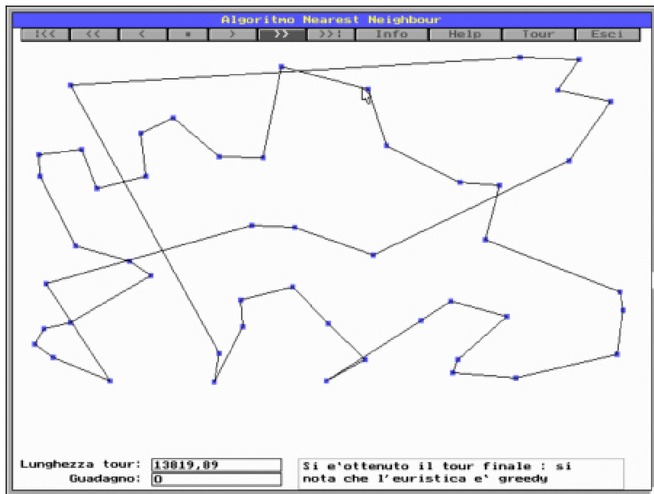
|<< | << | < | ■ | > | >> | >>| | Info | Help | Tour | Esci

Lunghezza tour: 0

Guadagno: 0

Traccio gli archi del tour collegando un nodo al nodo piu' vicino

Un esempio più grande



Estensioni dell'algoritmo costruttivo

L'algoritmo costruttivo aggiunge un elemento alla volta alla soluzione

È possibile generalizzare questo schema con algoritmi che

1. aggiungono più elementi ad ogni passo: la funzione di scelta $\varphi_A(B^+, x)$ individua sottoinsiemi $B^+ \subseteq B \setminus x$ da aggiungere, anziché un singolo elemento i
2. eliminano elementi, ma ne aggiungono un numero maggiore: la funzione di scelta $\varphi_A(B^+, B^-, x)$ individua sottoinsiemi $B^+ \subseteq B \setminus x$ da aggiungere e sottoinsiemi $B^- \subseteq x$ da togliere, con $|B^+| > |B^-|$

L'idea di fondo rimane visitare lo spazio di ricerca in modo "aciclico", cioè senza mai tornare indietro

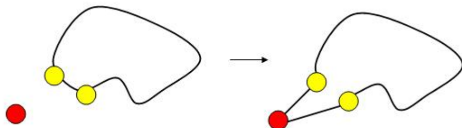
Il problema fondamentale è definire famiglie di sottoinsiemi tali che ottimizzare la funzione di scelta rimanga un problema polinomiale

- sottoinsiemi di dimensione limitata (ad es., $|B^+| = 2$ e $|B^-| = 1$)
- sottoinsiemi ottimizzabili efficientemente (cammini minimi,. . .)

Algoritmi di inserimento per il *TSP*

Diversi algoritmi euristici per il *TSP* definiscono lo spazio di ricerca $\mathcal{F}_A$ come l'insieme di tutti i cicli del grafo passanti per un nodo dato

- non si può ottenere un ciclo da un altro aggiungendo un singolo arco
- si può farlo togliendo un arco e aggiungendone due



- Si parte con un insieme di archi vuoto: $x^{(0)} = \emptyset$
  che rappresenta un ciclo degenere centrato sul nodo 1
- Si sceglie un arco (i, j) da togliere e un nodo k da aggiungere
- Se il ciclo non tocca tutti i nodi si torna al punto 2;
  altrimenti si termina

Viola lo schema classico, ma non rivisita mai soluzioni e dopo n passi fornisce una soluzione ammissibile (*ad ogni passo entra un nodo nuovo*)

Algoritmi di inserimento per il *TSP*

La funzione di scelta $\varphi_A(B^+, B^-, x)$ deve trovare un nodo e un arco;
le scelte possibili sono $O(n^2)$

- $|x|$ possibili archi (s_i, s_{i+1}) da togliere
- $n - |x|$ possibili nodi k da aggiungere con gli archi (s_i, k) e (k, s_{i+1})

L'algoritmo *Cheapest Insertion* usa come criterio di scelta

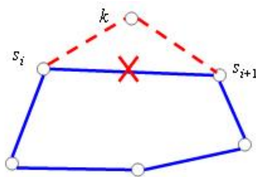
$$\varphi_A(B^+, B^-, x) = f(x \cup B^+ \setminus B^-)$$

La funzione obiettivo $f(x)$ è additiva, dunque estendibile a tutto $\mathcal{F}$

Siccome $f(x \cup B^+ \setminus B^-) = f(x) + c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}}$

$$\arg\min_{B^+,B^-} \varphi_A(B^+, B^-, x) = \arg\min_{i,k} \left(c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}}\right)$$

Il costo computazionale della valutazione di φ_A scende da $\Theta(n)$ a $\Theta(1)$



Algoritmo *Cheapest Insertion* per il *TSP*

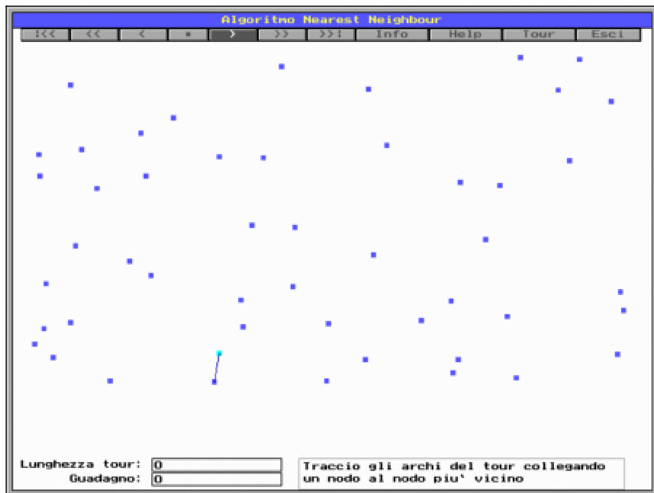
L'algoritmo *Cheapest Insertion*

- parte con un insieme di archi vuoto: $x^{(0)} = \emptyset$
  che rappresenta un ciclo degenere centrato sul nodo 1
- sceglie l' arco $(s_i, s_{i+1}) \in x$ e il nodo $k \notin N_x$ tali che
  $(c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}})$ sia minimo
- se il ciclo non tocca tutti i nodi torna al punto 2;
  altrimenti termina

Non è esatto, ma 2-approssimato, sotto la disuguaglianza triangolare

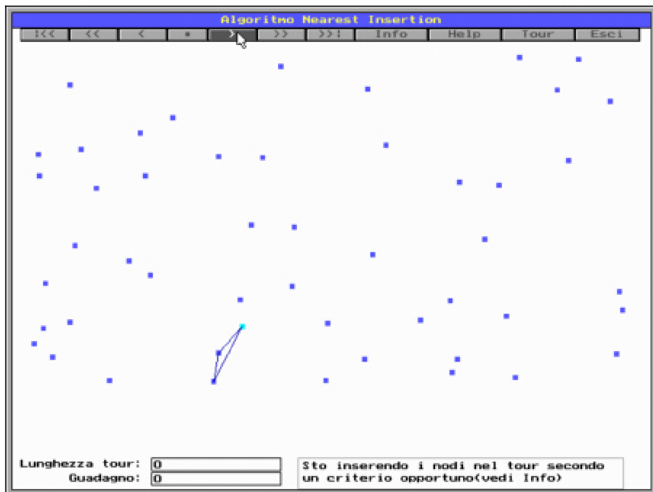
Un esempio

Si comincia come nel *Nearest Neighbour*



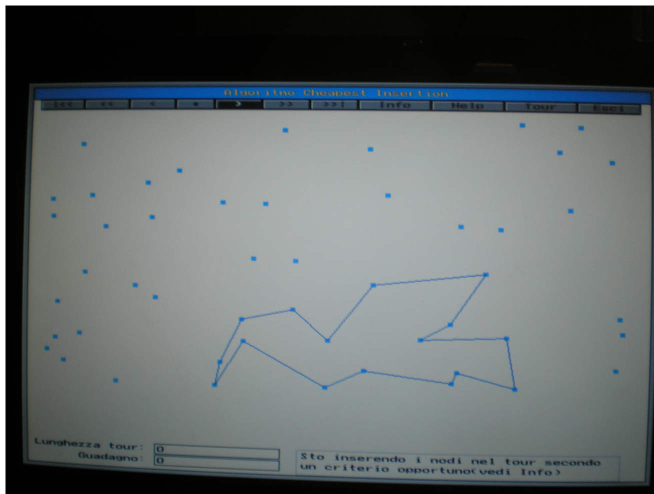
Un esempio

Ma si crea un ciclo, anziché un cammino



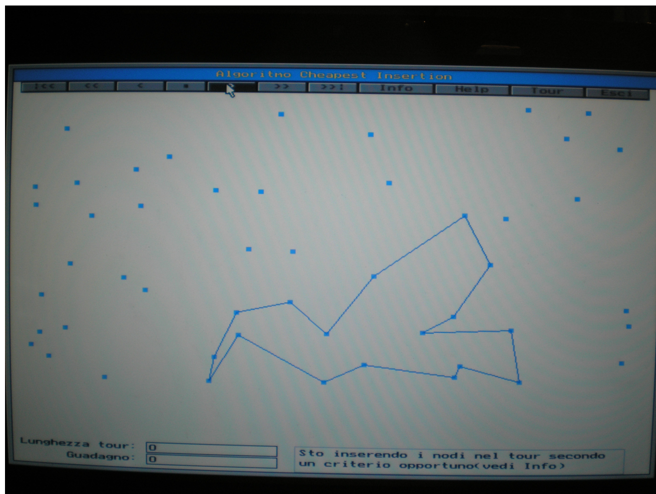
Un esempio

Ogni passo aggiunge il nodo che allunga di meno il ciclo



Un esempio

Ogni passo aggiunge il nodo che allunga di meno il ciclo



Un esempio

Si termina quando il ciclo tocca tutti i nodi

Algoritmo *Cheapest Insertion* per il *TSP*

L'algoritmo esegue n passi

- ad ogni passo, valuta $t(n-t)$ coppie arco-nodo
- ogni valutazione richiede tempo costante
- ogni valutazione eventualmente aggiorna la mossa migliore
- ad ogni passo si esegue l'aggiunta migliore e si valuta se terminare

La complessità totale è $\Theta\left(n^3\right)$

Si può ridurre a $\Theta\left(n^2 \log n\right)$ conservando gli inserimenti possibili per ogni nodo esterno in un *min-heap* (da aggiornare dopo ogni mossa)

Algoritmo *Nearest Insertion* per il *TSP*

L'algoritmo *Cheapest Insertion* tende a scegliere nodi vicini al ciclo x: minimizzare $c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}}$ implica che $c_{s_i,k}$ e $c_{s_{i+1},k}$ siano piccoli

Per accelerare, si può usare una funzione φ_A decomposta in due fasi

L'algoritmo *Nearest Insertion*

- parte con un insieme di archi vuoto: $x^{(0)} = \emptyset$
  che rappresenta un ciclo degenere centrato sul nodo 1
- Criterio di selezione: sceglie il nodo k più vicino al ciclo x

$$k = \arg\min_{\ell \notin N_x} \left(\min_{s_i \in x} c_{s_i,\ell} \right)$$

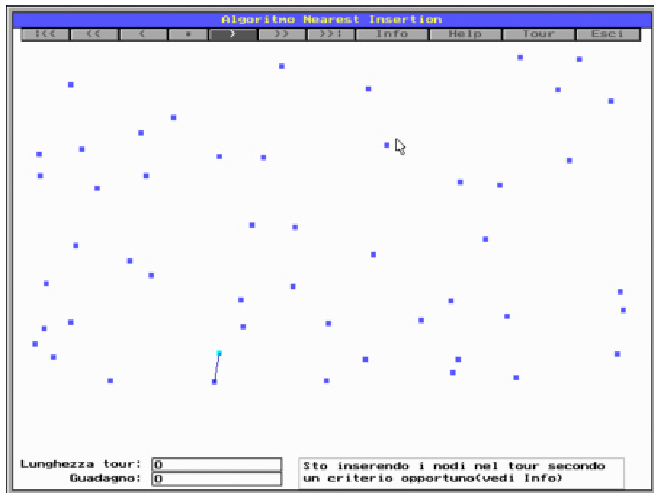
- Criterio di inserimento: sceglie l'arco (s_i, s_{i+1}) che minimizza f

$$(s_i, s_{i+1}) = \arg\min_{s_i \in x} \left(c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}} \right)$$

- Se il ciclo non tocca tutti i nodi, torna al punto 2;
  altrimenti termina

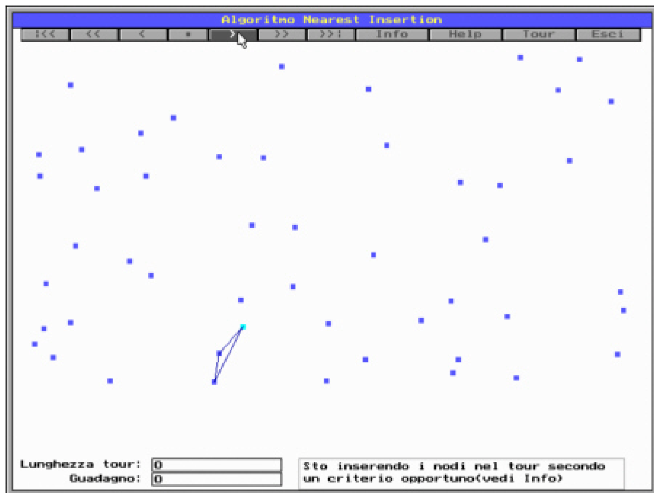
Non è esatto, ma 2-approssimato, sotto la disuguaglianza triangolare

Si comincia come nel *Nearest Neighbour* e nel *Cheapest Insertion*

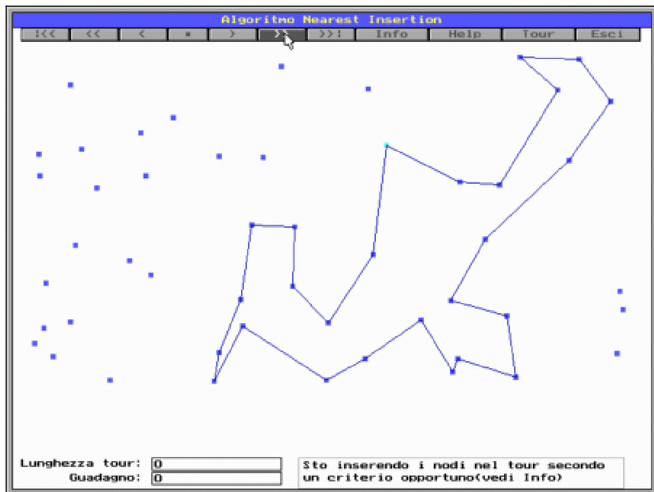


Un esempio

Si crea un ciclo come nel *Cheapest Insertion*

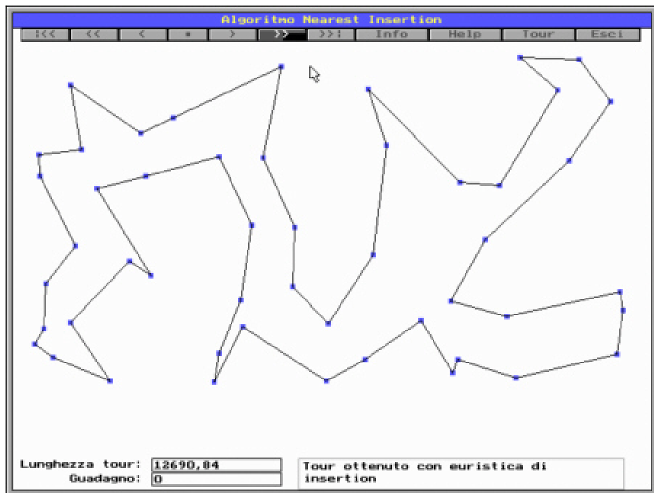


Ma il ciclo cresce diversamente: ogni volta entra il nodo più vicino, anche se questo aumenta il costo più di un altro nodo



Un esempio

Si termina quando il ciclo tocca tutti i nodi



Algoritmo *Nearest Insertion* per il *TSP*

L'algoritmo esegue n passi

- ad ogni passo, valuta $(n - t)$ nodi e trova il più vicino al ciclo
- ogni valutazione richiede $\Theta(n - t)$
- ad ogni passo, valuta t archi e trova il più conveniente da togliere
- ogni valutazione eventualmente aggiorna la mossa migliore
- ad ogni passo esegue l'aggiunta migliore e valuta se terminare

La complessità totale è $\Theta(n^3)$

Si può ridurre a $\Theta(n^2)$ conservando per ogni nodo esterno il nodo interno più vicino (e aggiornandoli dopo ogni passo)

Algoritmo *Farthest Insertion* per il *TSP*

La scelta del nodo più vicino al ciclo è naturale, ma ingannevole, dato che tutti i nodi vanno raggiunti prima o poi

In pratica, conviene servire al meglio i nodi più fastidiosi, cioè lontani

L'algoritmo *Farthest Insertion*

- parte con un insieme di archi vuoto: $x^{(0)} = \emptyset$
  che rappresenta un ciclo degenere centrato sul nodo 1
- Criterio di selezione: sceglie il nodo k più lontano dal ciclo x

$$k = \arg\max_{\ell \notin N_x} \left(\min_{s_i \in x} c_{s_i,\ell} \right)$$

(*il nodo più lontano dal più vicino del ciclo*)

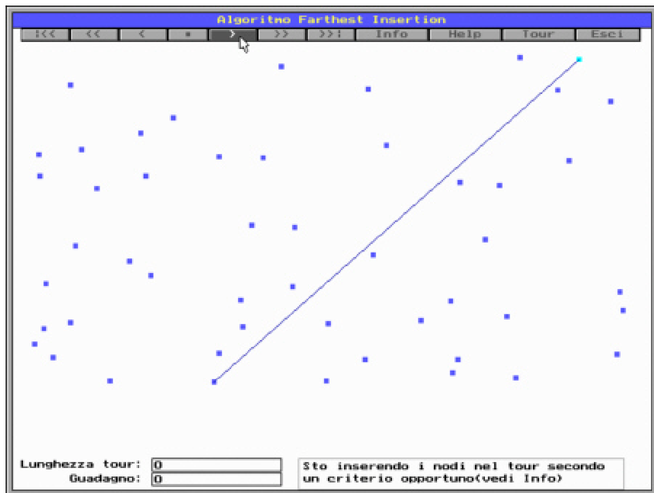
- Criterio di inserimento: sceglie l'arco (s_i, s_{i+1}) che minimizza

$$(s_i, s_{i+1}) = \arg\min_{s_i \in x} \left(c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}} \right)$$

- Se il ciclo non tocca tutti i nodi, torna al punto 2;
  altrimenti termina

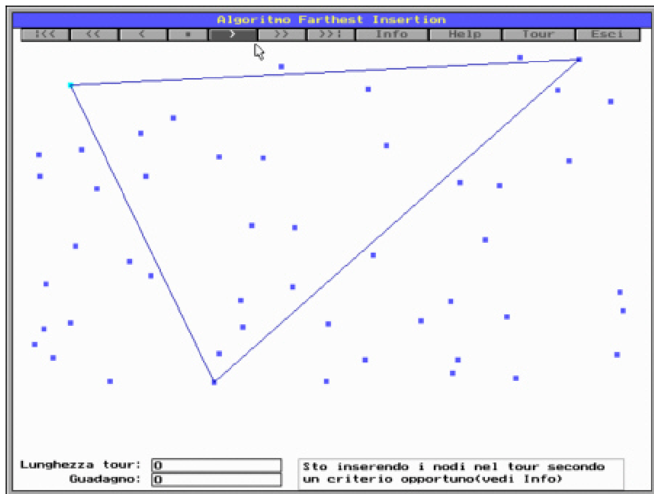
È log n-approssimato sotto la disuguaglianza triangolare, dunque peggio dei precedenti nel caso peggiore (ma sperimentalmente spesso è meglio)

Si parte raggiungendo subito il nodo più lontano

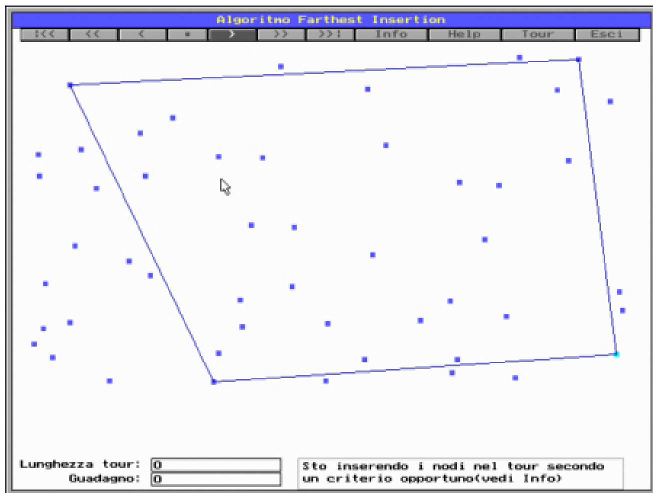


Un esempio

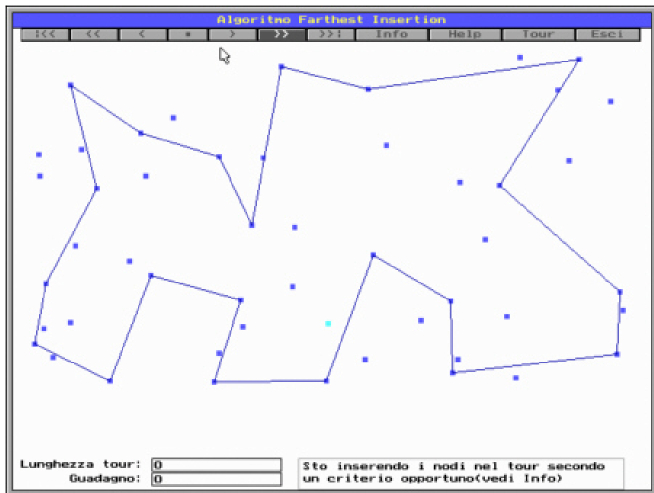
E si va avanti così



Però inserendo sempre questi nodi nel modo migliore possibile

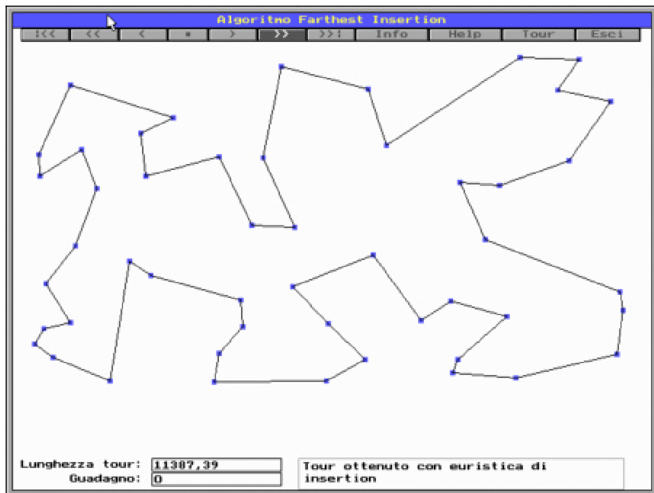


Il ciclo cresce molto più regolare: ha molti meno incroci e sinuosità



Un esempio

Si termina quando il ciclo tocca tutti i nodi



Algoritmo *Farthest Insertion* per il *TSP*

L'algoritmo esegue n passi

- ad ogni passo, valuta $(n - t)$ nodi e trova il più lontano dal ciclo
- ogni valutazione richiede $\Theta(n - t)$
- ad ogni passo, valuta t archi e trova il più conveniente da togliere
- ogni valutazione eventualmente aggiorna la mossa migliore
- ad ogni passo esegue l'aggiunta migliore e valuta se terminare

La complessità totale è $\Theta\left(n^3\right)$

Si può ridurre a $\Theta\left(n^2\right)$ conservando il nodo del ciclo più vicino ad ogni nodo esterno (e aggiornandoli dopo ogni passo)

Distance Heuristic per lo Steiner Tree Problem (*STP*)

Dato un grafo non orientato $G = (V, E)$, con costi sui lati $(c : E \to \mathbb{N})$ e un sottoinsieme di vertici speciali $U \subset V$, si cerca un albero di costo minimo che connetta i vertici speciali

Adotteremo come spazio di ricerca $\mathcal{F}$ l'insieme degli alberi contenenti il vertice speciale 1

Un algoritmo costruttivo classico, che aggiunge un lato alla volta

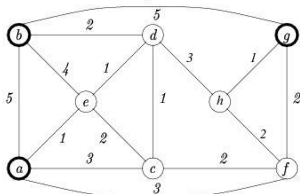
- produce soluzioni con lati ridondanti, dunque costose
- ha difficoltà a capire se i lati via via aggiunti sono utili o ridondanti

L'idea è aggiungere un vertice speciale per volta
e fermarsi quando tutti i vertici speciali sono connessi

- per mantenere la connessione, occorre aggiungere all'albero non un solo lato, ma un intero cammino
- trovare il cammino minimo da un vertice speciale all'albero x è facile

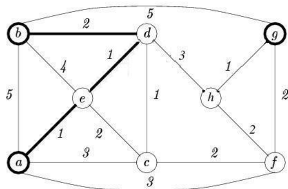
Ogni volta, si può determinare efficientemente l'insieme B^+ dei nuovi lati

Esempio



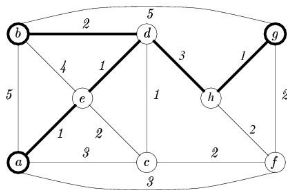
- si parte con il solo vertice speciale a (albero degenere)
- il vertice speciale più vicino è b, attraverso il cammino (a, e, d, b): $x = \{(a, e), (e, d), (d, b)\}$
- il vertice speciale più vicino è g, attraverso il cammino (g, h, d): $x = \{(a, e), (e, d), (d, b), (g, h), (h, d)\}$
- tutti i vertici speciali sono nella soluzione: si termina

Esempio



- si parte con il solo vertice speciale a (albero degenere)
- il vertice speciale più vicino è b, attraverso il cammino (a, e, d, b): $x = \{(a, e), (e, d), (d, b)\}$
- il vertice speciale più vicino è g, attraverso il cammino (g, h, d): $x = \{(a, e), (e, d), (d, b), (g, h), (h, d)\}$
- tutti i vertici speciali sono nella soluzione: si termina

Esempio



- si parte con il solo vertice speciale a (albero degenere)
- il vertice speciale più vicino è b, attraverso il cammino (a, e, d, b): $x = \{(a, e), (e, d), (d, b)\}$
- il vertice speciale più vicino è g, attraverso il cammino (g, h, d): $x = \{(a, e), (e, d), (d, b), (g, h), (h, d)\}$
- tutti i vertici speciali sono nella soluzione: si termina

In questo caso, si trova la soluzione ottima; in generale, è 2-approssimato

Equivale a cercare l'albero ricoprente minimo su un grafo con

- vertici ridotti ai vertici speciali
- lati corrispondenti ai cammini minimi

E le euristiche distruttive?

È l'approccio complementare

- si parte con l'intero insieme base B
- si elimina un elemento per volta, scelto
  - in modo da non uscire dallo spazio di ricerca $\mathcal{F}_A$
  - ottimizzando un criterio opportuno $\varphi_A(i, x)$
- si termina quando non c'è modo di rimanere nello spazio di ricerca

Un'euristica distruttiva (problema di minimo) si può descrivere come

Algorithm Stingy(I)

$x := B;$

$x^* := \emptyset;\ f^* := +\infty;$ { Miglior soluzione trovata sinora }

While $\mathrm{Red}_A(x) \neq \emptyset$ *do*

$\quad i := \arg \max_{i \in \mathrm{Red}_A(x)} \varphi_A(i, x);$

$\quad x := x \setminus \{i\};$

$\quad$ *If* $x \in X$ and $f(x) < f^*$ *then* $x^* := x;\ f^* := f(x);$

Return $(x^*, f^*);$

dove $\mathrm{Red}_A(x) = \{i \in x : x \setminus \{i\} \in \mathcal{F}_A\}$

Perché sono meno usate?

Se le soluzioni sono molto meno numerose dell'insieme base ($|x| \ll |B|$) un'euristica distruttiva

- richiede un numero di passi superiore
- ha maggiori probabilità di commettere un passo sbagliato
- talvolta, la valutazione di $\mathrm{Red}_A(x)$ e $\varphi_A(i,x)$ è più costosa

Combinare un'euristica costruttiva con una distruttiva è utile quando

- l'euristica costruttiva produce soluzioni ridondanti
- la soluzione migliore generata dall'euristica costruttiva non è l'ultima (*la fase distruttiva può visitare altre soluzioni*)

L'euristica distruttiva ausiliaria

- parte dalla soluzione x dell'euristica costruttiva, anziché da B
- spesso il suo spazio di ricerca coincide con l'insieme delle soluzioni:

$$\mathcal{F}_A = X \Rightarrow \mathrm{Red}_A(x) = \{i \in x : x \setminus \{i\} \in X\}$$

- spesso il criterio di scelta è l'obiettivo: $\varphi_A(i,x) = f(x \setminus \{i\})$

Euristica costruttivo/distruttiva per il *SCP*

c	6	8	24	12

	1	0	0	0
	1	0	0	0
A	1	1	0	0
	0	1	1	0
	0	0	1	0
	0	0	0	1

1. L'euristica costruttiva sceglie, nell'ordine, le colonne 1, 2, 4 e 3 (*ognuna copre nuove righe*)
2. La soluzione è ridondante: si può togliere la colonna 2 (*le colonne seguenti hanno coperto anche righe già coperte*)
3. Un *postprocessing* con un'euristica distruttiva banale fornisce in questo caso la soluzione ottima $x^* = \{1, 3, 4\}$ (*le colonne* 1, 3 *e* 4 *sono essenziali per coprire le righe* 1, 2, 5 *e* 6)