Metodi e Modelli per l'Ottimizzazione Combinatoria

Metodi euristici di ottimizzazione combinatoria (Parte I)

L. De Giovanni M. Monaci*

1 Introduzione

I metodi visti finora garantiscono, almeno in linea teorica, di risolvere un problema di ottimizzazione combinatoria in modo esatto, cioè di trovare una soluzione ammissibile che corrisponda all'ottimo della funzione obiettivo tra tutte le soluzione ammissibili. L'applicazione di *metodi esatti* non è sempre possibile, essenzialmente per due motivi concomitanti: la complessità intrinseca del problema (ad esempio, problemi NP-Hard) e il tempo a disposione per generare la soluzione.

Spesso i metodi esatti si basano sulla formulazione di un modello programmazione matematica del problema di ottimizzazione e la complessità del problema potrebbe rendere praticamente impossibile pervenire ad una formulazione matematica che sia sufficientemente accurata e maneggevole. Si pensi, ad esempio, ad un problema di configurazione di una rete di trasporti, in cui si voglia minimizzare il livello di congestione, che dipende dai comportamenti degli utenti della rete: anche se esistono diversi modelli comportamentali, spesso una valutazione realistica della congestione può essere ottenuta solo per via simulativa, il che è difficilmente rappresentabile in un modello di programmazione matematica (in generale, e in un modello di programmazione lineare intera, in particolare). In altri ambiti, pur disponendo di una "buona" formulazione matematica, il tempo a disposizione per la soluzione del problema non consente l'utilizzo di metodi che permettono di individuarne la soluzione ottima: ad esempio, i metodi basati su enumerazione implicita non danno garanzia sul tempo necessario per il loro completamento (risolvere un problema di TSP in alcune ore potrebbe essere ragionevole in alcuni contesti e non accettabile in altri).

Quando il problema e/o il contesto della soluzione non renda possibile applicare tecniche di soluzione esatte diventa necessario fornire delle "buone" soluzioni ammissibili in tempi di calcolo "ragionevoli". Si noti che, tipicamente, la determinazione di buone soluzioni

*Dipartimento di Ingegneria dell'Informazione, Università di Padova (monaci@dei.unipd.it)

approssimate è quello che basta nelle applicazioni reali (soprattutto se riferite a problemi di grandi dimensioni); questo è essenzialmente dovuto ad una serie di fattori:

- molti dei parametri in gioco nelle applicazioni reali sono delle stime che possono essere soggette ad errore, per cui non vale la pena di aspettare troppo tempo per avere una soluzione il cui valore (o la cui ammissibilità) è di valutazione incerta;
- spesso si è interessati ad avere una possibile soluzione per il problema in esame al fine di valutare velocemente degli scenari di lavoro (contesti operativi, integrazione di algoritmi di ottimizzazione in Sistemi di Supporto alle Decisioni interattivi);
- spesso si lavora in tempo reale, per cui si vuole avere una "buona" soluzione ammissibile in tempi molto ridotti (secondi di tempo di calcolo);

Questi aspetti spiegano perché, nelle applicazioni reali, sia così diffuso il ricorso a metodi che permettono di trovare delle "buone" soluzioni senza garantire la loro ottimalità ma garantendo un tempo di calcolo relativamente breve: tali metodi prendono il nome di *metodi euristici* (dal greco *heuriskein = scoprire*).

Per la maggior parte dei problemi di ottimizzazione combinatoria è possibile progettare delle *euristiche specifiche* che sfruttano le proprietà del particolare problema in esame e le conoscenze specifiche che derivano dall'esperienza. In effetti, molto spesso, un algoritmo di ottimizzazione si limita a codificare le regole utilizzate per una soluzione manuale del problema, quando disponibili. Ovviamente, la qualità delle soluzioni ottenute dipende dal livello di esperienza che viene incorporato nell'algoritmo: se tale livello è elevato, le soluzioni saranno di buona qualità (comunque non migliori di quelle correntemente prodotte); se il livello è scarso (al limite nullo, come può accadere per uno sviluppatore di algoritmi che ha conoscenze informatiche ma non dello specifico problema) il metodo rischia di limitarsi al "primo metodo ragionevole che ci è venuto in mente".

Negli ultimi anni l'interesse (sia accademico che applicativo) si è rivolto ad approcci euristici di tipo generale le cui prestazioni "sul campo" dominano quasi sempre quelle delle tecniche euristiche specifiche. La letteratura sulla su tali tecniche è ampia e ampliabile, con unico limite la "fantasia" dei ricercatori in questo campo. In effetti sono state proposte le tecniche più svariate e suggestive, al punto di rendere improbo qualsiasi tentativo di classificazione e sistemizzazione. Una possibile (e sindacabile) classificazione è la seguente:

- **euristiche costruttive**: sono applicabili se la soluzione si può ottenere come un sottoinsieme di alcuni elementi. In questo caso si parte da un insieme vuoto e si aggiunge iterativamente un elemento per volta. Se l'elemento viene scelto in base a criteri di ottimalità locale, si realizzano le cosiddette euristiche *greedy* (=ingordo, goloso, avido). Caratteristica essenziale è la progressività nella costruzione della soluzione: ogni aggiunta non viene rimessa in discussione in un secondo momento;

- **metodi metaeuristici**: si tratta di metodologie generali, degli schemi algoritmici concepiti indipendentemente dal problema specifico. Tali metodi definiscono delle componenti e le loro interazioni, al fine di pervenire ad una buona soluzione. Le componenti devono essere specializzate per i singoli problemi. Citiamo, tra le metaeuristiche più note e consolidate, la *Ricerca Locale*, *Simulated Annealing*, *Tabu Search*, *Variable Neighborhood Search*, *Greedy Randomized Adaptive Search Techniques Algoritmi genetici*, *Scatter Search*, *Ant Colony Optmization*, *Swarm Optmization*, *Reti neurali* etc.

- **algoritmi approssimati**: si tratta di metodi euristici a performance garantita. È possibile cioè dimostrare formalmente che, per ogni istanza del problema, la soluzione ottenuta non sarà peggiore dell'ottimo (eventualmente ignoto) oltre una certa percentuale;

- **iper-euristiche**: si tratta di temi al confine con l'intelligenza artificiale e il machine-learning, in cui la ricerca è in fase pionieristica. In questo caso, si mira a definire algoritmi che siano in grado di scoprire dei metodi di ottimizzazione, adattandoli *automaticamente* a diversi problemi.

- **etc. etc. etc.**

La trattazione fornita nel seguito è necessariamente molto schematica e si baserà su degli esempi. Per una ricognizione più approfondita si rimanda *ad esempio* a

> *C. Blum and A. Roli, "Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison", ACM Computer Surveys 35:3, 2003 (p. 268-308)*

articolo a sua volta ricco di ulteriori riferimenti per l'approfondimento di specifiche tecniche.

2 Euristiche costruttive

Le euristiche costruttive determinano (costruiscono) una soluzione ammissibile partendo solo dai dati di ingresso del problema in esame. Caratteristica comune è l'assenza (o la forte limitazione) del backtracking: si parte da una soluzione vuota e si determinano in modo itarativo nuovi elementi da aggiungere ad una soluzione fino ad arrivare ad una soluzione completa (*criterio di espansione*). Si tenga presente che, per alcuni problemi, può essere difficile determinare una soluzione ammissibile: in questo caso l'algoritmo cerca di costruire una soluzione che sia il più possibile ammissibile.

Tra i vari tipi di euristiche costruttive consideriamo gli algoritmi greedy, algoritmi che fanno uso di tecniche di ottimizzazione esatta, algoritmi che semplificano procedure potenzialmente esatte. Si tratta in ogni caso di tecniche la cui complessità computazionale rimane polinomiale.

2.1 Algoritmi greedy

L'idea di questi algoritmi è adottare come *criterio di espansione* basato sulla scelta più conveniente in quel momento (secondo un criterio fortemente locale) compatibilmente con i vincoli del problema: ad ogni iterazione viene aggiunto alla soluzione l'elemento che produce il miglioramento maggiore della funzione obiettivo. Lo schema concettuale di un algoritmo greedy è di questo tipo:

1. inizializza la soluzione S;
2. per ogni scelta da effettuare:
3. prendi la decisione più conveniente, compatibilmente con i vincoli del problema.

La grande diffusione degli algoritmi greedy è essenzialmente dovuta ai seguenti motivi:

- l'algoritmo simula quello che sarebbe il comportamento manuale più intuitivo nella determinazione della soluzione (è spesso il "primo metodo che ci viene in mente");
- l'implementazione dell'algoritmo risulta essere particolarmente semplice;
- il tempo di calcolo richiesto per determinare la soluzione è estremamente ridotto (se vengono implementate in modo efficiente le fasi 2. e 3., relative all'identificazione della prossima scelta da compiere e alla valutazione dell'ammissibilità della scelta);
- questi algoritmi costituiscono la base per tutti gli algoritmi più sofisticati (ad esempio forniscono la soluzione iniziale per algoritmi di ricerca locale, o soluzioni ammissibili nell'ambito di tecniche esatte basate su enumerazione implicita).

In alcuni casi gli algoritmi greedy sono basati su un pre-ordinamento degli elementi (*Dispatching Rule*): gli elementi che definiscono la soluzione vengono considerati secondo tale ordine ed eventualmente inseriti in soluzione. Generalmente i criteri di ordinamento utilizzati prevedono di associare a ciascuna scelta uno "score" che indichi la bontà della mossa, cercando di premiare ad ogni iterazione la mossa che sembra essere la più promettente. L'informazione relativa allo score può essere calcolata all'inizio dell'esecuzione in base ai dati di input. Spesso però lo stesso algoritmo euristico fornisce risultati migliori se il criterio di ordinamento degli elementi viene aggiornato dinamicamente in modo da tener conto delle scelte fatte in precedenza; ovviamente il continuo aggiornamento degli score degli elementi comporterà un aumento nel tempo di calcolo richiesto dall'algoritmo stesso.

Generalmente gli algoritmi greedy sono di tipo *primale*, ossia effettuano delle scelte che rispettino sempre tutti i vincoli. Esistono però anche delle versioni *duali* di tali algoritmi; queste partono da soluzioni non ammissibili e compiono delle scelte mirate al raggiungimento dell'ammissibilità, cercando di non peggiorare troppo il valore della soluzione.

2.2 Algoritmi basati su tecniche esatte di ottimizzazione

In questo caso il criterio di espansione è basato sulla soluzione di un problema di ottimizzazione più semplice del problema originale. Ad esempio, se si dispone di un modello di programmazione linare intera del problema (o di una sua approssimazione) si potrebbe risolvere il rilassamento continuo e utilizzare le informazioni ottenute (funzione obiettivo, valori delle variabili, costi ridotti etc.) per definire gli score utilizzati dal criterio di espansione. La soluzione del problema rilassato potrebbe essere effettuata all'inizio oppure ad ogni iterazione di espansione, una volta fissate le variabili relative agli elementi precedentemente introdotti nella soluzione in costruzione.

Generalmente il tempo di calcolo richiesto per l'esecuzione di questi algoritmi è maggiore di quello richiesto dagli algoritmi greedy, ma le soluzioni prodotte tendono ad essere di qualità decisamente superiore. In effetti, le scelte fatte tengono in qualche modo conto dell'ottimalità globale (e non solo locale) della scelta effettuata ad ogni passo.

2.3 Semplificazione di procedure esatte

Si tratta di algoritmi basati sull'esecuzione parziale di un metodo esatto, ad esempio un algoritmo di tipo enumerativo. La variante più semplice è quella ottenuta facendo terminare un algoritmo branch-and-bound dopo un certo time limit o dopo un prefissato numero di nodi e sfruttando la migliore soluzione ammissibile generata fino a quel momento.

Una variante con maggiori garanzie di trovare soluzioni ammissibili è nota come *beam search* e consiste nel semplificare l'algoritmo branch-and-bound in modo da generare, a partire da ciascun nodo, al massimo k nodi figli (dove k è un parametro da tarare in base al tempo di calcolo a disposizione). La scelta dei k sottoproblemi da generare viene solitamente effettuata valutando, per ciascun potenziale nodo figlio, una valutazione preventiva della bontà delle soluzioni contenute nel corrispondente sottoalbero (ad esempio, ma non solo, il bound) e prendendo i k nodi figli più promettenti. In questo modo si evita l'esplosione combinatoria: ad ogni livello si manterranno (al massimo) k nodi e l'albero del branch-and-bound si riduce ad un *fascio* (=beam) di $n \cdot k$ nodi (se n è il numero di livelli dell'albero) garantendo così una complessità polinomiale,se polinomiale è la procedura di valutazione di ogni nodo. Si noti che, se n è il numero di livelli dell'albero (legato alle dimensioni del problema), b è il numero di nodi figli del nodo generico e k la dimensione del fascio saranno valutati $O(n \cdot k \cdot b)$ nodi. Alla fine si avranno al massimo k nodi foglia corrispondenti a soluzioni tra le quali viene scelta la migliore.

2.4 Esempio: Problema dello zaino binario (KP/0-1)

Nel problema dello zaino si hanno n oggetti, ciascuno con un peso w_j e un profitto p_j, e se ne vogliono selezionare alcuni in modo da massimizzare il profitto senza eccedere la capacità dello zaino c. Un oggetto è "buono" se ha profitto alto e peso basso. Quindi l'algoritmo deve cercare di inserire gli oggetti privilegiando quelli che hanno valori elevati di profitto e valori bassi di peso; un criterio di ordinamento particolarmente efficace

consiste nel considerare gli oggetti secondo valori non crescenti del rapporto profitto su peso.

- **Algoritmo greedy per KP/0-1**

  1. Ordina gli oggetti per valori decrescenti del rapporto $\frac{p_j}{w_j}$.
  2. Inizializza: $S := \emptyset$ (oggetti selezionati),
     $\bar{c} := c$ (capacità residua del contenitore),
     $z := 0$ (valore della soluzione).
  3. **for** $j = 1, \ldots, n$ **do**
  4. **if** $(w_j \leq \bar{c})$ **then**
  5. $S := S \cup \{j\}$, $\bar{c} := \bar{c} - w_j$, $z := z + p_j$.
  6. **endif**
  7. **endfor**

Si noti come il criterio di espansione sia statico (il rapporto) e possa essere valutato una volta per tutte all'inizio dell'algoritmo.

2.5 Esempio: Set Covering

Il problema del *set covering* (SCP) ha in input un insieme N di elementi e un insieme di sottinsiemi di N ($\mathcal{N} \subseteq 2^N$). Ad ogni sottinsieme è associato un costo c_{ij} e si vogliono selezionare i sottinsiemi che coprano tutti gli elementi di N a costo minimo. Indichiamo con a_{ij} un parametro pari a 1 se l'elemento i di N è contenuto nel sottinsieme j di $\mathcal{N}$, 0 altrimenti. Nel problema del set covering, una colonna è "buona" se ha costo basso e copre molti elementi tra quelli ancora scoperti. L'idea di base dell'algoritmo greedy è quindi quella di calcolare lo score di ciascun sottinsieme non ancora inserito nella soluzione in funzione del costo e del numero di elementi aggiuntivi coperti.

- **Algoritmo greedy per SCP**

  1. Inizializza: $S := \emptyset$ (sottinsiemi selezionati),
     $\bar{M} := \emptyset$ (elementi coperti dai sottinsiemi selezionati),
     $z := 0$ (valore della soluzione).
  2. se $\bar{M} = M$ ($\Leftrightarrow$ tutte gli elementi coperti), STOP;
  3. determina il sottinsieme $j \notin S$ con rapporto $\dfrac{c_j}{\sum_{i \in M \setminus \bar{M}} a_{ij}}$ *minimo*;
  4. poni $S := S \cup \{j\}$, $\bar{M} := \bar{M} \cup \{i \notin \bar{M} : a_{ij} = 1\}$, $z := z + c_j$ e vai a 2.

Si noti come, in questo caso, la valutazione degli score sia dinamica, essendo legata non solo al sottinsieme in esame, ma anche alle scelte precedentemente effuettuare secondo il criterio di espansione, che modifica il numero di elementi *aggiuntivi* coperti.

- **Algoritmo basato su rilassamento continuo**[5]

  1. Inizializza: $S := \emptyset$ (sottinsiemi selezionati),
     $\bar{M} := \emptyset$ (elementi coperti dai sottinsiemi selezionati),
     $z := 0$ (valore della soluzione).
  2. se $\bar{M} = M$ ($\Leftrightarrow$ tutti gli elementi coperti), STOP;
  3. risolvi il rilassamento continuo del SCP con i vincoli ulteriori $x_j = 1$ ($j \in S$), e sia x^* la corrispondente soluzione;
  4. determina la colonna $j \notin S$ con x^*_j *massimo* (più prossimo a 1);
  5. poni $S := S \cup \{j\}$, $\bar{M} := \bar{M} \cup \{i \notin \bar{M} : a_{ij} = 1\}$, $z := z + c_j$ e vai a 2.

3 Algoritmi di Ricerca Locale

Dato un problema di ottimizzazione P definito da una funzione obiettivo f e da una regione ammissibile F, un *intorno* è una applicazione

$$N : s \rightarrow N(s)$$

che ad ogni punto s della regione ammissibile associa un sottoinsieme $N(s)$ della regione ammissibile F.

L'idea di base degli algoritmi di ricerca locale è quella di definire una soluzione iniziale (*soluzione corrente*) e cercare di migliorarla esplorando un intorno (opportunamente definito) di questa soluzione. Se l'ottimizzazione sull'intorno della soluzione corrente produce una soluzione migliorante il procedimento viene ripetuto partendo, come soluzione corrente, dalla soluzione appena determinata. L'algoritmo termina quando non è più possibile trovare delle soluzioni miglioranti nell'intorno della soluzione corrente, oppure quando è stata determinata la soluzione ottima (valore della soluzione uguale a qualche bound); in alternativa si può far terminare l'algoritmo dopo un prefissato tempo di calcolo o numero di iterazioni.

[5]Il SCP può essere modellato come segue:

$$\begin{aligned} \min & \sum_{j \in \mathcal{N}} c_j x_j & & \\ s.t. & \sum_{j \in \mathcal{N}} a_{ij} x_j & \geq 1 & \quad \forall\, i \in N \\ & x_j & \in \{0,1\} & \quad \forall\, j \in \mathcal{N} \end{aligned}$$

Lo schema concettuale di base di un algoritmo di ricerca locale è il seguente (problema di minimo):

1. Determina una soluzione corrente iniziale x;
2. **if** $(\exists\ x' \in N(x) : f(x') < f(x))$ **then**
3. $\quad x := x'$
4. $\quad$ **if** $(f(x) = LB)$ return(x) $(\Leftrightarrow x$ soluzione ottima$)$
5. $\quad$ goto 2
6. **endif**
7. return(x) $(\Leftrightarrow x$ ottimo locale$)$

Lo schema mostra che il metodo è estremamente generale e può essere applicato per risolvere problemi molto diversi tra loro. Per particolarizzare l'algoritmo bisogna specificare nel dettaglio le scelte da compiere nei passi 1. (determinazione della soluzione iniziale) e 2. (definizione dell'intorno e criterio di esplorazione).

Per quel che riguarda la definizione della soluzione iniziale, si potrebbe pensare che sia meglio partire da una soluzione "buona" piuttosto che da una soluzione "scadente"; in realtà non esiste nessun risultato teorico che confermi questa sensazione. Generalmente quello che viene fatto è eseguire più volte l'algoritmo di ricerca locale partendo da soluzioni differenti (generate casualmente) in modo da poter esplorare zone differenti della regione ammissibile.

Per quel che riguarda la definizione dell'intorno, questa dipende dal tipo di mosse che si vogliono effettuare. È ovvio che se, per qualunque soluzione ammissibile, l'intorno coincide con l'intera regione ammissibile, la procedura restituisce la soluzione ottima del problema; questo però avviene per enumerazione di tutte le soluzioni della regione ammissibile (cosa ovviamente impossibile - o quantomeno poco pratica - nelle applicazioni reali). In generale, maggiore è la dimensione dell'intorno e più alta è la probabilità di sfuggire dagli ottimi locali; per questo si vorrebbe, per quanto possibile, avere intorni che contengano molte soluzioni. Ovviamente si ha anche l'esigenza di essere in grado di esplorare l'intorno in modo efficiente; spesso però queste due esigenze sono contrapposte, per cui bisogna trovare un compromesso ingegneristico tra la dimensione dell'intorno e la "bontà" degli ottimi locali che ne deriveranno.
Altra caratteristica che si vorrebbe per l'intorno è il fatto di essere *connesso*; questo vuol dire che qualunque sia la soluzione di partenza, è sempre possibile raggiungere qualunque soluzione ammissibile (compresa quella ottima) tramite una opportuna sequenza di mosse; quindi se l'intorno è connesso, un algoritmo "fortunato" troverebbe sempre la soluzione ottima del problema.

A volte può essere utile definire l'intorno implicitamente, come l'insieme di soluzioni raggiungibili, a partire dalla soluzione corrente, tramite mosse del tipo scelto

$$N(s) := \{\bar{s} \in F : \bar{s} = m(s) \text{ per qualche mossa } m\}$$

In questa ottica è possibile introdurre il concetto di *distanza* tra soluzioni e definire l'intorno di una soluzione come l'insieme di tutte le soluzioni la cui distanza dalla soluzione corrente non supera una certa soglia. Nel caso in cui una soluzione possa essere rappresentata da vettori binari a n componenti (come accade nei problemi di programmazione binaria), si può utilizzare la distanza di Hamming, che consiste nel numero di componenti che sono diverse tra le due soluzioni s_1 e s_2

$$d_H(s_1, s_2) = \sum_{j=1}^{n} |s_1(j) - s_2(j)|$$

Infine, per quel che riguarda la strategia di esplorazione dell'intorno, è possibile effettuare il primo scambio migliorante (first improvement) oppure il migliore scambio ammissibile (best improvement). Talvolta, al fine di introdurre della casualità nell'algoritmo, si preferisce determinare le k migliori soluzioni dell'intorno e scegliere casualmente una di queste soluzioni come prossima soluzione di partenza.

Per alcuni problemi può essere difficile trovare una soluzione corrente ammissibile; in tal caso l'algoritmo di ricerca locale può partire da una soluzione non ammissibile e cercare di spostarsi verso soluzioni ammissibili. Allo stesso modo, è possibile che l'algoritmo passi, nelle iterazioni intermedie, per soluzioni non ammissibili; per cercare di evitare le soluzioni non ammissibili si cerca di inserire nella funzione obiettivo un termine di penalità che tenga conto del fatto che alcuni vincoli non sono soddisfatti. Quindi la funzione di valutazione utilizzata per esplorare l'intorno non necessariamente coincide con la funzione obiettivo originale.

Gli algoritmi di ricerca locale sono concettualmente molto semplici ma spesso forniscono buoni risultati per alcuni problemi di ottimizzazione combinatoria. Questo accade soprattutto per quei problemi che hanno una struttura che definisce implicitamente un intorno, ad esempio perchè il numero di elementi in soluzione è fissato (come accade per il TSP). In tal caso, un intorno può essere definito come l'insieme di soluzioni ottenute dalla soluzione corrente sostituendo un elemento in soluzione con un altro elemento non in soluzione.

3.1 Esempio: KP-01

Possibili intorni di una soluzione S sono ottenibili nel seguente modo:

- provando ad aggiungere un oggetto alla soluzione;
- rimuovendo un oggetto in soluzione in favore di un oggetto non in soluzione;

Un corrispondente algoritmo di ricerca locale potrebbe allora avere questa forma:

1. inizializza la soluzione iniziale S;
2. considera tutte le soluzioni ammissibili della forma:
   - $S \cup \{j\}$ (con $j \notin S$)

- $S \cup \{j\} \setminus \{i\}$ (con $j \notin S$ e $i \in S$)

indicando con R la migliore di queste soluzioni;

3. se $\sum_{j \in R} p_j \leq \sum_{j \in S} p_j$, STOP (ottimo locale);
4. poni $S := R$ e vai al punto 2.

Es: $n = 6$, $c = 85$,

$$(p_j) = (\ 110,\ 150,\ 70,\ 80,\ 30,\ 5\ )$$
$$(w_j) = (\ 40,\ 60,\ 30,\ 40,\ 20,\ 5\ )$$

Partendo dalla soluzione $S = \{3\}$ di valore pari a 70, le soluzioni considerate sono:

- $S = \{1, 3\}$ (val.= 180, aggiunto oggetto 1);
- $S = \{1, 4\}$ (val.= 190, aggiunto oggetto 4, eliminato oggetto 3);
- $S = \{1, 4, 6\}$ (val.= 195, aggiunto oggetto 6): ottimo locale.

3.2 Esempio: Set Covering

Possibili intorni di una soluzione S sono ottenibili nel seguente modo:

- provando ad eliminare una colonna dalla soluzione;
- rimuovendo una colonna in soluzione ed inserendo una colonna non in soluzione;

Un algoritmo di ricerca locale ha allora questa forma:

1. inizializza la soluzione iniziale S;
2. considera tutte le soluzioni ammissibili della forma:
   - $S \setminus \{j\}$ (con $j \in S$)
   - $S \setminus \{j\} \cup \{i\}$ (con $j \in S$ e $i \notin S$)

   indicando con R la migliore di queste soluzioni;
3. se $\sum_{j \in R} c_j \geq \sum_{j \in S} c_j$, STOP (ottimo locale);
4. poni $S := R$ e vai al punto 2.