Metodi e Modelli per l'Ottimizzazione Combinatoria

Metodi euristici di ottimizzazione combinatoria

L. De Giovanni

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 soluzioni ammissibili. L'applicazione di *metodi esatti* non è sempre possibile, essenzialmente per due motivi concomitanti: la complessità intrinseca del problema (ad esempio, particolari problemi NP-Hard) e il tempo a disposione per generare la soluzione. Premettiamo che la rinuncia all'utilizzo di metodi esatti deve essere una scelta ponderata, basata non solo sulla complessità teorica del problema (non basta, insomma, che un problema sia NP-hard), ma anche sulla effettiva praticabilità di metodi esatti (da ricercarsi eventualmente nella letteratura), al tempo di elaborazione disponibile, alle dimensioni dei problemi da risolvere. Ad esempio, non si dovrebbe mai rinunciare al tentativo di formulare il problema con un modello di programmazione matematica (in particolare lineare intera mista), non solo come strumento di analisi del problema, ma anche come strumento operativo, visto che la crescente potenza dei solver potrebbe rendere praticabile la sua implementazione e la sua risoluzione in tempi accettabili.

Tuttavia, spesso i metodi esatti si basano sulla formulazione di un modello di programmazione matematica del problema di ottimizzazione e la complessità del problema potrebbe rendere praticamente impossibile pervenire ad una formulazione 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 esatti: ad esempio, i metodi basati su enumerazione implicita non danno garanzia sul tempo necessario per il loro completamento (risolvere un problema 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 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 a *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;
- in alcuni casi 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 *eurìskein = 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 difficilmente 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 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 sistematizzazione universalmente accettata. 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. Ad esempio, 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: in linea di principio, ogni aggiunta non viene rimessa in discussione in un secondo momento;

- **metodi metaeuristici**: si tratta di metodologie generali, di 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 Optimization*, *Swarm Optimization*, *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 (che spesso, però, è piuttosto elevata);

- **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)*

L'articolo è 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 iterativo nuovi elementi da aggiungere ad una soluzione fino ad arrivare ad una soluzione completa (*criterio di espansione*).

*In parte, adattamento dalle dispense di Michele Monaci, Dipartimento di Ingegneria dell'Informazione, Università di Padova (monaci@dei.unipd.it)

Tra i vari tipi di euristiche costruttive consideriamo gli algoritmi *greedy*, gli algoritmi che fanno uso di tecniche di ottimizzazione esatta, e gli 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 un *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 forniscono blocchi da integrare in 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 sfruttano un 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 una volta per tutte all'inizio dell'esecuzione in base ai dati di input (pre-ordinamento). 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.

Per provare a ottenere delle soluzioni diverse, e possibilmente migliori, usando la stessa procedura, si può iterare l'algoritmo utilizzando ogni volta un ordinamento diverso, ottenuto da una *randomizzazione* della dispatching rule. Ad esempio, lo score potrebbe essere corretto con una componente casuale, in modo da avere la possibilità di scegliere, ad ogni passo, non l'elemento migliore, ma un elemento "abbastanza buono": in questo modo si potrebbe rendere meno miope l'algoritmo e conservare alcuni elementi per passi successivi, quando le scelte diventano maggiormente critiche. Oppure, ad ogni passo, si potrebbe considerare la scelta casuale tra i migliori n elementi residui.

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, applicate a problemi per cui può essere difficile determinare una soluzione ammissibile: queste partono da soluzioni non ammissibili e cercano di costruire una soluzione ammissibile, compiendo delle scelte mirate a ridurre il grado di inammissibilità, 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, prendendo delle decisioni con criteri *greedy* in uno schema potenzialmente esatto. 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 attraverso una visita breath first *parziale* dell'albero. Per ogni nodo, si generano tutti i b nodi figli potenziali ma, per ciascun livello, si sviluppano (cioè si fa branching su) 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 sviluppare viene solitamente effettuata associando a ciascun potenziale nodo figlio una valutazione preventiva della bontà delle soluzioni contenute nel corrispondente sottoalbero (ad esempio, ma non solo, il bound, o una valutazione rapida di una possibile soluzione del sottoalbero attraverso una procedura di completamento greedy, o una loro somma pesata etc.) e prendendo i k nodi figli più promettenti del livello corrente. 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 a 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. Dalla formula citata, fissato k, è noto il numero di nodi, e potendo stimare il tempo necessario per effettuare la valutazione di un nodo, si può predeterminare il tempo di esecuzione complessivo di una beam search. Oppure, se è noto il tempo massimo a disposizione, è possibile dimensionare k al valore massimo che permetta di completare la ricerca di tutti i nodi nel tempo previsto.

Nella forma base, la tecnica beam search non prevede backtracking (non è possibile tornare indietro, una volta effettuate delle scelte dei nodi da sviluppare): per questo motivo è stata inclusa in questa sezione sulle euristiche costruttive, anche se il fatto di dover definire le varie componenti in modo specifico per ciascun problema all'interno di un *framework* ben definito rende questa tecnica assimilabile a una metaeuristica. In effetti, il confine (come abbiamo già detto) è labile e, per la beam search, esistono delle varianti, come ad esempio la *recovery beam search* dove si gestisce il backtracking, consentendo, se ci si accorge che qualche sottoalbero a un certo livello non è "promettente", di tornare a un livello precedente (in questo caso, però si potrebbe perdere la polinomialità delle procedura).

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 W. 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),
     $\bar{W} := W$ (capacità residua del contenitore),
     $z := 0$ (valore della soluzione).

3. **for** $j = 1, \ldots, n$ **do**
4. **if** $(w_j \leq \bar{W})$ **then**
5. $S := S \cup \{j\}, \ \bar{W} := \bar{W} - w_j, \ z := z + p_j.$
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.

- **Beam search per KP/0-1**

  Consideriamo l'applicazione al seguente problema:

$$\begin{aligned} \max \quad & 30x_1 + 36x_2 + 15x_3 + 11x_4 + 5x_5 + 3x_6 \\ & 9x_1 + 12x_2 + 6x_3 + 5x_4 + 3x_5 + 2x_6 \leq 17 \\ & x_i \in \{0, 1\} \quad \forall\, i = 1 \ldots 6 \end{aligned}$$

  Il risultato è rappresentato in Figura 1 e considera le seguenti scelte implementative:

  - il branching è binario ($b = 2$): al livello i dell'albero, fissa a 0 a a 1 la variabile x_i, secondo l'ordine stabilito dai valori decrescenti del rapporto $\frac{p_j}{w_j}$ (nel nostro caso le variabili sono già ordinate). Di conseguenza, il numero di livelli è pari al numero n di variabili (nel nostro caso 6).
  - $k = 2$;
  - la valutazione di ogni nodo si ottiene applicando l'algoritmo euristico di cui sopra (di complessità $O(n)$ una volta ordinate le variabili) tenendo conto dei valori delle variabili fissati ai livelli precedenti. Si tratta quindi di valutare velocemente, ad ogni nodo un lower bound (soluzione ammissibile) e scegliere, a ogni livello, i $k = 2$ nodi con lower bound più elevato.

2.5 Esempio: Set Covering

Il problema del *set covering* (SCP) ha in input un insieme M di elementi e un insieme di sottoinsiemi di M ($\mathcal{M} \subseteq 2^M$). Ad ogni sottoinsieme $j \in \mathcal{M}$ è associato un costo c_j e si vogliono selezionare i sottoinsiemi che coprano tutti gli elementi di M a costo minimo. Indichiamo con a_{ij} un parametro pari a 1 se l'elemento i di M è contenuto nel sottoinsieme j di $\mathcal{M}$, 0 altrimenti. Nel problema del set covering, un sottoinsieme è "buono" 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 sottoinsieme non ancora inserito nella soluzione in funzione del costo e del numero di elementi aggiuntivi coperti.

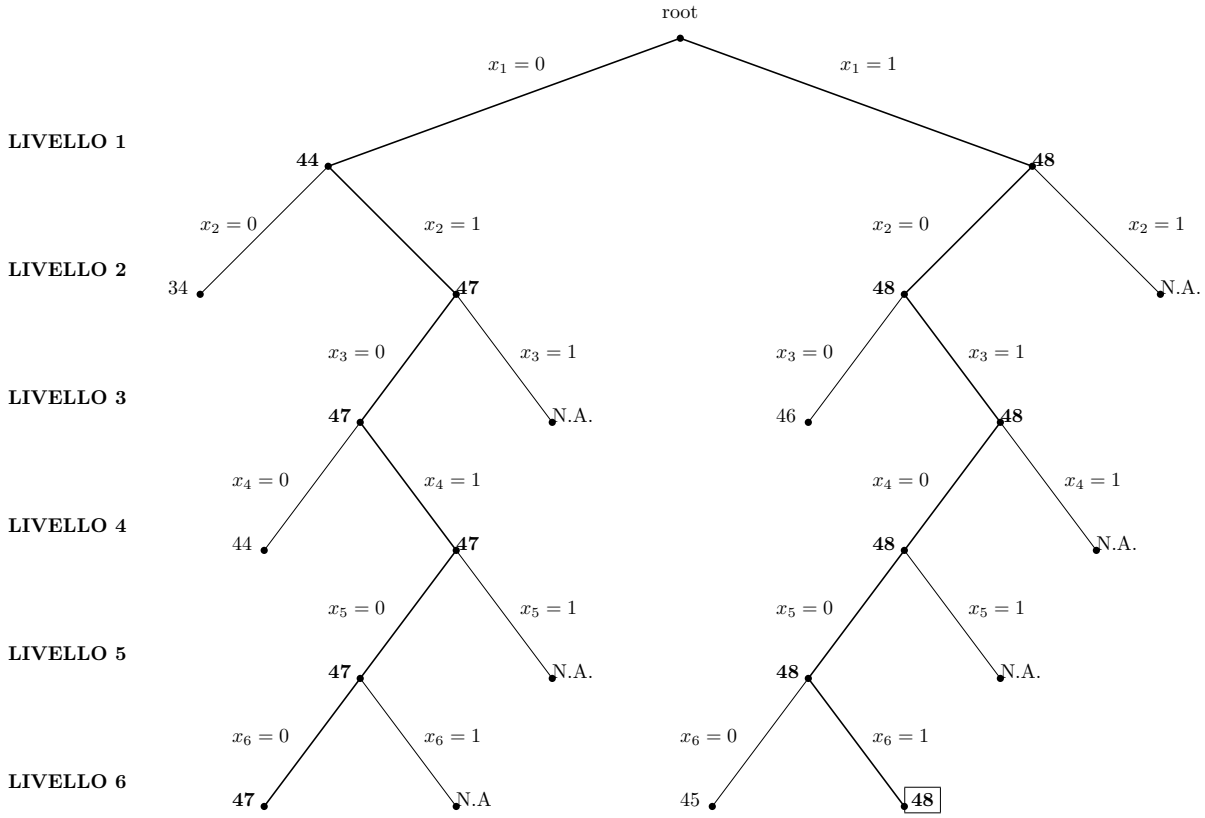


Figure 1: Albero di ricerca della beam search per un problema KP/0-1.

- **Algoritmo greedy per SCP**

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

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

- **Algoritmo basato su rilassamento continuo**

  Il SCP può essere modellato come segue:

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

  Tale modello può essere sfruttato dalla seguente procedura euristica:

  1. Inizializza: $S := \emptyset$ (sottoinsiemi selezionati),
     $\bar{M} := \emptyset$ (elementi coperti dai sottoinsiemi 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 $S := S \cup \{j\}$, $\bar{M} := \bar{M} \cup \{i : a_{ij} = 1\}$, $z := z + c_j$ e vai a 2.

3 Ricerca di vicinato: la ricerca locale

Dato un problema di ottimizzazione P definito da una funzione obiettivo f e da una regione ammissibile X, un *intorno* o *vicinato* (*neighborhood*) è una applicazione

$$N : s \to N(s)$$

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

L'idea di base della ricerca di vicinato (*neighborhood search*) 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.

La versione base della ricerca di vicinato è la *ricerca locale*: l'algoritmo termina quando non è più possibile trovare delle soluzioni miglioranti nell'intorno della soluzione corrente (tale soluzione rappresenta un *ottimo locale*), oppure quando è stata determinata una soluzione con valore della funzione obiettivo uguale a qualche bound (tale soluzione è ottima); in alternativa si può far terminare l'algoritmo dopo un prefissato tempo di calcolo o numero di iterazioni e restituire la migliore soluzione trovata fino a quel momento (corrispondente, per la ricerca locale, con la soluzione corrente).

Lo schema concettuale di base di un algoritmo di ricerca locale è il seguente (problema di minimo con a disposizione un valore LB di un lower bound per la funzione obiettivo):

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) $(x$ è una soluzione ottima$)$
5. $\quad$ goto 2
6. **endif**
7. return(x) $(x$ è un ottimo locale$)$

Ribadiamo che, in generale, la ricerca locale garantisce di trovare solo degli *ottimi locali*, ossia delle soluzioni *nel cui intorno* non ci siano soluzioni miglioranti.

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:

- come si determina la soluzione iniziale;

- la rappresentazione delle soluzioni, che è alla base dei successivi elementi;
- l'applicazione che, a partire da una soluzione, ne genera il vicinato (*mosse*);
- la funzione di valutazione delle soluzioni;
- strategia di esplorazione del vicinato.

Questi elementi sono alla base della ricerca di vicinato in generale, come vedremo nelle sezioni successive, e non solo della ricerca locale.

3.1 Soluzione iniziale

La soluzione iniziale potrebbe essere generata casualmente, anche se, di solito, si preferisce partire da una buona soluzione iniziale, determinata attraverso una procedura euristica (ad esempio un'euristica costruttiva, o l'implementazione dei metodi correntemente adottati), o dalla soluzione correntemente applicata dal "committente" (se disponibile). In effetti, non esiste alcun risultato teorico o evidenza pratica secondo i quali, in generale, sia meglio partire dalla migliore soluzione possibile: di solito, si applica la ricerca locale a partire da soluzioni diverse, ottenute da diverse euristiche, oppure casualmente, o ancora tramite la randomizzazione di tecniche euristiche. Questo permette di esplorare zone diverse della regione ammissibile e ottenere diverse soluzioni (diversi ottimi locali) tra le quali restituire la migliore.

3.2 Rappresentazione della soluzione

La rappresentazione delle soluzioni codifica le caratteristiche delle soluzioni stesse e fornisce la base "concreta" per le elaborazioni che permettono di esplorare lo spazio delle soluzioni. La scelta della rappresentazione non è univoca e, come vedremo, influenza anche pesantemente il progetto degli altri elementi.

> Ad esempio, le soluzioni del problema KP/0-1 (dove, ricordiamo, si deve decidere quali tra n oggetti, ciascuno con peso w_i e profitto p_i, caricare in uno zaino di capienza W) possono essere rappresentate come:
>
> 1. lista degli oggetti nello zaino;
> 2. vettore caratteristico dell'insieme degli oggetti nello zaino, cioè un vettore binario di dimensione n (numero di oggetti);
> 3. sequenza ordinata degli n oggetti.

Si noti come la rappresentazione della soluzione non sempre indica in modo diretto quale sia la soluzione codificata. Ad esempio, nella terza opzione per il problema KP/0-1, non è immediato capire quale sia la soluzione in termini di oggetti da caricare. Si rende quindi necessaria una *decodifica* (più o meno immediata) che permetta di passare dalla rappresentazione alla soluzione.

Sempre per tornare all'esempio dello zaino, la decodifica per le prime due rappresentazioni è immediata. Per la terza rappresentazione, la decodifica consiste nello scorrere la sequenza ordinata e inserire nello zaino gli oggetti fino a saturarne la capienza (applicazione dell'euristica costruttiva presentata nella sezione precedente, sulla base di un ordinamento diverso da quello greedy).

3.3 Definizione del vicinato

Data una soluzione (detta *centro* del vicinato), le soluzioni che fanno parte del vicinato sono ottenute applicando delle *mosse* che perturbano la soluzione centrale, trasformandola in una *soluzione vicina*.

Per il problema KP/0-1, possibili mosse sono: inserimento di un oggetto nello zaino, oppure lo scambio di un oggetto nello zaino con uno fuori.

3.3.1 Dimensione del vicinato

Generalmente, si considerano mosse che determinano delle perturbazioni limitate delle soluzioni, in modo da potersi muovere in modo "controllato" nello spazio delle soluzioni. In questo modo, la *dimensione* del vicinato, cioè il numero di soluzioni vicine, rimane limitato, permettendone una valutazione rapida. Chiaramente, deve essere raggiunto un compromesso tra la dimensione (e quindi i tempi di esplorazione) del vicinato e l'efficacia del vicinato stesso, cioè la sua capacità di contenere soluzioni miglioranti e, quindi, di raggiungere ottimi locali migliori: al limite, se definissimo delle mosse molto complesse che includano nel vicinato tutte le soluzioni ammissibili, saremmo sicuri di ottenere un miglioramento (anzi, di raggiungere la soluzione ottima, in una mossa) al prezzo di dover ricorrere all'enumerazione, con tempi esponenziali, cioè improponibili. Quindi, ogni volta che si progetta un vicinato, bisogna porsi una prima domanda:

1) *quali sono le dimensioni del vicinato (il numero di vicini generati dalle mosse)?*

   Nell'esempio, le dimensioni dei vicinati di aggiunta di un oggetto nello zaino e di scambio di oggetti inseriti con oggetti non inseriti sono, rispettivamente, $O(n)$ e $O(n^2)$.

A questo proposito, alcuni autori parlano di *forza* di un vicinato: un vicinato è tanto più forte, quanto più un algoritmo di ricerca locale basato su di esso è in grado di produrre buone soluzioni indipendentemente dalla qualità delle soluzioni di partenza. Senza introdurre concetti di *landscape analysis*, che sono al di fuori degli obiettivi del corso, facciamo osservare che la ricerca locale trova ottimi locali e, in particolare, che *il concetto di ottimo locale dipende (anche) da come si definisce il vicinato*: una soluzione potrebbe non avere soluzioni miglioranti in un determinato vicinato ma, se si considera un vicinato diverso, questo potrebbe contenere delle soluzioni miglioranti. In altre parole, le caratteristiche di "convessità" dello spazio delle soluzioni dipendono, in prima analisi dal problema, dalla funzione obiettivo e dalla definizione del vicinato.

Nel KP/0-1, un vicinato più forte potrebbe ottenersi scambiando due oggetti nello zaino con due oggetti fuori, al prezzo, in questo caso, di una maggiore dimensione $O(n^4)$.

Si noti che, disponendo di un vicinato forte, si potrebbe partire direttamente con la ricerca locale, senza perdere tempo nella generazione di buone soluzioni iniziali. Anzi, conviene generare molte soluzioni iniziali in maniera casuale, per avere un buon campionamento della regione ammissibile e aumentare la possibilità di trovare buoni ottimi locali, se non un ottimo globale.

Altra caratteristica che si vorrebbe per il vicinato è il fatto di essere *connesso*: data una qualsiasi soluzione, è sempre possibile raggiungere qualsiasi altra soluzione ammissibile (compresa quindi quella ottima) tramite un'opportuna sequenza di mosse. I vicinati connessi sono preferibili e, di solito, più efficaci, perché non precludono a priori l'esplorabilità dello spazio delle soluzioni ammissibili.

Per KP/0-1, il vicinato di aggiunta è chiaramente non connesso (le soluzioni raggiungibili sono solo quelle che contengono gli oggetti nella soluzione di partenza). Anche il secondo vicinato è non connesso (sono raggiungibili solo soluzioni con un numero di oggetti nello zaino pari a quello della soluzione di partenza). Un vicinato connesso sarebbe quello che comprende due tipi di mosse: aggiunta di un oggetto nello zaino ***e*** eliminazione di un oggetto dallo zaino (si noti però che, in un contesto di ricerca locale, le mosse di eliminazione non sarebbero mai selezionate perché non miglioranti).

3.3.2 Influenza della rappresentazione della soluzione

La definizione di vicinato è strettamente legata all'adozione di una particolare rappresentazione della soluzione.

Nell'esempio, le mosse descritte presuppongono la scelta della rappresentazione 1. La definizione delle stesse mosse è facilmente ottenibile con la rappresentazione 2: trasformare uno 0 in 1 (aggiunta), scambio tra uno 0 e un 1 (scambio di oggetti), trasformare un 1 in 0 (eliminazione).

Con la rappresentazione 3, è più naturale definire altre mosse, visto che una soluzione vicina è determinata dalla modifica della sequenza del centro del vicinato. La mossa potrebbe essere lo scambio di due oggetti nella sequenza: ad esempio, se $n = 7$ e la soluzione centro è $1-2-3-4-5-6-7$, possibili vicini sono $1-6-3-4-5-2-7$ (ottenuto scambiando 2 e 6) oppure $5-2-3-4-1-6-7$ (scambio di 1 con 5). La dimensione di questo vicinato è $O(n^2)$ ed è possibile mostrare che il vicinato è connesso (almeno rispetto allo spazio delle soluzioni massimali, cioè quelle cui non è possibile aggiungere oggetti lasciandole ammissibili).

3.3.3 Valutazione e complessità del vicinato

Un altro importante aspetto da considerare nel progetto del vicinato è legato all'*efficienza* della sua esplorazione, cioè della valutazione delle soluzioni che ne fanno parte. Infatti, uno dei fattori che determinano il successo delle tecniche basate su ricerca di vicinato è la capacità di valutare molte soluzioni molto velocemente. Chiaramente, il tempo per l'esplorazione di un vicinato dipende non solo, come abbiamo visto, dalla dimensione, ma anche dalla complessità computazionale della valutazione di un singolo vicino. A questo proposito, è sempre importante considerare la possibilità di una *valutazione incrementale* che sfrutti le informazioni del centro del vicinato. Quindi, una seconda domanda da porsi nella progettazione di un vicinato è:

2) *Qual è la complessità computazionale della valutazione (possibilmente incrementale) delle soluzioni vicine?*

La possibilità di una valutazione incrementale efficiente è legata al grado di perturbazione introdotto da una mossa: per questo motivo si tende a privilegiare vicinati determinati da mosse semplici (spesso meno forti ma di rapida valutazione), rispetto a mosse che determinano perturbazioni rilevanti (vicinati più forti ma che, oltre ad avere maggiori dimensioni, richiedono una valutazione meno efficiente).

In totale, la *complessità del vicinato* è data dal prodotto delle dimensioni del vicinato per la complessità di valutazione di ciascun vicino.

> Nell'esempio KP/0-1, la valutazione di ogni vicino derivato dalle mosse di aggiunta o eliminazione di oggetti è ottenibile in tempo costante, a partire dalla valutazione del centro del vicinato (basta aggiungere e/o sottrarre il valore dell'oggetto coinvolto): la complessità di questi vicinati è quindi $O(1 \cdot n) = O(n)$. Per il vicinato di scambio di oggetti inclusi/non inclusi nella soluzione, come definito per le rappresentazioni 1 e 2, il tempo di valutazione a partire dal centro rimane costante (basta sottrarre/aggiungere il valore dei due oggetti coinvolti), determinando una complessità del vicinato $O(1 \cdot n^2) = O(n^2)$. Se invece si considera la rappresentazione 3 e il vicinato di scambio di due oggetti nella sequenza, la valutazione di una soluzione vicina non può essere fatta in modo incrementale (se non parzialmente) e la complessità rimane (comunque) $O(n)$: la complessità del vicinato è quindi $O(n^3)$.

3.4 Funzione di valutazione delle soluzioni

Di solito, ciascuna soluzione è valutata calcolando il corrispondente valore della funzione obiettivo. Potrebbe però essere conveniente utilizzare una diversa funzione obiettivo. Ad esempio, in alcuni problemi fortemente vincolati, può essere difficile trovare una soluzione di partenza ammissibile. In tal caso l'algoritmo di ricerca locale può partire da una soluzione non ammissibile e cercare di spostarsi verso soluzioni ammissibili, utilizzando come funzione di valutazione (da minimizzare) una misura del livello di non ammissibilità, con l'obiettivo di arrivare a 0. Analogamente, molte soluzioni nel vicinato potrebbero essere non ammissibili e, pertanto, sarebbero, in linea di principio, escluse. Questo potrebbe

limitare fortemente le caratteristiche di connessione del vicinato e, pertanto, può essere preferibile consentire che l'algoritmo passi, nelle iterazioni intermedie, per soluzioni non ammissibili; per cercare di evitare le soluzioni non ammissibili, la funzione di valutazione combina la funzione obiettivo con un termine di penalità che tenga conto del fatto che alcuni vincoli non sono soddisfatti.

> Nell'esempio KP/0-1, abbiamo visto come un vicinato determinato dalle mosse di aggiunta e eliminazione sia connesso solo in teoria, visto che la ricerca locale escluderebbe la scelta di soluzioni ottenute per eliminazione di oggetti, che, comunque, peggiorano il valore della funzione obiettivo. Si potrebbe quindi utilizzare una funzione di valutazione che combina (ad esempio come somma pesata) due componenti: il valore della funzione obiettivo e il livello di inammissibilità della soluzione: se X è l'insieme di oggetti nello zaino previsti da una soluzione la funzione di valutazione $\tilde{f}$ diventa:
>
> $$\tilde{f}(X) = \alpha \sum_{i \in X} p_i - \beta \max \left\{ 0, \sum_{i \in X} w_i - W \right\} \qquad (\alpha, \beta > 0).$$
>
> In questo modo, una volta saturata la capienza dello zaino con mosse di aggiunta, la ricerca locale sarà in grado di proseguire se una buona calibratura di pesi α e β permette di sforare momentaneamente la capienza dello zaino per poi arrivare, con successive eliminazioni, ad una soluzione migliore.

3.5 Strategie di esplorazione

Secondo lo schema di base della ricerca locale di base presentato sopra, il passaggio da una soluzione corrente a un'altra avviene quando si trova nel vicinato *una* soluzione migliorante. La scelta della soluzione migliorante dipende dalla *strategia di esplorazione* adottata. Le scelte *classiche* sono le seguenti:

- *first improvement*: l'esplorazione del vicinato termina appena si trova il primo vicino migliorante, che quindi viene scelto come nuova soluzione corrente;
- *steepest descent* o *best improvement*: si esplora comunque tutto il vicinato e si passa al vicino che garantisce il massimo miglioramento.

Sono possibili tecniche alternative, alcune delle quali, per inserire *casualità* nell'algoritmo, determinano le k soluzioni del vicinato che garantiscono i più alti miglioramenti e scelgono casualmente una di queste soluzioni come prossima soluzione corrente. L'esecuzione ripetuta di tale implementazione della ricerca locale permette di trovare ottimi locali diversi tra i quali si sceglie il migliore. Un'altra possibilità è quella di *memorizzare* alcuni dei migliori vicini non visitati e utilizzarli, alla fine della prima discesa, come soluzioni iniziali di una nuova ricerca locale che potrebbe portare a ottimi locali diversi.

3.6 Applicazione della ricerca locale al problema del commesso viaggiatore (Traveling Salesman Problem - TSP)

A titolo esemplificativo, illustriamo alcune possibili scelte implementative per definire un algoritmo di ricerca locale per uno dei più noti problemi di ottimizzazione combinatoria. Il problema del commesso viaggiatore (Travelling Salesman Problem - TSP) è un problema di ottimizzazione definito su un grafo diretto (TSP asimmetrico) o non diretto (TSP simmetrico). Nel caso di un grafo diretto $D = (N, A)$, si considerano definiti dei costi c_{ij} associati ad ogni arco $(i, j) \in A$. Nel caso di un grafo non diretto $G = (V, E)$, si considerano dei costi c_e associati a ciascuno spigolo $e \in E$. Il TSP consiste nel determinare, se esiste, un *ciclo hamiltoniano* di costo minimo sul grafo. Ricordiamo che un ciclo hamiltoniano è un ciclo che visita tutti i nodi del grafo una ed una sola volta. Si fa notare che possiamo supporre, senza perdita di generalità, che il grafo sia completo, ossia che contenga archi / spigoli per ogni coppia ordinata / non ordinata di nodi: basta considerare, per ogni coppia di nodi, un arco / spigolo con costo pari a quello del cammino minino, sul grafo, tra i due nodi. Il TSP è un problema $\mathcal{NP} - hard$ e pertanto la sua soluzione esatta richiede degli algoritmi a complessità "esponenziale" (a meno che $\mathcal{P} = \mathcal{NP}$). In ogni caso, come già evidenziato, non sempre problemi $\mathcal{NP} - hard$ devono essere trattati con metodi euristici o metaeuristici: per il TSP sono stati definiti metodi esatti che riescono a trovare la soluzione ottima per problemi con anche centinaia di nodi in pochi secondi, fino a trattare problemi con decine di migliaia di nodi, anche se con tempi elevati. I metodi esatti avanzati sono principalmente basati su sofisticate tecniche di branch-and-cut, anche se semplici modelli implementati con solver standard sono sufficienti per dimensioni ridotte (vedi il sito http://www.tsp.gatech.edu/index.html, dove sono anche disponibili dei modelli e le librerie *Concorde* con vari metodi per TSP). Pertanto, questa sezione ha puro scopo esemplificativo e si riferisce ad alcuni semplici metodi adatti alla ricerca *rapida* di *buone* soluzioni per problemi di *grandi dimensioni*.

Nel seguito faremo riferimento al TSP simmetrico (in pratica, $c_{ij} = c_{ji}$), anche se i metodi presentati possono essere facilmente adattati al caso asimmetrico. Per convenienza espositiva, indicheremo con c_{ij} il costo c_e di uno spigolo avente come estremi i e j.

3.6.1 Determinazione della soluzione iniziale tramite euristiche

Oltre che partire da una soluzione casuale, ottenuta considerando una sequenza casuale di visita dei nodi del grafo, esistono diverse euristiche costruttive per TSP, tra le quali citiamo le seguenti.

Visita del più vicino (Nearest Neighbor): si parte da un nodo qualsiasi $i_0 \in N$ e si aggiunge un nodo alla volta, con criterio di espansione che seleziona il nodo più vicino all'ultimo nodo inserito. Lo schema è il seguente:

1. Seleziona un nodo $i_0 \in V$; poni $costo = 0$, $Ciclo = \{i_0\}$ e $i = i_0$.
2. seleziona $j = \arg \min_{j \in V \setminus Ciclo} \{c_{ij}\}$.

3. poni $Ciclo = Ciclo \cup \{j\}$; $costo = costo + c_{ij}$
4. poni $i = j$
5. se sono rimasti nodi da visitare, vai a 2
6. $Ciclo = Ciclo \cup \{i_0\}$; $costo = costo + c_{ii_0}$ (chiude il ciclo)

L'algoritmo è semplice e di complessità ridotta ($O(n^2)$, migliorabile con particolari strutture dati) ma ha prestazioni scadenti. In particolare, il ciclo degrada via via che viene costruito, visto che le scelte iniziali tendono a lasciare fuori i nodi più sfavoriti, non tenendo conto del fatto che si deve tornare al nodo di partenza. Per cercare di migliorare le performance, con effetti comunque scadenti, o per ottenere diverse soluzioni di partenza per la ricerca locale, si potrebbe eseguire n volte l'algoritmo, a partire dagli n diversi nodi del grafo (scelta di i_0 al passo 1); oppure si potrebbe randomizzare la scelta del nodo più vicino ad ogni passo (scegliendo, al passo 2 casualmente tra i k nodi più vicini a i).

Inserimento del più vicino/lontano (Nearest/Farthest Insertion): per considerare che il cammino si deve chiudere, questa euristica parte da un ciclo di lunghezza due e inserisce, ad ogni passo, un nodo nel ciclo, con criterio di espansione che seleziona il nodo più vicino/lontano dal ciclo. Il ciclo iniziale è ottenuto selezionando la i nodi i e j tali che c_{ij} è il minimo/massimo: il costo iniziale è quindi $c_{ij} + c_{ji}$. A ogni iterazione, se C è l'insieme dei nodi nel ciclo corrente, si seleziona il nodo

$$r = \arg \min_{i \in V \setminus C} / \max_{i \in V \setminus C} \{c_{ij} : j \in C\}.$$

Il nodo r viene inserito tra i nodi i e j consecutivi nel ciclo per cui è minimo il costo di inserimento

$$c_{ir} + c_{rj} - c_{ij}.$$

La complessità dell'euristica e circa $O(n^3)$, e produce risultati nettamente migliori della nearest neighbor, soprattutto nella versione *farthest insertion* (tende a mantenere un ciclo maggiormente bilanciato e a non degradare negli ultimi inserimenti, rispetto alla nearest insertion). In ogni caso, le due versioni, come la randomizzazione della scelta del nodo da inserire o del punto di inserimento, possono essere usate per ottenere punti di partenza alternativi per la ricerca locale.

Inserimento del meno costoso (Best Insertion): a partire da un ciclo di due nodi, determinato come in precedenza, a ciascuna iterazione si inserisce nel ciclo corrente C il nodo r che minimizza il costo per l'inserimento in un punto qualsiasi del ciclo, cioè

$$r = \arg \min_{i \in V \setminus C} \{c_{ir} + c_{rj} - c_{ij} : i, j \text{ consecutivi in } C\}.$$

La complessità è di nuovo $O(n^3)$, e anche per questa euristica è possibile randomizzare la scelta del nodo da inserire e ripetere la procedura, scegliendo il miglior ciclo hamiltoniano generato o utilizzando le soluzioni ottenute per diverse partenze della ricerca locale.

3.6.2 Rappresentazione della soluzione

Esistono diverse possibili rappresentazioni della soluzione per TSP, tra cui citiamo le seguenti:

- rappresentazione per archi: la soluzione è intesa come insieme di archi o spigoli (che devono rappresentare un ciclo, per l'ammissibilità) ed è rappresentata da una matrice binaria $M \in \{0,1\}^{n \times n}$ in cui righe e colonne fanno riferimento ai nodi: $M(i,j) = 1$ se e solo se l'arco (i,j) fa parte della soluzione;
- *adjacency representation*: si codifica ogni nodo come un numero intero tra 1 e n e si considera che una soluzione è definita se si indicano, per ogni nodo, il successivo nel ciclo. Si utilizza quindi un vettore di n nodi: il nodo in posizione i è quello da visitare nel ciclo dopo il nodo codificato con i;
- *path representation*: considerando che una soluzione del TSP corrisponde a una permutazione delle n città, la soluzione è intesa come sequenza ordinata di nodi, ed è rappresentata da un vettore ordinato di n nodi.

Nel proseguio si farà riferimento alla path representation, che corrisponde al modo più naturale di rappresentare la soluzione del TSP e, insieme alla adjacency representation, gode della proprietà che un qualsiasi vettore di nodi (senza ripetizioni) rappresenta una soluzione ammissibile (mentre, nel primo caso, non tutte le matrici rappresentano un tour!). La decodifica della path representation è immediata: basta costruire il ciclo seguendo l'ordine indicato nel vettore.

3.6.3 Vicinato

Classicamente, il vicinato per il TSP è ottenuto tramite scambio di k archi nella soluzione con k archi non appartententi alla soluzione. Per ottenere delle soluzioni ammissibili, è necessario che gli archi della soluzione non siano consecutivi nel ciclo di partenza; inoltre, una volta definiti i k archi da eliminare, si possono direttamente definire i k archi da introdurre nella soluzione per formare un nuovo ciclo. Questo tipo di vicinato è chiamato $k-opt$, ed è stato introdotto da Lin e Kernighan nel 1973. Degli esempi con $k = 2$ e $k = 3$ sono dati in Figura 2. Le mosse di k-scambio possono essere definite direttamente come operazioni sul vettore della *path representation*. Ad esempio, il vicinato $2-opt$ è ottenuto definendo una qualsiasi coppia i, j di nodi e invertendo la sottosequenza dei nodi tra i e j (*substring reversal*): nell'esempio, $i = 3$, $j = 6$ e la sequenza $3, 4, 5, 6$ viene sostituita da $6, 5, 4, 3$; un altro possibile vicino si ottiene per $i = 3$ e $j = 7$, che porta alla sequenza $< 1, 2, 7, 6, 5, 4, 3, 8, 1 >$ e così via.

Per quanto riguarda la complessità dei vicinati $k - opt$, dobbiamo calcolare il numero di vicini e la complessità della valutazione di un singolo vicino. Per $k = 2$, si ha un vicino per ogni coppia di archi non consecutivi o, sulla path representation, per ogni coppia di nodi i, j tali che i precede j nella sequenza centro del vicinato. Tenendo conto che possiamo fissare il nodi di partenza-arrivo (ad esempio il nodo 1) senza perdere soluzioni (è indifferente da

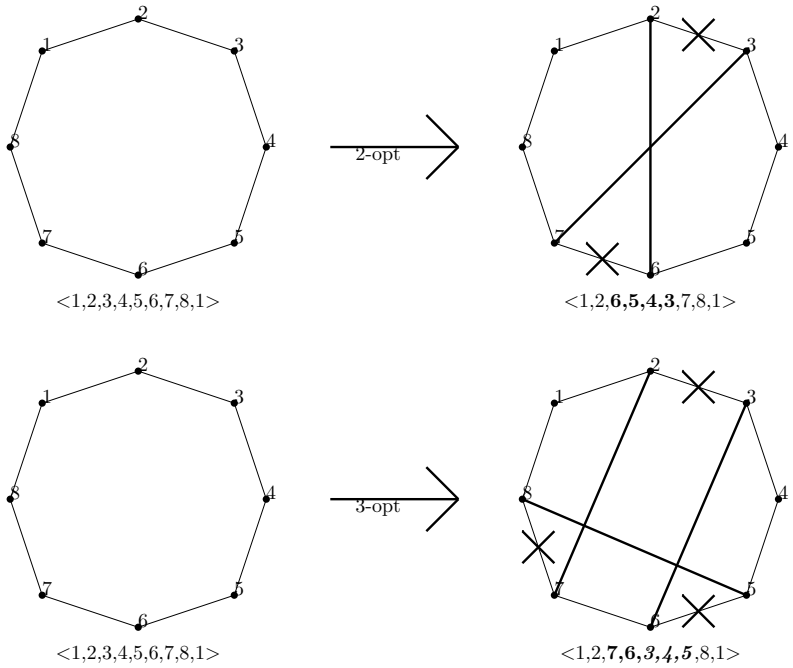


Figure 2: Esempi di vicini $2-opt$ e $3-opt$.

dove cominciamo a percorrere il ciclo), i e j sono da scegliere tra i restanti nodi e, quindi, si hanno $\frac{(n-1)(n-2)}{2} = O(n^2)$ vicini. Analogamente, per il vicinato $3-opt$ si ha un vicino per ogni tripletta di archi non consecutivi, per un totale di $O(n^3)$ vicini. In generale, è facile dimostrare che il numero di vicini $k-opt$ è $O(n^k)$. Circa la complessità della valutazione di ciascun vicino, si tratta di sottrarre dal valore della soluzione centrale il costo dei k archi eliminati e di aggiungere il costo dei k archi aggiunti, che si può fare in tempo costante $O(1)$: si ha quindi a disposizione una valutazione incrementale estremamente efficiente, al solo "costo" aggiuntivo di memorizzare, per il centro del vicinato, il valore della soluzione. Ad esempio, nel vicinato $2-opt$ la formula di valutazione del vicino ottenuto scambiando la sottosequenza tra i nodi i e j è:

$$C_{new} = C_{old} - c_{hi} - c_{jl} + c_{hj} + c_{il}$$

dove C_{new} è il costo del vicino, C_{old} è il costo della soluzione centro del vicinato, h è il nodo che precede i nella soluzione centro, l è il nodo successivo a j nella soluzione centro: si tratta di eliminare gli archi (h,i) e (j,l) e di sostituirli con gli archi (h,j) e (i,l).

In generale, la complessità del vicinato $k-opt$ è $O(n^k)$.

Bisogna quindi valutare il miglior compromesso tra la complessità del vicinato e le prestazioni della ricerca locale, il ché è ottenibile per via sperimentale, attraverso il confronto dei tempi di calcolo e delle soluzioni ottenibili utilizzando i diversi vicinati. A questo proposito, le prove effettuate da Lin mostrano che passando da $k = 2$ a $k = 3$ si ha un miglioramento

sensibile dei risultati, tale da giustificare l'incremento di complessità. Miglioramenti praticamente trascurabili si hanno invece per $k = 4$. Inoltre, il vicinato proposto risulta molto forte: sempre grazie alle prove di Lin su grafi con 48 nodi, si verifica che, utilizzando permutazioni casuali dei nodi come soluzione di partenza, si ottiene l'ottimo del problema nel 5% dei casi. Questo vuol dire che, con r ripetizioni della ricerca locale, si ha una probabilità di trovare la soluzione ottima pari a $p(r) = 1 - 0.95^r$: notare che $p(100) = 0.99$. Purtroppo, Lin ha anche verificato che, per n crescente, la probabilità di ottenere l'ottimo diminiuisce molto velocemente.

3.6.4 Funzione di valutazione delle soluzioni

Come implicito nelle argomentazioni precedenti, le soluzioni sono valutate con la mera funzione obiettivo: in effetti, ogni permutazione di nodi è una soluzione ammissibile sufficientemente differenziata, in termini di costo, dagli altri vicini, e non si ritiene utile introdurre penalità o altre componenti.

3.6.5 Strategia di esplorazione

Possono essere utilizzate sia la steepest descent che la first improvement.

4 Oltre la ricerca locale: la ricerca di vicinato (neighborhood search – trajectory methods)

Come osservato, il grosso pregio della ricerca locale sta nel rappresentare un ottimo compromesso tra semplicità realizzativa e risultati ottenuti. Tuttavia, il criterio di arresto utilizzato "intrappola" il metodo in minimi locali. In alcuni casi eccezionali, le caratteristiche dei vicinati e del relativo spazio di ricerca (connessione, convessità) garantiscono che un minimo locale sia anche globale, ma questa rimane una cirscostanza eccezionale[1]. Abbiamo già accennato a degli accorgimenti implementativi che cercano di sfuggire dagli ottimi locali, tra i quali ricordiamo, per riepilogo:

- *Random Multistart*: si tratta della tecnica più semplice, che consiste nel reiterare la ricerca locale con soluzioni iniziali diverse, generate casualmente o con euristiche randomizzate;

[1] Tra queste eccezioni, citiamo il caso dei problemi di programmazione lineare. Infatti, il metodo del simplesso può essere visto come un metodo di ricerca locale dove: la soluzione di partenza si ottiene con la fase I, se non già disponibile; la rappresentazione delle soluzioni è un insieme di colonne che definiscono una matrice di base; il vicinato risulta da mosse di scambio di una colonna in base con una fuori base; la valutazione del vicino si ottiene in modo incrementale, considerando i costi ridotti e le altre informazioni nel tableau via via aggiornato; la strategia di esplorazione è assimilabile a una first improvement particolarmente efficiente, visto che le regole per la scelta della colonna entrante e uscente permettono di trovare una soluzione migliorante senza esplorare esplicitamente i vicini. Inoltre, le proprietà dei costi ridotti e i teoremi della programmazione lineare, derivanti anche dalla convessità della funzione obiettivo, garantiscono che il minimo locale trovato (soluzione di base senza soluzioni miglioranti attorno, perché tutti i costi ridotti sono non negativi) è anche un ottimo globale.

- la modifica dinamica del vicinato (visto che la definizione di minimo locale dipende anche dal vicinato): ad esempio, nel TSP, si parte con un vicinato $2-opt$ e, se non si trovano vicini $2-opt$ miglioranti, si passa ad un vicinato $3-opt$. Su questa osservazione si basano delle tecniche avanzate tipo *Variable Neighborhood Descent* o *Variable Neighborhood Search* (che in realtà sono molto più complesse e sono al di fuori degli scopi del corso);

- *randomizzare* la strategia di esplorazione, scegliendo in modo random tra k vicini miglioranti;

- introdurre dei meccanismi di backtrack, basati sulla *memoria* di alcune possibili scelte alternative di vicini che possono essere considerate in un secondo momento, una volta raggiunto un minimo locale per altra via.

Quelli citati sono solo alcuni dei meccanismi utilizzati per sfuggire ai minimi locali. Negli ultimi decenni, la ricerca di metodi sistematici per generalizzare la ricerca locale e sfuggire ai minimi locali ha prodotto numerosi risultati, che possono essere raccolti sotto l'etichetta di *Neighborhood Search* o *Trajectory Methods*. L'ultimo nome deriva dal fatto che tali metodi cercano di costruire una traiettoria nello spazio di ricerca, come definito dal o dai vicinati, registrando la migliore soluzione che si incontra durante questa "passeggiata". Il meccanismo di base per muoversi nello spazio di ricerca è quello visto per la ricerca locale, cioè l'esplorazione di vicinati (da cui neighborhood search), nel quale però la regola di scegliere soluzioni miglioranti e fermarsi quando non ce ne sono viene modificata. Il fatto di accettare mosse peggioranti apre la possibilità di poter tornare, ad un certo punto, a una soluzione già esplorata, creando così un potenziale loop infinito: le varie tecniche sviluppate cercano proprio di definire dei meccanismi per sfuggire dai minimi locali e evitare di ciclare. Tali meccanismi si basano, come negli accorgimenti visti sopra, su due principi fondamentali: la randomizzazione dell'esplorazione (che permette di scegliere strade alternative se si dovesse tornare su una soluzione già visitata) e la memoria (che permette di ricordare le soluzioni già eplorate, per evitare di tornare sui propri passi). Nel primo caso, di fatto, si assume che non ci sia struttura nello spazio di ricerca, mentre nel secondo caso, si tende a considerare questa struttura, ottenendo, spesso, risultati migliori. Esistono diverse tecniche classificabili come Neighborhood Search e tra queste descriveremo brevemente (rimandando alla già citata ricognizione o a testi specifici per approfondimenti) la *Simulated Annealing*, per motivi eminentemente storici e per alcune proprietà di interesse teorico, e la *Tabu Search*, che si è mostrata, a oggi, il compromesso ingegneristico migliore tra qualità delle prestazioni, sforzo computazionale richiesto e semplicità (che vuol dire anche tempi e costi) di sviluppo.

4.1 Simulated Annealing*

L'idea di base di questi algoritmi, proposti da Kirkpatrick nel 1983, è di simulare il comportamento di un processo termodinamico di ricottura (*annealing*) di materiali solidi (vetro, metallo etc.). Se un materiale solido viene riscaldato oltre il proprio punto di fusione e poi viene raffreddato (*cooling*) in modo da riportarlo allo stato solido, le sue proprietà strutturali dipendono fortemente dal processo di raffreddamento (*cooling schedule*). Essenzialmente, un algoritmo di simulated annealing simula il cambiamento di energia di un sistema (considerato come insieme di particelle) sottoposto a raffreddamento, fino a che non converge ad uno stato solido; questo permette di cercare soluzioni ammissibili di problemi di ottimizzazione, cercando di convergere verso soluzioni ottime.

In un processo termodinamico, il sistema tende a organizzarsi in configurazioni a minima energia (configurazioni stabili) e la probabilità che un sistema operi un cambio di stato cui corrisponde un incremento δE di energia è pari a

$$p(\delta E) = \exp(-\delta E/kt)$$

dove k è la costante di Boltzmann e t è la temperatura del sistema.

Gli algoritmi di simulated annealing utilizzano tecniche di ricerca locale per definire ed esplorare l'intorno di una soluzione corrente e valgono quindi gli accorgimenti progettuali descritti in precedenza (soluzione iniziale, rappresentazione della soluzione, definizione del vicinato, valutazione dei vicini). Inoltre, si assimila l'energia del sistema fisico (da minimizzare) alla funzione obiettivo del problema in esame cosicché l'accettazione di mosse peggioranti viene subordinato a una "legge" probabilistica (randomizzazione dell'esplorazione del vicinato). In particolare, se l'intorno contiene una soluzione migliorante, allora questa diviene la nuova soluzione corrente ed il procedimento viene iterato. Altrimenti si valuta il peggioramento del valore della soluzione che si avrebbe spostandosi dalla soluzione corrente verso la miglior soluzione dell'intorno e si effettua questo spostamento in base ad una certa probabilità. L'algoritmo termina quando viene raggiunto un numero prefissato di iterazioni o un prefissato tempo di calcolo, oppure se viene dimostrata l'ottimalità della soluzione corrente, o dopo un certo numero di iterazioni senza miglioramento della soluzione.

Come accennato, il peggioramento del valore della soluzione viene visto come un aumento di energia del sistema termodinamico associato, per cui la probabilità di accettare la mossa peggiorante è data dalla formula di Metropolis

$$p = \exp(-\delta/t)$$

dove δ è l'entità del peggioramento (differenza tra il valore della nuova soluzione ed il valore della soluzione corrente), e t è la *temperatura di processo*. Se la mossa viene accettata, allora l'algoritmo continua partendo dalla nuova soluzione corrente; altrimenti,

*In parte, adattamento dalle dispense di Michele Monaci, Dipartimento di Ingegneria dell'Informazione, Università di Padova (monaci@dei.unipd.it)

viene considerata la probabilità di accettare la seconda miglior soluzione dell'intorno, e così via fino a che non viene trovata una qualche soluzione che viene accettata. Se non ci sono soluzioni vicine miglioranti e nessuna soluzione peggiorante viene accetta, il metodo si ferma e restituisce la migliore soluzione esplorata. Si noti come, anche se si tornasse su uno stesso minimo locale, l'esplorazione dello spazio di ricerca proseguirebbe ogni volta su una traiettoria probabilisticamente diversa, evitando cicli.

La formula di Metropolis stabilisce che la probabilità di accettare la mossa peggiorante diminuisce al crescere del peggioramento indotto dalla mossa stessa e cresce al crescere della temperatura di processo; questo è un parametro di controllo dell'algoritmo, che generalmente viene inizializzato ad un valore $t_0 > 0$ relativamente elevato e aggiornato (in linea di principio ridotto), con una certa frequenza, secondo una predeterminata *funzione di riduzione* (o *profilo di raffredamento* o *cooling schedule*), oppure, in modo dinamico, quando il numero di mosse accettate è stato superiore a una certa soglia per un certo numero di iterazioni. Questo è dovuto al fatto che la probabilità di accettare mosse peggioranti deve essere relativamente alta all'inizio dell'algoritmo (quando la sequenza di soluzioni considerate dipende fortemente dalla soluzione iniziale), per divenire poi più bassa man mano che la soluzione migliora (quando si stanno esplorando zone promettenti della regione ammissibile).

L'importanza dell'approccio deriva da motivi storici (può essere considerato il primo metodo di ricerca locale generalizzata applicato a problemi di ottimizzazione combinatoria, introdotto da Kirkpatrick nel 1983 e ispirato dai lavori in campo fisico di Metropolis nel 1953) e da alcune proprietà teoriche di convergenza. Infatti, il comportamento di un algoritmo di Simulated Annealing può essere modellato utilizzando le catene di Markov. Fissata la temperatura, la probabilità di muoversi dalla soluzione i alla soluzione j è una quantità $p_{ij} > 0$ che dipende solo da i e j. La corrispondente matrice di transizione corrisponde a quel che è noto come catena di Markov omogenea (almeno se il numero di iterazioni eseguite a temperatura costante è sufficientemente grande): se è possibile trovare una sequenza di scambi che trasformano una soluzione in un'altra con probabilità non nulla, allora il processo corrisponde ad una distribuzione stazionaria, indipendentemente dalla soluzione di partenza. Poichè la temperatura tende a zero, questa distribuzione stazionaria tende ad essere una distribuzione uniforme nell'insieme delle soluzioni ottime, per cui la probabilità che l'algoritmo converga ad una soluzione ottima tende a 1. In ogni caso, dato che lo spazio delle soluzioni ha generalmente dimensione esponenziale, il numero di iterazioni (e quindi anche il tempo di calcolo) necessario per assicurare la convergenza dell'algoritmo ad una soluzione ottima è anch'esso esponenziale (tanto vale applicare un approccio per enumerazione...). Dal punto di vista dei risultati computazionali, alcune nuove metodologie, come la tabu search descritta di seguito, si sono dimostrate migliori.

4.2 Tabu Search

La Tabu Search, introdotta da Fred Glover nel 1989, è un metodo basato su ricerca locale in grado di superare il limite dei minimi locali sfruttando sistematicamente meccanismi di *memoria*. In effetti, la ricerca locale o la simulated annealing non mantengono traccia della *storia* dell'esplorazione dello spazio delle soluzioni, se non ricordando la migliore soluzione esplorata fino a quel momento (che coincide con la soluzione corrente nella ricerca locale) e il corrispondente valore della soluzione, o il numero di iterazioni effettuate e/o non miglioranti. In questo modo, se l'accettazione di mosse non miglioranti porta, come visto, a rivisitare una certa soluzione, non si hanno elementi "strutturali" per evitare di tornare su soluzioni già visitate e l'unico modo per evitare di ciclare è affidarsi al caso, come nella simulated annealing, dove ci si basa sulla randomizzazione dell'esplorazione di vicinati non miglioranti. Nella tabu search, invece, si cerca di sfruttare la struttura dello spazio di ricerca, memorizzando alcune informazioni sulle soluzioni già visitate in modo da orientare la ricerca e sfuggire da minimi locali evitando di ciclare. Di base, questo si ottiene modificando il vicinato delle soluzioni in funzione della storia dell'esplorazione stessa, rendendo "tabu", cioè non esplorabili in quel momento, alcuni vicini.

Un semplice esempio può chiarire l'idea di base della tabu search: supponiamo di trovarci in un minimo locale x e sia $y \in N(x)$ il vicino migliore (anche se peggiore di x) secondo il vicinato N usato. Se si accetta di spostarsi su y, all'iterazione successiva, una soluzione migliorante nel vicinato $N(y)$ sarà sicuramente x ed è abbastanza probabile che x sia scelta come prossima soluzione corrente. Si innesca quindi un ciclo tra x e y, dal quale si potrebbe sfuggire semplicemente *ricordando* che x è una soluzione già visitata e impedendo (rendendo "tabu") la sua selezione.

4.2.1 La tabu list

L'esempio chiarisce come, più in generale, permettendo di spostarsi su soluzioni non miglioranti, esiste il pericolo di tornare su soluzioni già visitate, se non subito, anche dopo un certo numero di mosse. Pertanto, l'idea della Tabu Search è di mantere una struttura (*Tabu List*) nella quale vengono memorizzate le ultime t soluzioni visitate, che sono proibite (non legali, *tabu*) per le successive iterazioni. Indicando con x^i la soluzione considerata alla i-esima iterazione, alla generica iterazione k la lista tabu è una struttura del tipo

$$T(k) := \{x^{k-1}, x^{k-2}, \dots, x^{k-t}\}$$

e impedisce il verificarsi di cicli di lunghezza $\leq t$.

Si noti come, di norma, non si mantenga memoria di tutte le soluzioni visitate, per diversi motivi. In primo luogo, la memoria fisica necessaria potrebbe esplodere, rendendo, tra le altre cose, molto poco efficiente la verifica dell'appartenenza di un vicino alla tabu list. Per questo motivo, solitamente non si memorizzano le soluzioni visitate ma le *mosse* che sono state selezionate nelle ultime t iterazioni, o delle caratteristiche salienti delle ultime t soluzioni, in modo da rendere tabu le mosse (inverse) o le soluzioni che posseggono certe caratteristiche tabu. In effetti, tra gli elementi che definiscono l'efficienza del metodo,

bisogna considerare l'efficienza della verifica dell'appartenenza di un vicino alla lista tabu, il ché dipende dalla lunghezza della lista e dalla complessità del confronto da effettuare (ad esempio, tutta la soluzione o solo alcune caratteristiche).

Chiariamo questo punto considerando il caso del TSP con vicinato $2-opt$: anziché memorizzare gli ultimi t cicli hamiltoniani visitati, si possono memorizzare t coppie di archi oggetto di eliminazione nelle ultime mosse selezionate: se abbiamo scelto di cancellare gli archi (i,j) e (h,l) e, di conseguenza, aggiunto gli archi (i,h) e (j,l), questi archi non saranno scambiati per le prossime t mosse. Oppure si potrebbe decidere di rendere tabu tutte le mosse che coinvolgano i nodi i, j, h e l.

In effetti, in questo contesto, ciclare non vuol dire semplicemente tornare su una certa soluzione, ma ripercorrere ciclicamente una certa traiettoria nello spazio di ricerca. Quindi, anche se si dovesse tornare su una soluzione già visitata, l'importante è proseguire su una strada diversa, cosa possibile se si inibiscono le mosse (o caratteristiche salienti) recentemente considerate e pertanto contenute nella tabu list.

Un altro motivo per limitare la memoria delle soluzioni visitate è che, se si rendono tabu molti vicini (si pensi in particolare al caso di divieto su caratteristiche delle soluzioni), dopo un certo numero di passi si rischia di impoverire molto il vicinato "legale", impedendo un'adeguata esplorazione delle soluzioni. Per questo motivo, la lunghezza t della lista tabu (detta *tabu tenure*) è un parametro critico che deve essere dimensionato opportunamente: un t troppo basso rende la tabu list troppo corta e rimane il rischio di ciclare; per contro, se t è troppo elevato, la tabu list è troppo lunga e, come visto, si rischia di vincolare troppo la ricerca (perdendo dei vicini potenzialmente buoni) anche se ormai ci si è allontanati da una certa soluzione o da un certo minimo locale in modo sufficiente a rendere improbabile ciclare.

4.2.2 Criteri di aspirazione

Proprio per il fatto di non memorizzare delle "soluzioni tabu" ma delle "mosse/caratteristiche tabu", potrebbe succedere che alcune soluzioni nel vicinato siano dichiarate tabu, nonostante non siano state mai visitate. Potrebbe anche succedere che la soluzione generata da una mossa tabu abbia delle caratteristiche che la rendano comunque interessante: ad esempio, la soluzione potrebbe essere migliore di tutte le soluzioni visitate fino a quel momento (e, quindi, sicuramente, si tratta di una soluzione mai visitata). Sarebbe quindi opportuno ignorare il tabu e considerare tali soluzioni. Si definiscono pertanto dei *criteri di aspirazione* (*aspiration criteria*): se una mossa tabu genera una soluzione che soddisfa un criterio di aspirazione, la mossa viene comunque effettuata, nonostante possa essere tabu (*overruling*).

4.2.3 Criteri di arresto

Un ulteriore ingrediente è la definizione di opportuni criteri di arresto, visto che quello utilizzato dalla ricerca locale (non esistono vicini miglioranti, cioè siamo in un minimo

locale) non è più applicabile. In pratica, viene utilizzata una combinazione dei seguenti criteri:

- viene fissato a priori un numero massimo di iterazioni o un tempo di calcolo massimo;
- si trova una soluzione che è possibile certificare come ottima (ad esempio, per problemi di minimo, il suo valore è pari a un dato lower bound);
- viene fissato un numero massimo di iterazioni senza miglioramento (il numero di iterazioni dall'ultima volta che è stata aggiornata la migliore soluzione trovata supera una certa soglia);
- il vicinato della soluzione corrente, una volta escluse le soluzioni tabu, è vuoto, e non ci sono soluzioni vicine che soddisfano un eventuale criterio di aspirazione.

L'ultimo criterio è proprio della tabu search e potrebbe verificarsi per problemi fortemente vincolati, dove il numero di vicini ammissibili è molto ridotto. La presenza dell'ulteriore restrizione della tabu list rende ancora più critiche le caratteristiche di connessione dei vicinati, e pertanto si rivelano essenziali le tecniche che permettono di procedere nell'esplorazione di soluzioni non ammissibili, per poi tornare su soluzioni ammissibili (si parla in questi casi di *granular tabu search*). Questo, come abbiamo visto per la ricerca locale, si può ottenere penalizzando la valutazione di soluzioni non ammissibili. Un altro metodo per gestire il passaggio da soluzioni non ammissibili è quello di alternare fasi di ricerca primale e duale: finché è disponibile una soluzione vicina ammissibile, la funzione di valutazione coincide con la funzione obiettivo del problema (fase primale); quando il vicinato non contiene soluzioni ammissibili, si cambia funzione di valutazione e si utilizza una misura dell'inammissibilità cercando, nelle iterazioni successive (fase duale), di portare a 0 questa valutazione, ottenendo quindi una soluzione ammissibile dalla quale far ripartire una nuova fase primale.

4.2.4 Schema base della Tabu Search

In sintesi, nella tabu search, il vicinato associato alla soluzione corrente x non dipende solo da x ma anche dalla storia pregressa (insieme di soluzioni visitate finora), visto che alcune soluzioni che appartengono a $N(x)$ potrebbero essere escluse, a seconda dell'itinerario seguito per arrivare a x. Si parla, in casi come questo, di reattività della tabu search, in quanto l'esplorazione successiva dipende dalle soluzioni (buone o meno buone) visitate in precedenza. Se sintetizziamo la storia dell'esplorazione nel parametro k che corrisponde al numero dell'iterazione corrente (la storia comprende, ad esempio, lo stato della tabu list $T(k)$), potremmo definire il vicinato in funzione della soluzione corrente x e di k, e indicarlo con $N(x, k) \subseteq N(x)$.

Definiti gli ingredienti essenziali, schematizziamo la tabu search di base come segue:

1. genera una soluzione iniziale x e poni $k := 0$, $T(k) = \emptyset$, $x^* = x$;
2. genera il vicinato $N(x)$;

3. scegli la soluzione y che ottimizza il criterio di valutazione $\tilde{f}(y)$ tra tutte le soluzioni in $N(x,k)$ o tra le soluzioni in $N(x) \setminus N(x,k)$ che soddisfano qualche criterio di aspirazione;

4. ottieni $T(k+1)$ da $T(k)$, inserendo y (o la mossa $x \mapsto y$ o qualche caratteristica di y) e, se $|T(k)| \geq t$, eliminando la soluzione (o la mossa o la caratteristica) "più vecchia";

5. se $f(y)$ è migliore di $f(x^*)$, poni $x^* := y$;

6. se non è soddisfatto un *criterio di arresto*, poni $k = k+1$, $x = y$ e torna al passo 2;

7. return (x^*).

Lo schema è molto semplice e ricalca quello della ricerca locale, modificato con gli ulteriori ingredienti quali tabu list, criteri di aspirazione, criteri di arresto. Pertanto, oltre a quanto detto sopra per le componenti specifiche della tabu search, valgono tutti gli accorgimenti progettuali già discussi per la ricerca locale circa:

- la determinazione di una soluzione iniziale;
- la rappresentazione della soluzione;
- la definizione del vicinato, con relativa complessità e possibilità di valutazione incrementale;
- la funzione di valutazione delle soluzioni (che potrebbe essere diversa dalla funzione obiettivo f, come evidenziato dall'utilizzo di $\tilde{f}$ nello schema presentato).

Riguardo alle strategie di eplorazione, si noti come, di base, si utilizzi una strategia steepest descent, anche se nulla vieta, per velocizzare la ricerca, l'adozione di strategie first improvement. Inoltre, l'eventuale presenza di criteri di aspirazione rende necessaria la valutazione di tutti i vicini, anche quelli tabu, che potrebbe essere evitata (per aumentare l'efficienza) se tali criteri non fossero utilizzati.

4.2.5 Intensificazione e diversificazione

Lo schema di base sopra descritto permette di sviluppare algoritmi che, in genere, forniscono delle buone prestazioni. Queste possono essere ulteriormente migliorate, in modo decisivo per le applicazioni, estendendo sistematicamente l'utilizzo della memoria dell'esplorazione per alternare fasi di *intensificazione* e *diversificazione* della ricerca.

L'intensificazione consiste nell'esplorazione approfondita di alcune aree dello spazio di ricerca che sembrano promettenti: ad esempio, ci si concentra su soluzioni che possiedono determinate caratteristiche, o su soluzioni tra loro relativamente "simili". La diversificazione consiste invece nel cercare di individuare aree poco visitate dello spazio delle soluzioni, con lo scopo di individuare nuove aree promettenti su cui intensificare la ricerca:

ad esempio si privilegia la selezione di soluzioni con caratteristiche diverse dalla migliore soluzione corrente. L'alternarsi delle fasi di intensificazione e diversificazione intende orientare la ricerca in modo efficiente verso l'individuazione di diversi minimi locali e, quindi, di soluzioni globalmente migliori. L'intensificazione e la diversificazione possono essere applicate a tutte le metaeuristiche e la loro esposizione esaustiva esula dai nostri scopi: ci limitiamo qui a fornire alcune idee su come queste possano essere implementate nel particolare contesto della tabu search.

Per ottenere l'intensificazione della ricerca si può, ad esempio:

- enumerare (anche implicitamente) tutte le soluzioni con determinate caratteristiche;
- utilizzare dei vicinati alternativi, anche se di complessità più elevata;
- adottare criteri di aspirazione più permissivi;
- modificare la funzione di valutazione dei vicini, *penalizzando* le soluzioni che si discostano, in termini di caratteristiche, da quella corrente.

Possibili tecniche di diversificazione sono le seguenti:

- utilizzare, all'interno dello stesso algoritmo Tabu Search, intorni diversi. Ad esempio, per il TSP, qualora si verifichi un criterio di arresto con un vicinato $2-opt$, la ricerca può proseguire con un vicinato $3-opt$, fino a che non si trova una soluzione migliorante. In generale, si possono definire diversi vicinati che permettono di raggiungere soluzioni più o meno distanti (dissimili) dalla soluzione centro, e si stabiliscono dei criteri di priorità nell'esplorazione di tali vicinati; a ciascun intorno è associata una lista tabu, la cui gestione è del tutto indipendente da quella delle altre liste tabu.
- modificare la funzione di valutazione dei vicini, *premiando* le soluzioni che si discostano, in termini di caratteristiche, da quella corrente;
- alla fine di una fase di intensificazione, considerare la migliore soluzione ottenuta x' e costruire una nuova soluzione di partenza che sia il più possibile diversa da x' (complementare), in modo da ricercare, attraverso un'ulteriore intensificazione, una migliore soluzione partendo da un punto in un'area diversa dello spazio di ricerca;
- un modo più raffinato (e coerente con i principi della tabu search, che si basa sull'utilizzo sistematico della memoria) è l'introduzione di una *memoria di lungo termine*, che raccolga delle informazioni sulla storia dell'esplorazione. In effetti, la tabu list, nella sua definizione base, rappresenta una memoria di breve termine (recency-based memory, si memorizzano poche mosse recenti), utilizzata per dirigere la ricerca locale al fine di rendere trascurabile la probabilità di ciclare in seguito all'accettazione di mosse peggioranti. Attraverso una memoria di lungo termine, possono essere dati nuovi e diversi indirizzi alla ricerca locale. Ad esempio, possono

essere collezionate statistiche sulle caratteristiche più o meno esplorate (perché più o meno presenti nelle soluzioni via via selezionate nel costruire la traiettoria nello spazio di ricerca) e, in base a queste statistiche, incentivare (ad esempio premiando nella funzione di valutazione) la selezione di soluzioni portatrici di caratteristiche poco esplorate.

Un modo semplice di ottenere l'alternarsi di fasi di intensificazione e diversificazione in contesti di tabu search è la *gestione dinamica della lunghezza della tabu list* (parametro t, tabu tenure) che, quindi, non ha più la mera funzione di evitare i cicli potenzialmente innescati dall'accettazione di mosse peggioranti. In particolare, nelle fasi di intensificazione, t si tiene a valori bassi (il minimo valore t_0 che impedisce di ciclare). Qualora la migliore soluzione disponibile x^* non venga aggiornata per un determinato numero di iterazioni, il valore di t cresce, mentre torna a diminuire (fino al valore limite t_0) quando si aggiorna x^*. Si noti che, all'aumentare di t, aumentano il numero di soluzioni, mosse o caratteristiche tabu e, di conseguenza, si tende ad accettare soluzioni che siano sufficientemente diverse dalle ultime esplorate e, in definitiva, a muoversi velocemente verso aree dello spazio delle soluzioni diverse da quella corrente, ottenendo quindi una diversificazione.

4.2.6 Un esempio di applicazione della tabu search

Consideriamo il problema della colorazione di un grafo. Colorare un grafo non orientato significa assegnare a ogni vertice un colore in modo tale che vertici adiacenti (estremi dello stesso spigolo) abbiano colori diversi. Una $k-$colorazione di un grafo è una colorazione che utilizza k colori. Dato un grafo non orientato $G = (N, E)$, il problema della colorazione di un grafo consiste nel determinare il numero cromatico di G, indicato con $\gamma(G)$, ossia il numero minimo di colori necessari per colorare G e una relativa assegnazione di colori a ciascun vertice (applicazione tipica è la colorazione di cartine geografiche).

Innanzitutto, è necessario definire le componenti base della ricerca locale: ci concentriamo qui sulla rappresentazione della soluzione, sulla definizione del vicinato e della funzione di valutazione (una soluzione di partenza si potrebbe banalmente ottenere colorando tutti i nodi con colori diversi, anche se esistono diverse euristiche possibili). In effetti, come vedremo, il problema è piuttosto vincolato e presenta problematiche inerenti la connessione dei vicinati (già definita come la possibilità, data una soluzione di partenza, di raggiungere una qualsiasi soluzione attraverso una successione di mosse), il ché è particolarmente rilevante nel caso di applicazione di una tabu search (visto che la tabu list tende a impoverire ulteriormente i vicinati).

La **rappresentazione della soluzione** consiste in un vettore di lunghezza $n = |V|$, con un elemento per ciascun vertice riportante il colore assegnato al vertice stesso. Degli esempi sono riportati in Figura 3, con una 3-colorazione (funzione obiettivo $f = 3$) e una 2-colorazione ($f = 2$). Si noti come non tutte le assegnazioni di colori sono ammissibili, visto che potrebbero violare i vincoli sulla colorazione di vertici adiacenti.

Una prima **definizione di vicinato** potrebbe derivare dalle mosse che cambiano il colore di un nodo alla volta, cercando di non aumentare il numero di colori utilizzati nella

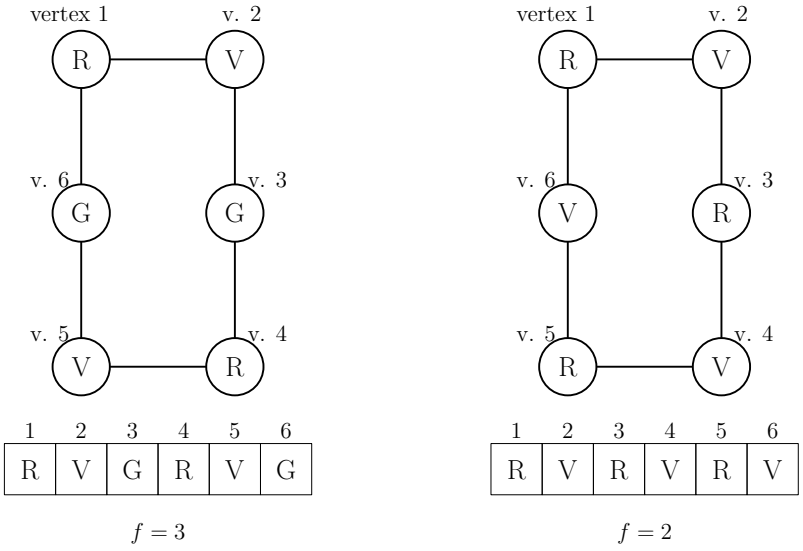


Figure 3: Esempi di soluzioni per colorazione di grafi e relative rappresentazioni: una 3-colorazione e una 2-colorazione.

soluzione centro del vicinato: si tratta quindi di generare un vicino per ciascun nodo e per ciascuno dei colori utilizzati da altri nodi nella colorazione centro. Nell'esempio, a partire dalla 3-colorazione in Figura 3, si otterrebbero 2 vicini per ogni nodo, per un totale di 12 vicini: $\underline{\text{V}}$VGRVG, $\underline{\text{G}}$VGRVG, R$\underline{\text{R}}$GRVG, R$\underline{\text{G}}$GRVG, RV$\underline{\text{R}}$RVG etc.

Già da questo piccolo esempio si vede che nessuno dei vicini è ammissibile, rendendo evidente le caratteristiche di scarsa connessione del vicinato definito. Abbiamo quindi due alternative: cambiare vicinato o ammettere il passaggio per soluzioni non ammissibili. Prima di scegliere tra le due alternative, consideriamo alcune osservazioni sulla funzione di valutazione delle soluzioni.

La scelta naturale per la **funzione di valutazione** delle soluzioni sarebbe la funzione obiettivo da minimizzare, ossia il numero di colori utilizzati dalla colorazione proposta. In effetti, il numero cromatico di un grafo tende a essere basso e, comunque, non molto inferiore al valore della soluzione di partenza che si può ottenere con un'euristica. Ne consegue che molte soluzioni ammissibili utilizzano lo stesso numero di colori e, quindi, lo spazio di ricerca risulta estremamente piatto, cioè molte soluzioni vicine, indipendentemente dal vicinato scelto, hanno lo stesso valore della funzione obiettivo, configurando dei *plateau*. Di conseguenza, la tabu search seguirà una traiettoria abbastanza casuale nello spazio di ricerca, visto che un numero elevatissimo di vicini rappresentano delle direzioni altrettanto appetibili, rischiando così di visitare molte soluzioni equivalenti prima di trovare, altrettanto casualmente, una soluzione con un numero di colori inferiore (se non interviene prima un criterio di arresto).

Risulta pertanto opportuno utilizzare una funzione $\tilde{f}$ di valutazione delle soluzioni diversa dalla funzione obiettivo f. Inoltre, una diversa definizione di $\tilde{f}$ potrebbe essere sfruttata anche per mantenere un vicinato simile a quello proposto e consentire il passaggio per

soluzioni non ammissibili, in modo di ovviare alla non connessione del vicinato stesso. Si tratta quindi di utilizzare una $\tilde{f}$ che, da un lato, penalizzi le soluzioni non ammissibili, dall'altro risulti sufficientemente variata tra i vicini, di modo da evidenziare meglio direzioni di ricerca promettenti.

Le osservazioni precedenti ci portano a ridefinire le diverse componenti della tabu search come segue. Innanzitutto, visto che la ricerca del numero cromatico riguarda un insieme ristretto di possibili valori, possiamo fissare un numero di colori desiderato: ad esempio, se disponiamo di una k'-colorazione (ottenuta all'inizio con un'euristica), fissiamo un numero di colori $k = k' - 1$.

Quindi trasformiamo la $k'-$colorazione disponibile in un assegnamento di k colori ai vertici (ad esempio, eliminando un colore tra quelli usati trasformandolo in un altro su base probabilistica o seguendo delle regole euristiche). A questo punto la soluzione x ottenuta (un semplice assegnamento di colori, non necessariamente una $k-$colorazione) potrebbe non essere ammissibile, cioè, potrebbero esserci degli spigoli con ambedue gli estremi dello stesso colore. Chiamiamo questi spigoli *monocromatici* e consideriamo come **valutazione della soluzione** $\tilde{f}(x)$ il numero di spigoli monocromatici, che è misura del grado di inammissibilità: chiaramente, se $\tilde{f}(x) = 0$ la soluzione x rappresenta una k-colorazione (che, nell'esempio, utilizza un colore in meno rispetto alla soluzione di partenza), altrimenti, la tabu search mira a minimizzare la funzione $\tilde{f}$ fino a ottenere il valore 0, nel qual caso si reitera il procedimento considerando un k inferiore. Si noti come diverse soluzioni hanno un diverso grado di inammissibilità e, pertanto, $\tilde{f}$ risulta sufficientemente variata.

Il **vicinato** può essere definito (in analogia con quanto detto sopra) attraverso mosse che cambiano il colore di un vertice per volta in tutti i $k - 1$ modi possibili. In particolare, per ridurre le dimensioni del vicinato, possiamo concentrarci sulle inammissibilità e considerare solo le mosse che cambiano il colore di un vertice che sia estremo di un arco monocromatico.

Per quanto riguarda la **tabu list**, consideriamo la memorizzazione delle t mosse più recenti: una mossa può essere memorizzata con la coppia (v, r), dove v è il vertice coinvolto e r è il colore assunto dal vertice stesso: in questo modo, per t mosse sarà proibito far assumere a v il colore r.

Le restanti componenti (in particolare i criteri di arresto o di aspirazione) possono essere definite in maniera standard.

L'utilizzo della funzione di valutazione $\tilde{f}$ e del vicinato consentono (empiricamente) di ottenere buoni risultati (migliorabili con l'utilizzo, ad esempio, di intensificazione e diversificazione) su un problema difficile di sicuro interesse teorico e pratico come la colorazione di grafi.

5 Metodi basati su popolazione

5.1 Principi

I metodi descritti nella sezione precedente hanno la caratteristica di costruire una traiettoria *puntuale* nello spazio delle soluzioni, considerando, ad ogni iterazione *una sola* soluzione. Esistono delle metaeuristiche che, invece, mantengono *una popolazione* di soluzioni, ossia un insieme di più soluzioni, e, ad ogni iterazione, combinano tra loro queste soluzioni per ottenere una nuova popolazione. L'idea è quella che, attraverso opportuni operatori di ricombinazione, si possano ottenere delle soluzioni migliori rispetto a quelle correnti. Questi metodi si dicono *population based* e, in molti casi, sono ispirati da meccanismi naturali, presupponendo una tendenza della natura a organizzarsi in strutture "ottimizzate". Negli ultimi anni ci sono stati molti studi in questo ambito, peraltro con forti connotazioni interdisciplinari (Ricerca Operativa, Intelligenza Artificiale, Soft Computing etc.) che hanno portato alla definizione di diversi paradigmi di ottimizzazione, ad esempio, *Evolutionary Computation*, *Scatter Search e path relinking*, *Ant Colony Optimization*, *Swarm Optimization* etc. Nel seguito descriveremo gli *Algoritmi Genetici* che, oltre a essere probabilmente la prima metaeuristica population based proposta, sono estremamente diffusi nelle applicazioni, principalmente per la loro semplicità implementativa e adattabilità a praticamente tutti i problemi di ottimizzazione nei più svariati ambiti (molti tool di "ottimizzazione" general purpose sono basati su algoritmi genetici). più che per le effettive prestazioni sul campo.

5.2 Gli algoritmi genetici

I principi di base degli algoritmi genetici sono stati stabiliti da Holland nel 1975, e si ispirano alle teorie evoluzionistiche di Darwin, pubblicate nel 1859. Parafrasando (con molta licenza) queste teorie, possiamo vedere gli individui, nelle loro diverse fasi evolutive, come "soluzioni" sempre più adattate all'ambiente in cui vivono, e assimilare a un qualche processo di "ottimizzazione" l'evoluzione di una popolazione di individui: gli individui (genitori/*parent*) si combinano tra loro (riproduzione) per generare nuovi e diversi individui (figli/*offspring*) che entrano a far parte delle popolazioni (generazioni) successive; la partecipazione ai processi riproduttivi è più probabile per gli individui maggiormente adattati all'ambiente, secondo i principi di selezione "naturale" (*natural selection*) e "sopravvivenza degli individui più idonei" (*survival of the fittest*).

Gli algoritmi genetici cercano di simulare il processo evolutivo, facendo corrispondere a ciascun individuo una *soluzione*, e al livello di adattamento all'ambiente una *fitness*, cioè una misura quantitativa della qualità della soluzione stessa, provando così a far sopravvivere soluzioni di qualità sempre più elevata.

5.2.1 Schema generale di un algoritmo genetico

Gli algoritmi genetici partono da una popolazione iniziale di soluzioni (gli individui dei sistemi biologici) e la fanno evolvere iterativamente. Ad ogni iterazione, le soluzioni sono valutate (fitness, livello di adattamento all'ambiente) e, sulla base di questa valutazione, vengono selezionate alcune di esse (principio di selezione), privilegiando le soluzioni (genitori) con fitness maggiore (survival of the fittest). Le soluzioni selezionate vengono tra loro ricombinate (riproduzione) per generare delle nuove soluzioni (offspring) che tendono a trasmettere le (buone) caratteristiche delle soluzioni genitori nelle successive generazioni. Lo schema generale di un algoritmo genetico è il seguente:

1. *Codifica* delle soluzioni dello specifico problema.
2. Creazione di un insieme iniziale di soluzioni (*popolazione iniziale*).
3. Ripeti, fino alla soddisfazione di un *criterio di arresto*
   (a) *Seleziona* coppie (o gruppi) di soluzioni (parent).
   (b) *Ricombina* i genitori generando nuove soluzioni (offspring).
   (c) Valuta la *fitness* delle nuove soluzioni.
   (d) *Rinnova* la popolazione, utilizzando le nuove soluzioni.
4. Restituisci la migliore soluzione generata.

Come per tutte le metaeuristiche, si tratta di uno schema molto generale, che deve essere specializzato per i diversi problemi. Il punto di partenza è la

- codifica delle soluzioni

in base alla quale devono essere definiti i diversi *operatori genetici*, principalmente:

- metodi per generare un opportuno insieme di soluzioni della popolazione iniziale;
- funzione che valuta la fitness di ciascuna soluzione;
- operatori di ricombinazione;
- operatori di passaggio generazionale.

5.2.2 Codifica degli individui

Gli operatori genetici si basano su una *rappresentazione genetica* della soluzione che codifica le caratteristiche di una soluzione. Tale rappresentazione corrisponde al *cromosoma* degli individui biologici, al punto che spesso si parla indifferentemente di soluzione, individuo o cromosoma. Sempre proseguendo con l'analogia, ciascun cromosoma è ottenuto come sequenza (stringa) di *geni*. Ciascun gene è solitamente associato a una variabile decisionale del problema e assume uno dei valori possibili per quella variabile: a seconda

dei diversi valori effettivamente assunti dai diversi geni, si ottiene un diverso cromosoma e, quindi, una diversa soluzione. Per passare dal cromosoma alla soluzione, è necessaria una *decodifica* (che potrebbe essere immediata).

Esempio*: problema KP/0-1.* Si può utilizzare una *codifica binaria*, associando a ciascun oggetto un numero d'ordine da 1 a n (numero di oggetti) e utilizzando un gene per ogni oggetto che può assumere i valori 0 o 1. La decodifica è immediata: il gene i vale 1 se e solo se l'oggetto i è nello zaino. Un esempio di cromosoma per $n = 10$ è il seguente:

1	0	0	1	1	0	0	0	1	0

una stringa binaria che rappresenta una soluzione con gli oggetti 1,4,5 e 9 nello zaino.

Esempio*: problema TSP.* Se n è il numero di città, si utilizzano n geni che possono assumere, ciascuno, un valore associato a una città. Il gene in posizione i indica la città da visitare in posizione i nel ciclo hamiltoniano. Il cromosoma è quindi una stringa (sequenza) di città la cui decodifica è immediata, indicando direttamente l'*ordine* di visita, la permutazione delle città (corrisponde alla *path representation*). Un esempio con 10 città codificate da 0 a 9 è il seguente:

3	2	6	1	8	0	4	7	1	5

e indica il ciclo hamiltoniano $3 \to 2 \to 6 \to 1 \to 8 \to 0 \to 4 \to 7 \to 1 \to 5$.

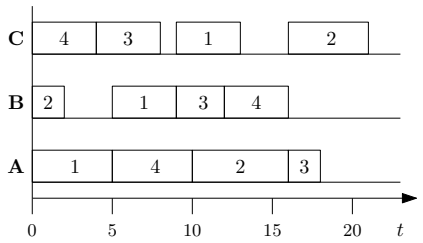
Esempio*: un problema di sheduling (*Job Shop Scheduling*).* Sono dati n job da eseguire su m macchine. Ogni job è composto da una sequenza ordinata di k task. Il task j del job i deve essere eseguito su una determinata macchina, impegnandola in modo continuativo ed esclusivo per un tempo t_{ij}. È fissato l'ordine di esecuzione dei task di uno stesso job e ogni task deve aspettare che il task precedente sia terminato. Si vuole determinare l'ordine di esecuzione dei diversi task sulle diverse macchine in modo da minimizzare il tempo di completamento dei job. Il problema è una generalizzazione del problema della lettura dei giornali presentato nella prima parte del corso: ogni ragazzo corrisponde a un job, ogni giornale a una macchina e la lettura di un giornale da parte di un ragazzo corrisponde a un task. Una possibile codifica utilizza una sequenza di $n \times k$ geni. Ciascun gene può assumere dei valori tra 1 e n. Ad esempio, sia dato un problema con 4 job, ciascuno con 3 task da eseguire sulle macchine A, B e C. La sequenza dei task e i tempi di completamento sono indicati nella seguente tabella:

Job	macchina , t_{ij}		
1	A , 5	B , 4	C , 4
2	B , 2	A , 6	C , 5
3	C , 4	B , 2	A , 2
4	C , 4	A , 5	B , 4

Un possibile cromosoma è il seguente:

4	2	1	1	3	4	2	3	1	2	3	4

Si noti come non serva indicare anche il numero di task nel cromosoma, essendo fissata la sequenza dei task dello stesso job. In questo caso, la decodifica non è immediata e richiede l'uso di una euristica, ad esempio la seguente, di complessità lineare: si scorre la sequenza dei geni e sia i il job indicato dal gene corrente; si considera il primo task j del job i non ancora considerato e lo si schedula sulla corrispondente macchina *al più presto* (la macchina deve essere libera e il task precedente terminato). In pratica, l'ordine dei geni indica la priorità di ciascun task sulle macchine (che, di fatto, è la variabile decisionale del problema). Il valore della funzione obiettivo si ottiene considerando il tempo in cui finisce l'ultimo task. In base a questa euristica, il cromosoma precedente corrisponde alla soluzione indicata nel diagramma di Gantt



che ha valore della funzione obiettivo pari a 21.

Nella formulazione originaria, gli algoritmi genetici suggerivano di utilizzare una codifica binaria per tutti i problemi (che è sempre possibile, basti pensare che, nelle implementazioni al calcolatore, un cromosoma corrisponderà a una struttura dati in una memoria binaria), mentre le ricerche e i risultati sperimentali successivi indicano che, per migliorare le performance, conviene utilizzare codifiche specifiche, così come abbiamo esemplificato. Pertanto, la codifica dipende fortemente del problema e influenza il progetto delle successive componenti dell'algoritmo genetico.

5.2.3 Popolazione iniziale

Il metodo base per ottenere la popolazione iniziale è la generazione *casuale* di un numero N di individui. In effetti, una proprietà importante è la *diversificazione* degli individui, che permette di avere un patrimonio genetico più ricco e, quindi, maggiore possibilità di ottenere dei buoni individui nelle generazioni successive. Tuttavia, per accelerare la convergenza generale del metodo e non lasciare semplicemente al caso il compito di scoprire alcune buone caratteristiche che vorremmo includere nella soluzione, si possono introdurre nella popolazione iniziale alcune *soluzioni generate con euristiche* (costruttive o una rapida ricerca locale) eventualmente randomizzate, per ottenere una varietà di buoni individui con diverse buone caratteristiche. È comunque importante che il numero di tali soluzioni

sia limitato, in modo da non condizionare troppo le caratteristiche delle soluzioni che verranno generate nelle successive iterazioni, facendole convergere sì velocemente, ma verso individui che somigliano agli individui di partenza (ottenendo, probabilmente, dei minimi locali) impedendo l'esplorazione di individui con caratteristiche diverse e, magari, migliori.

5.2.4 Funzione di fitness

La funzione di fitness serve a dare una misura quantitativa dell'*idoneità* di un individuo. La fitness, infatti, guida i processi di selezione degli individui, di modo che, di generazione in generazione, si tenderà a far "sopravvivere" gli individui con fitness maggiore, cioè a passare da una generazione all'altra il loro corredo genetico e quindi le caratteristiche. Siccome siamo interessati ad ottenere dei valori ottimali della funzione obiettivo, si solito si lega la funzione di fitness al valore della funzione obiettivo (o a una sua misura inversa per problemi di minimo). Tuttavia, come abbiamo già discusso per la ricerca locale, potrebbe essere utile considerare una funzione di fitness diversa, che includa, ad esempio, dei termini di penalità da utilizzare, tra l'altro, per:

- penalizzare soluzioni non ammissibili, se si decide di mantenerle nella popolazione corrente. Questo potrebbe avere dei vantaggi perché degli individui globalmente inammissibili potrebbero essere portatori di caratteristiche interessanti che conviene mantenere nel corredo genetico;
- penalizzare soluzioni simili all'ottimo corrente. Questo potrebbe servire per implementare delle fasi di diversificazione della ricerca, in modo da privilegiare la selezione di individui diversi da quelli finora generati;
- penalizzare soluzioni troppo dissimili dall'ottimo corrente, in fasi di intensificazione.

5.2.5 Operatori di selezione

Gli operatori di selezione scelgono, tra la popolazione corrente, gli individui che partecipano ai processi riproduttivi, cioè gli individui per i successivi operatori di ricombinazione. Come abbiamo visto, in accordo con i principi di selezione naturale, la selezione deve favorire gli individui più adatti, cioè quelli con fitness più elevata.

Se si selezionassero soltanto gli individui "migliori", l'algoritmo potrebbe *convergere prematuramente* verso ottimi locali, perché, dopo qualche iterazione, tutti gli individui tenderebbero a essere simili agli individui migliori della popolazione iniziale, impedendo la possibilità di scoprire individui con caratteristiche diverse e, magari, migliori. Per questo motivo la selezione opera, di nuovo, su base *probabilistica*: gli individui con fitness maggiore hanno una maggiore probabilità di essere selezionati per le successive ricombinazioni.

Fissato questo principio, esistono svariate modalità per realizzarlo, ad esempio:

- si seleziona una coppia di genitori (o più in generale un gruppo di uno o più individui) per volta;

- si seleziona in base alla fitness un sottoinsieme della popolazione corrente (*mating pool*) e gli individui in questo sottoinsieme saranno utilizzati (*pescati*) dagli operatori di ricombinazione;

Un primo metodo per ottenere una selezione guidata dalla fitness è il *metodo Montecarlo*[2], secondo cui la probabilità p_i di selezionare l'individuo i è semplicemente *proporzionale* al valore f_i della sua fitness:

$$p_i = \frac{f_i}{\sum_{k=1}^{N} f_k}.$$

In questo modo, soprattutto se combinato con il primo metodo di selezione citato, si potrebbero privilegiare eccessivamente gli individui migliori, soprattutto in presenza di uno o pochi *superindividui* con un valore della fitness molto superiore a quello degli altri: tali individui tenderebbero a essere selezionati molto frequentemente, generando così offspring simile a loro e, di nuovo, in poche iterazioni la popolazione potrebbe convergere verso individui non troppo dissimili dai superindividui (minimo locale). La letteratura propone vari metodi per ovviare a questo inconveniente, tra cui citiamo i seguenti:

- *Linear ranking*: gli individui si ordinano per fitness crescente e si selezionano proporzionalmente alla loro posizione (ranking) nell'ordinamento. In questo modo si annulla l'effetto dei valori della fitness, che potrebbero essere molto diversi tra loro, mentre viene considerata solo la posizione reciproca. Più precisamente, se σ_i è la posizione dell'individuo i nell'ordinamento, si ha

  $$p_i = \frac{2\sigma_i}{N(N+1)}.$$

  La definizione garantisce $\sum_i p_i = 1$ (le p_i sono una distribuzione di probabilità).

- *Torneo-n*: la selezione di ciascun individuo si ottiene scegliendo in modo uniforme n individui ($2 \leq n \ll N$) e quindi selezionando il migliore tra questi.

5.2.6 Operatori di ricombinazione (*crossover*)

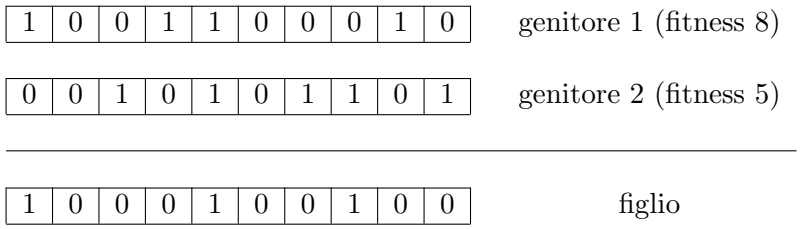
Gli operatori di ricombinazione agiscono su uno o più individui generando uno o più figli che "somigliano" ai genitori: si tratta quindi di individui diversi dai genitori ma che ne fondono le caratteristiche. Di solito, il numero di genitori è maggiore o uguale a due e, spesso, esattamente due, in analogia con la maggior parte dei processi riproduttivi naturali. Questo proprio perchè, utilizzando un solo individuo, si tenderebbe a fare una copia del genitore, non avendo i meccanismi base per ottenere soluzioni diverse. Sempre

[2]Per implementare il metodo Montecarlo, date le probabilità desiderate di selezionare ciascun individuo i (p_i), si ordinano gli N individui per fitness crescente e si associa a ogni individuo un segmento dell'intervallo $[0, 1]$ di ampiezza p_i a partire da 0 e in modo contiguo. Si genera quindi casualmente un numero k tra 0 e 1 e si sceglie l'individuo i se $\sum_{j=0}^{i-1} p_j < k \leq \sum_{j=i+1}^{n} p_j$.

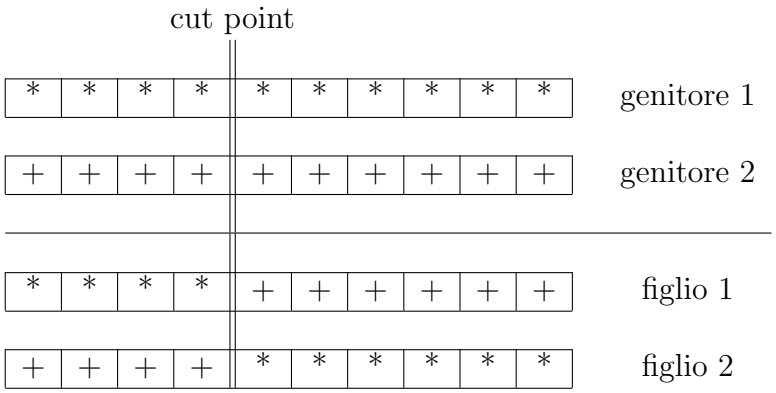
in analogia con la nomenclatura usata in biologia, gli operatori di ricombinazione che utilizzano due (o più) genitori si dicono *crossover*. Diamo di seguito alcuni degli operatori di crossover maggiormente utlizzati, nella loro versione base.

Crossover uniforme: a partire da due genitori si genera un figlio. I geni del figlio sono copiati dal primo genitore con probabilità p e dal secondo genitore con probabilità $(1-p)$. Si solito, si pone $p = 0.5$, oppure si calcola in modo proporzionale alla fitness (di modo da far maggiormente somigliare il figlio al genitore con fitness più elevata).

Esempio*: crossover uniforme su un cromosoma binario*:

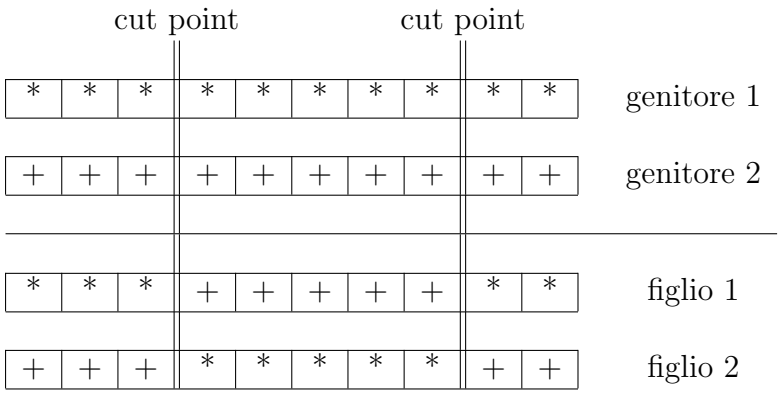


Cut-point crossover: si basa sull'ipotesi che geni vicini controllino caratteristiche tra loro correlate[3], per cui, affinché i figli preservino le caratteristiche dei genitori, è necessario passare *blocchi* di geni contigui. In pratica si considerano due genitori e si definiscono *casualmente* k punti di taglio, $k \geq 1$ ($k-$cut-point crossover). Quindi si ottiene un primo figlio copiando i blocchi definiti dai punti di taglio alternativamente dal primo e dal secondo genitore e, in maniera complementare si ottiene un secondo figlio. Di seguito riportiamo un esempio di $1-$cut-point crossover



e di $2-$cut-point crossover

[3]Questa, peraltro, è una delle ipotesi che permettono di dimostrare l'efficacia degli algoritmi genetici da un punto di vista puramente teorico.



Il crossover fornisce il meccanismo base per la generazione di nuovi e diversi individui. Per renderlo più efficace, è oppurtuno integrarlo con i seguenti accorgimenti.

Operatore di mutazione

Gli individui generati con gli operatori di crossover tendono riportare delle caratteristiche dei genitori, qualora queste siano diffuse nella popolazione. Supponiamo ad esempio che, casualmente, ad una certa iterazione, molti individui abbiano lo stesso valore per il gene $i-$esimo: i figli tenderanno ad avere lo stesso valore per quel gene. Questo fenomeno è noto come assorbimento genetico (*genetic drift*), cioè la convergenza casuale di uno o più geni verso lo stesso valore. Viene quindi introdotto un operatore a complemento del crossover, detto *mutazione*. La mutazione modifica *casualmente* il valore di alcuni geni, scelti altrettanto casualmente.

> **Esempio***: possibile operatore di mutazione su un cromosoma binario.* Si considera separatamente ciascuno degli n geni di una soluzione x e lo si modifica con probabilità p_m:
>
> **for** $i := 1$ **to** n **do**
>
> $p :=$ *rand(0,1)*;
>
> **if** $p \leq p_m$ **then** $x_i := 1 - x_i$ **else** $x_i := x_i$;
>
> **next** i

L'operatore di mutazione ha anche l'importante funzione di contrastare la *convergenza prematura* della popolazione, cioè una situazione nella quale tutti gli individui della popolazione sono tra loro simili. Questo potrebbe facilmente avvenire, nonostante le attenzioni nella selezione dei genitori, proprio perchè il crossover tende a generare figli che somigliano ai genitori e, quindi, con il procedere delle iterazioni che privilegiano comunque gli individui migliori, si somigliano tra loro. La stessa situazione si potrebbe presentare nella popolazione iniziale, sebbene la generazione della popolazione iniziale debba curare la diversificazione degli individui. In ogni caso, è possibile che nessun individuo della popolazione corrente possieda delle caratteristiche che invece sono desiderabili per l'ottimalità

della soluzione. L'operatore di mutazione è quindi utilizzato per introdurre, *in maniera casuale*, delle caratteristiche non possedute dalla popolazione corrente.

Infine, controllando dinamicamente i parametri che regolano la probabilità di mutazione, si potrebbero implementare delle fasi di *diversificazione*: si tratta di introdurre delle misure della convergenza della popolazione verso determinati cromosomi o blocchi di geni, e, sopra una determinata soglia, aumentare la probabilità di mutazione (che di solito assume valori molto bassi, dell'ordine di 10^{-3}), per modificare gli individui generati e ottenere delle soluzioni diverse.

L'operatore di mutazione imita la mutazione dei meccanismi di riproduzione naturale, che avviene durante il crossover dei cromosomi, introducendo delle caratteristiche più o meno "fortunate" in termini di adattamento all'ambiente.

Integrazione con ricerca locale

In natura, il livello di adattamento all'ambiente non dipende solo dal corredo genetico di un individuo, ma anche dalle esperienze, che permettono un ulteriore sviluppo del potenziale genetico e aumentano le capacità di sopravvivere e di entrare nei processi riproduttivi: ad esempio, si mandano i figli a scuola. Un meccanismo simile può essere simulato, a complemento degli operatori di ricombinazione, attraverso una ricerca locale: a partire dal figlio generato, si applica un algoritmo di ricerca locale e si sostituisce il figlio con il corrispondente minimo locale. In questo caso è importante trovare un compromesso tra la qualità delle soluzioni e lo sforzo computazionale. Per questo, di norma, si preferisce applicare operatori di ricerca locale non sistematicamente a tutti i figli, ma soltanto a una selezione (casuale o guidata dalla fitness) di pochi individui della popolazione.

Gestione di offspring non ammissibile

Gli operatori di crossover e mutazione potrebbero generare dei cromosomi corrispondenti a soluzioni non ammissibili: si pensi, banalmente, a un problema dello zaino con codifica binaria. Esistono diversi modi per gestire la presenza di cromosomi corrispondenti a soluzioni non ammissibili, tra cui:

- rifiutare le soluzioni non ammissibili. Il metodo non è molto utilizzato, in quanto potrebbero essere necessari diversi tentativi prima di generare (casualmente) un cromosoma ammissibile;
- accettare la presenza nella popolazione di cromosomi non ammissibili. Come accennato, tali cromosomi potrebbero contenere delle caratteristiche desiderabili e, utilizzati dagli operatori di ricombinazione, potrebbero portare a buone soluzioni. Pertanto questi sono ammessi nella popolazione, ma penalizzati opportunamente dalla funzione di fitness, in relazione al grado di inammissibilità;
- "riparare" i cromosomi non ammissibili. Si tratta di applicare delle tecniche di repairing, specifiche per ciascun problema, che attuino una mutazione forzata di un cromosoma, trasformandolo in soluzione ammissibile.

Esempio*: KP/0-1 con codifica binaria.* Dato un cromosoma che sfora la capacità dello zaino, si eliminano uno a uno gli oggetti nell'ordine inverso del rapporto premio/peso, fino a ottenere un cromosoma ammissibile che entra nella popolazione.

- progettare una codifica e/o degli operatori che garantiscano automaticamente l'ammissibilità dei cromosomi generati. Si tratta della soluzione tipicamente migliore, quando sia possibile implementarla senza eccessivo aggravio computazionale.

  Esempio*: TSP.* Come abbiamo visto, una possibile codifica è data dal cromosoma posizionale corrispondete alla path representation. Affinché un cromosoma sia ammissibile, è necessario e sufficiente che tutti i geni siano diversi tra loro. Tale caratteristica, però, può essere distrutta da un crossover uniforme o da un cut/point crossover, come si vede nel seguente esempio con 10 città:

1	4	9	2	6	8	3	0	5	7	genitore 1
0	2	1	5	3	9	4	7	6	8	genitore 2
1	4	9	**5**	3	**9**	**4**	0	5	7	figlio 1
0	2	1	**2**	**6**	**8**	3	7	6	8	figlio 2

  L'operatore può essere modificato per preservare l'ammissibilità dei figli, ottenendo l'*order crossover*: definiti i due punti di taglio, il figlio 1 (risp. 2) riporta le parti esterne del genitore 1 (risp. 2); i restanti geni si ottengono copiando le città mancanti nell'ordine in cui appaiono nel genitore 2 (risp. 1):

1	4	9	2	6	8	3	0	5	7	genitore 1
0	2	1	5	3	9	4	7	6	8	genitore 2
1	4	9	2	3	6	8	0	5	7	figlio 1
0	2	1	4	9	3	5	7	6	8	figlio 2

  Analogamente, per evitare che la mutazione infici l'ammissibilità, di definisce la *mutazione per inversione*: si generano casualmente due punti della sequenza e si inverte la sottosequenza tra i due punti (corrisponde ad una mossa 2-opt):

1	4	**9**	2	6	**8**	3	0	5	7
	$\longrightarrow$		$\longleftarrow$				$\longrightarrow$		
1	4	**8**	6	2	**9**	3	0	5	7

Si noti come gli operatori **order crossover** e **mutazione per inversione** possano essere utilizzati in tutti i casi in cui la soluzione sia ottenibile come permutazione di elementi (si pensi al caso del KP/0-1 con codifica/decodifica ottenuta come sequenza ordinata degli oggetti).

5.2.7 Sostituzione della popolazione

Ad ogni iterazione, la nuova popolazione è ottenuta considerando la popolazione dell'iterazione precedente e l'offspring generata. Chiaramente, se si aggiungessero semplicemente i nuovi individui, la popolazione crescerebbe esponenzialmente e, pertanto, sono necessarie delle politiche di *population management*.

Solitamente, il numero di individui nelle varie iterazioni viene mantenuto costante, controllato da un parametro N. Non mancano tuttavia casi in cui tale numero viene fatto variare dinamicamente (ad esempio, più alto per diversificare la ricerca e più basso per intensificare). Una volta generati R nuovi individui tramite ricombinazione (potrebbe essere R), le politiche base di population management sono le seguenti:

- *Generational Replacement*: si generano $R = N$ nuovi individui che sostituiscono gli N vecchi individui (imita i sistemi biologici);
- *Steady State*: al contrario della precedente, rimpiazza solo un minimo numero di elementi della generazione precedente, selezionati con criteri guidati dalla fitness (sono tendenzialmente, su base sempre probabilistica, gli individui peggiori a essere sostituiti);
- *Tecniche elistiche*: come generational replacement, ma si mantengono alcuni (poche unità) degli individui con fitness maggiore della popolazione precedente;
- *Selezione dei migliori*: si mantengono nella popolazione corrente gli N migliori individui tra gli $N + R$. La selezione può essere deterministica o probabilistica (si selezionano, con il metodo Montecarlo, N individui tra gli $N + R$, con probabilità proporzionale alla fitness).

Nella pratica, sono spesso utilizzate delle tecniche miste. Inoltre, come abbiamo visto, una delle caratteristiche da preservare nella popolazione è comunque una sufficiente diversificazione. Pertanto, per evitare una prematura convergenza del metodo, l'accettazione di un nuovo individuo nella popolazione potrebbe essere subordinata a una verifica che stabilisca quanto questo sia diverso dagli altri, ad esempio utilizzando la distanza di Hamming come misura di diversità (potrebbe essere inutile, soprattutto in fasi di diversificazione, inserire nella popolazione un individuo il cui cromosoma sia esattamente lo stesso di un altro già presente). In ogni caso, una soglia di "diversità" gestita dinamicamente potrebbe essere impiegata per implementare fasi di intensificazione e diversificazione.

5.2.8 Criteri di arresto

Possibili criteri arresto per gli algoritmi genetici sono i seguenti (o loro combinazioni):

- si fissa un numero massimo $K_{\max}$ di iterazioni (o generazioni);
- si fissa un tempo di esecuzione limite $T_{\max}$;
- si considera un massimo numero $K'_{\max}$ di generazioni senza miglioramento: si ferma l'algoritmo se l'ultimo individuo che migliorava la funzione obiettivo è stato trovato $K'_{\max}$ iterazioni prima;
- convergenza della popolazione: è un criterio proprio degli algoritmi basati su popolazione e consiste nel verificare se la popolazione è sufficientemente differenziata. Misure di convergenza possono essere una scarsa varianza della fitness degli individui (misura indiretta) o la somiglianza dei cromosomi della popolazione (misura diretta, ad esempio la solita distanza di Hamming). Ovviamente, questo criterio potrebbe essere utilizzato per interrompere una fase di intensificazione e innescare una diversificazione.

5.2.9 Scelte implementative

Come per tutte le metaeuristiche, lo schema base degli algoritmi genetici è piuttosto semplice e le prestazioni effettive dipendono fortemente dalle scelte implementative delle diverse componenti sopra elencate.

Per un algoritmo genetico, inoltre, un altro fattore determinante per le prestazioni è il valore degli svariati parametri che lo caratterizzano, tra i quali la dimensione della popolazione, gli eventuali pesi per le penalità della fuzione di fitness e, soprattutto, la *probabilità* di applicazione dei diversi operatori genetici.

In effetti, gli algoritmi genetici sono fortemente basati sulla componente probabilistica: come abbiamo messo in evidenza, praticamente tutti gli operatori hanno un comportamento non deterministico, controllato da opportuni parametri. Questa caratteristica rappresenta, al contempo, il punto di forza e il punto debole degli algoritmi genetici. Da un lato, infatti, li rende estremamente *robusti e adattabili* a diversi problemi: se la popolazione iniziale è sufficientemente diversificata da rappresentare un buon campionamento dello spazio delle soluzioni, un algoritmo genetico è in grado di trovare delle buone soluzioni per praticamente tutti i problemi di ottimizzazione combinatoria (o anche per altre classi di problemi). Di qui la notevole diffusione del loro utilizzo in svariati campi. Dall'altro lato, però, la presenza di numerosi parametri e l'intervento della casualità rendono gli algoritmi genetici molto *poco controllabili*: in effetti, una delle fasi più impegnative è la calibrazione dei parametri, cioè la scelta di valori standard che siano applicabili direttamente a tutte le istanze di uno stesso problema, il che richiede un test accurato, come accenneremo in seguito. Tale fase, per un algoritmo genetico è estremamente importante, ma viene spesso lasciata all'utente, che si limita a cercare la configurazione

dei parametri ideale per il singolo problema, ripetendo svariate volte lo stesso algoritmo: anche se il singolo run è rapido, l'utente spende comunque molto tempo (tanto valeva utilizzare metodi diversi, dando loro maggiore tempo di calcolo a disposizione).

Un altro motivo della diffusione degli algoritmi genetici è la loro classificazione come *weak methods* (o *soft computing*), ad indicare il fatto che la loro implementazione e le loro prestazioni richiedono poche conoscenze a priori del problema. In effetti, le uniche componenti che dipendono necessariamente dal problema sono la codifica/decodifica dei cromosomi e la relativa valutazione della fitness, mentre per gli altri operatori genetici si possono utilizzare implementazioni standard (ad esempio, utilizzando una codifica binaria, gli operatori si riducono a manipolatori di stringhe binarie). Per questo motivo, esistono molte "librerie" di algoritmi genetici e, inoltre, molti pacchetti di ottimizzazione "general purpose" sono basati su algoritmi genetici.

Un ultimo cenno va all'importanza di alternare, negli algoritmi genetici come peraltro in tutte le metaeuristiche, fasi di intensificazione e di diversificazione. Queste possono essere implementate nei diversi operatori genetici, come sopra descritto e di seguito riassunto a titolo puramente esemplificativo:

- variando dinamicamente la probabilità di mutazione;
- introducendo opportune penalità nella fitness, per penalizzare o incentivare (con pesi dinamici, cioè variabili durante le iterazioni) individui con determinate caratteristiche;
- legando la probabilità di accettazione di un nuovo individuo non solo alla fitness, ma anche al suo grado di diversità dai restanti individui;
- aumentando, per intensificare, il numero di individui sottoposti a ricerca locale dopo la loro generazione.

6 Le prestazioni degli algoritmi metaeuristici

Come abbiamo visto, dato un problema di ottimizzazione combinatoria, esistono diversi approcci risolutivi e, all'interno dello stesso approccio, diverse scelte implementative. Inoltre, per molte metaeuristiche, è necessario stabilire i valori dei parametri che ne determinano il comportamento (lunghezza della tabu list, profili di raffreddamento, probabilità di applicazione di operatori genetici etc.). Le scelte implementative e la determinazione dei parametri sono fattori che concorrono a determinare le prestazioni di un algoritmo e devono essere effettuate *entrambe* con attenzione, sulla base di opportune valutazioni.

6.1 Valutazione degli algoritmi

Le scelte progettuali richiedono la definizione di criteri che devono considerare diversi aspetti, tra i quali:

- semplicità implementativa, considerando le risorse (economiche, di tempo, di personale) disponibili;
- i tempi di calcolo, cioè l'efficienza computazionale dell'algoritmo, considerando il tempo effettivamente disponibile per trovare soluzioni;
- la qualità delle soluzioni ottenute, cioè la "bontà" (o efficacia) dell'algoritmo;
- per gli algoritmi con componente probabilistica, la robustezza o affidabilità dell'algoritmo (*reliability*), cioè la capacità di produrre buone soluzioni in ogni run, indipendentemente dalle particolari scelte casuali.

In particolare, la misura della "bontà" di un algoritmo euristico è un fattore critico: riportiamo di seguito una sintesi di possibili criteri e modalità di valutazione.

6.1.1 Analisi sperimentale

Si basa sull'effettiva implementazione su elaboratore dell'algoritmo, in modo da poter effettuare una sperimentazione. I passi sono i seguenti:

1. implementazione dell'algoritmo;
2. selezione di un *opportuno* insieme di istanze (casi specifici) del problema. Le istanze possono essere quelle reali, e/o generate casualmente, e/o dei benchmark standard forniti dalla letteratura. La scelta del campione dipende dagli scopi della valutazione: ad esempio, se vogliamo vedere il comportamento dell'algoritmo in una specifica azienda, sarà opportuno considerare molte istanze reali; se vogliamo dimostrare che il nostro algoritmo è migliore di altri in generale, sarà necessario riferirsi a benchmark standard; se vogliamo testare la robustezza, sarà opportuno inserire nel campione diverse istanze generate casualmente;
3. si effetuano i test, registrando, per ogni esecuzione, la valutazione delle soluzioni prodotte e i tempi di calcolo richiesti. Nel caso di presenza di parametri (praticamente sempre per le metaeuristiche), è opportuno anteporre una fase di calibrazione dei parametri stessi (come di seguito descritto) e utilizzare la stessa definizione dei parametri per tutte le prove. Inoltre, se l'algoritmo non è deterministico, bisogna considerare un numero congruo di esecuzioni[4] su ogni istanza, e valutare dei valori di performance medi, o delle statistiche più accurate, che includano la robustezza[5];
4. confronto dei risultati ottenuti: la funzione obiettivo si confronta con il valore della soluzione ottima (quando questo è noto) oppure con dei bound o con le prestazioni di algoritmi alternativi, ottenendo delle misure relative di bontà. Allo stesso modo si possono fare delle valutazioni relative dei tempi di calcolo.

[4]Considerando che i linguaggi di programmazione mettono a disposizione generatori *pseudo*casuali, ogni esecuzione deve utilizzare un seme casuale diverso.

[5]In genere, come misura di robustezza, si usano media e deviazione standard degli indicatori di performance (valore della funzione obiettivo, tempi di esecuzione etc.).

Questo approccio è sempre praticabile e abbastanza semplice, almeno concettualmente, e spesso è quello praticato, anche se le conclusioni che se ne traggono non sono valide in generale, visto che l'analisi dipende fortemente dalle istanze considerate.

6.1.2 Analisi probabilistica

Si basa sul concetto di *istanza media* del problema, espressa come distribuzione di probabilità sulla classe di tutte le istanze possibili. Il tempo di esecuzione ed il valore della soluzione vengono trattati come variabili aleatorie, di cui si studia la tendenza, generalmente al tendere delle dimensioni delle istanze all'infinito secondo una certa distribuzione di probabilità (generalmente uniforme). Questo approccio ha forti fondamenti teorici ma spesso risulta impraticabile (possibile solo per algoritmi molto semplici). Inoltre, non è ben certa l'estendibilità dell'approccio ai casi reali, visto che l'effettiva distribuzione di probabilità dei dati potrebbe essere non nota o troppo complicata per essere trattata analiticamente.

6.1.3 Analisi del caso peggiore

Si basa sulla determinazione della massima deviazione (assoluta e relativa) che la soluzione prodotta dall'algoritmo può avere rispetto alla soluzione ottima. L'analisi viene condotta relativamente alle condizioni di caso peggiore per l'algoritmo. Il risultato ottenuto è molto forte e di validità generale (algoritmi a performace garantita), anche se potrebbe essere difficile derivarlo. Inoltre, spesso, le indicazioni risultanti sono molto pessimistiche rispetto al comportamento medio dell'algoritmo.

6.2 Calibrazione dei parametri

Come abbiamo visto, tutte le metaeuristiche prevedono dei parametri che permettono la loro adattabilità ai diversi problemi o diverse classi di istanza. È necessario quindi procedere alla loro calibrazione (*parameter calibration* o *estimation*): per ogni parametro se ne determina il valore, che può essere assoluto (popolazione fatta da $N = 100$ individui) o funzione di qualche caratteristica misurabile dell'istanza (popolazione fatta da $N = 10n$ individui, dove n è la dimensione dell'istanza, ad esempio in numero di città in un TSP). Si vuole sottolineare che è buona norma fissare una volta per tutte i valori (assoluti o funzioni), e non adattarli a ogni singola istanza, altrimenti si rischia di spendere tempo a "ottimizzare i parametri" (spesso con ricerche pseudo-esaustive, se non alla cieca) per ottimizzare un singolo problema.

Le tecniche di calibrazione dei parametri sono diventate recentemente oggetto di ricerca, e variano da tecniche a scatola nera per individuare i parametri che garantiscono le migliori performance (*black box optimization*), a tecniche di adattamento automatico dei parametri (*adaptivity*), coinvolgendo domini interdisciplinari, come l'intelligenza artificiale. In questa sede, comunque, ci limitiamo ad accennare alle tecniche standard, di semplice realizzazione.

Si tratta, in sostanza, di effettuare delle prove ripetute con set di parametri diversi su un insieme ridotto di istanze (istanze di test), tale da rendere i tempi per i test ragionevoli. Le prove sono valutate con i criteri visti sopra, di modo da scegliere i set di parametri che sperimentalmente garantiscono le performance migliori sulle istanze di test.

I parametri da calibrare sono *generalmente* pochi per le metaeuristiche a traiettoria, mentre tendono a essere molti per le metauristiche a popolazione. Ovviamente, i parametri interagiscono tra loro nel determinare le prestazioni, di modo che la difficoltà della calibrazione cresce esponenzialmente con il numero di parametri. Inoltre, un fattore che complica ulteriormente la calibrazione è la presenza di componenti casuali, che rendono più difficile interpretare l'influenza sulle prestazioni della variazione di un parametro, visto che la prestazione stessa potrebbe semplicemente dipendere dal caso. Ad esempio, la calibrazione degli algoritmi genetici potrebbe essere il vero passaggio critico nel loro utilizzo, mentre la loro implementazione può essere relativamente semplice.

7 Conclusioni: metaeuristiche ibride

In queste dispense sono state date solo alcune delle possibili metaeuristiche per ottimizzazione combinatoria. L'approccio va però inteso in senso molto più flessibile, e quelli proposti sono solo spunti per scelte progettuali che devono essere adattate e rimesse in discussione in funzione del particolare problema da risolvere. In questo senso possiamo interpretare lo sviluppo negli ultimi anni di *metaeuristiche ibride*, che cercano di coniugare i pregi di diversi schemi algoritmici. L'ibridazione può avvenire a vari livelli e secondo schemi diversi, e una loro trattazione richiederebbe un approfondimento che è fuori dagli scopi del corso e si rimanda a testi specifici. Forniamo di seguito solo alcuni esempi di possibili ibridazioni, che di solito danno luogo a schemi algoritmici più potenti:

- la presenza della ricerca locale negli operatori genetici di ricombinazione è un esempio in questa direzione, che cerca di limitare l'eccessiva influenza della casualità e aumentarne il livello di controllabilità. L'idea può essere estesa considerando un metodo in cui un algoritmo genetico viene utilizzato per generare delle soluzioni "buone" da considerare come punti di partenza di metodi a traiettoria (ad esempio delle tabu search);

- una tabu search potrebbe prevedere, in alcune fasi, l'accettazione di vicini non miglioranti su basi probabilistiche, in analogia con la simulated annealing;

- sono anche possibili ibridazioni tra metaeuristiche e metodi esatti, nei due sensi. Ad esempio, gli algoritmi di branch-and-cut, integrano al loro interno euristiche per trovare buone soluzioni ammissibili. Oppure si possono progettare dei vicinati da esplorare in modo esatto, sfruttando l'efficienza dei moderni solver per programmazione a numeri interi;

- etc. etc. etc.