Appunti sui metodi metaeuristici di ricerca

A. Agnetis[*], C. Meloni[†]

1 Introduzione

Nell'affrontare un problema di ottimizzazione, possono essere utilizzati vari approcci, a seconda sia della difficoltà specifica e delle dimensioni del problema in esame, sia degli obiettivi reali che si vogliono ottenere. Cosi', in quei casi in cui è necessario pervenire alla soluzione ottima di un certo problema, si utilizzerà un approccio di enumerazione implicita, sia esso basato su una formulazione di programmazione intera, o sullo sfruttamento delle proprietà combinatorie del problema. Se è invece sufficiente avere una garanzia sul massimo errore commesso, in termini relativi, si potrà far ricorso ad un algoritmo ρ-approssimato, ammesso che ve ne siano per il particolare problema in esame. Infatti, mentre certi metodi esatti (come i piani di Gomory) possono applicarsi, almeno in linea di principio, a qualsiasi problema di programmazione lineare a numeri interi, per quanto riguarda gli algoritmi approssimati non esiste, per ora, un approccio sistematico allo stesso livello di generalità.

Una strada diversa, che può diventare l'unica percorribile se le dimensioni e/o la complessità del problema sono elevate, è quella che consiste nel cercare soluzioni di tipo *euristico*, ottenute cioè applicando un algoritmo concepito in modo da produrre soluzioni che si sperano buone, ma senza garanzia a priori sulla vicinanza all'ottimo. Un algoritmo euristico (o, semplicemente, un'*euristica*) deve essere in grado di produrre una soluzione in tempo relativamente breve. Mentre chiaramente è possibile progettare euristiche specifiche per qualunque problema di ottimizzazione combinatoria, negli ultimi anni hanno acquisito importanza via via maggiore alcuni approcci euristici di tipo generale, detti *metaeuristiche*. La struttura e l'idea di fondo di ciascuna metaeuristica sono sostanzialmente fissate, ma la realizzazione delle varie componenti dell'algoritmo dipende dai singoli problemi. Spesso peraltro queste metaeuristiche traggono ispirazione (se non legittimazione) da alcune analogie con la natura fisica. Ad esempio, nel *simulated annealing* si paragona il processo di soluzione di un problema di ottimizzazione combinatoria, in

[*]Dipartimento di Ingegneria dell'Informazione - Università di Siena
[†]Dipartimento di Informatica e Automazione - Università di Roma Tre

cui si passa iterativamente da una soluzione ammissibile a un'altra, via via migliorando la funzione obiettivo, al processo con cui un solido, raffreddandosi, si porta in configurazioni con via via minore contenuto di energia. Negli *algoritmi genetici*, un insieme di soluzioni ammissibili è visto come un insieme di individui di una popolazione che, accoppiandosi e combinandosi tra loro, danno vita a nuove soluzioni che, se generate in base a criteri di miglioramento della specie, possono risultare migliori di quelle da cui sono state generate.
Gli approcci metaeuristici possono vedersi in realtà in modo omogeneo, come generalizzazioni di un unico approccio fondamentale, che è quello della *ricerca locale*. La ricerca locale si basa su quello che è, per certi versi, l'approccio più semplice e istintivo all'ottimizzazione: andare per tentativi. In effetti, l'idea di funzionamento della ricerca locale (come del resto quella delle varie metaeuristiche) è talmente elementare che è sorprendente constatare la loro relativa efficacia. Va tuttavia detto fin da ora che a fronte di questa relativa semplicità, l'applicazione di una qualunque metaeuristica a un problema di ottimizzazione combinatoria richiede comunque una messa a punto accurata e talvolta laboriosa.
Consideriamo un problema di minimizzazione, e una sua soluzione ammissibile x, con associato il valore della funzione obiettivo $f(x)$. La ricerca locale consiste nel definire un *intorno* di x (detto, nella terminologia della ricerca locale, *vicinato*), e nell'esplorarlo in cerca di soluzioni migliori, se ve ne sono. Se, in questo vicinato di x, si scopre una soluzione y per cui $f(y) < f(x)$, allora ci si sposta da x a y e si riparte da y con l'esplorazione del suo intorno. Se invece nel vicinato di x non si scopre nessuna soluzione migliore, allora vuol dire che x è un *minimo locale*. Nella ricerca locale *classica*, arrivati in un minimo locale, l'algoritmo si ferma e restituisce questo minimo come valore di output. Ovviamente, non si ha nessuna garanzia in generale che tale valore costituisca una soluzione ottima del problema; anzi, tipicamente esso può essere anche molto distante dall'ottimo globale. Le metaeuristiche, in effetti, sono state concepite proprio per cercare di ovviare al problema di rimanere *intrappolati* in un minimo locale.
Vediamo dapprima più da vicino la ricerca locale, e poi esamineremo gli approcci metaeuristici.

2 Ricerca locale

In base a quanto detto sopra, possiamo riassumere schematicamente l'algoritmo generale di ricerca locale come segue. Con x indichiamo una generica soluzione ammissibile del problema, $N(x)$ il suo vicinato, e $f(x)$ indica la funzione obiettivo.

Algoritmo di ricerca locale

1. Scegli una soluzione iniziale x;
2. Genera le soluzioni nel vicinato $N(x)$;
3. Se in $N(x)$ c'è una soluzione y tale che $f(y) < f(x)$, allora poni $x := y$ e vai a 2, altrimenti STOP.

Lo spostamento da x a y al passo 3 viene spesso chiamato *mossa*. Nella ricerca locale, come si vede, si ha un miglioramento della funzione obiettivo in corrispondenza di una mossa.
Per applicare l'approccio della ricerca locale a un particolare problema, bisogna fare alcune scelte di fondo.

- Occorre avere a disposizione una soluzione iniziale ammissibile. Questa può essere generata da un'euristica ad hoc per il particolare problema, o addirittura può essere generata casualmente. È possibile anche eseguire l'algoritmo a partire da diverse soluzioni iniziali, ottenendo cosi' diverse soluzioni euristiche, e scegliere poi la migliore.
- Bisogna definire in modo preciso e opportuno il vicinato di una soluzione.
- Definito il vicinato, occorre avere a disposizione un algoritmo efficiente per esplorarlo. Non è utile definire il vicinato in un modo teoricamente potente ma non essere in grado di esplorarlo in un tempo di calcolo ragionevole.

A seconda di come vengono fatte queste scelte, l'algoritmo risultante può essere più o meno efficiente, e la soluzione proposta dalla ricerca locale risultare più o meno buona. Vediamo alcuni esempi di come applicare questi concetti a problemi di ottimizzazione combinatoria.

2.1 Ricerca locale per il TSP

In questa sezione, per semplicità ci riferiamo al TSP simmetrico, ma con alcune modifiche lo stesso approccio può applicarsi anche al caso asimmetrico. Una soluzione ammissibile, com'è noto, è costituita da un ciclo hamiltoniano su un grafo completo con n nodi (le n città). La definizione più classica e più nota di vicinato per il TSP è quella che fa uso dei cosiddetti k-scambi: partendo da una soluzione ammissibile, il vicinato è costituito da tutte le soluzioni ottenute cancellando k archi (non adiacenti) e sostituendoli con altri k in modo da ricreare un ciclo. Questo tipo di vicinato viene chiamato *k-opt*. Particolarmente

semplici e interessanti sono i casi in cui $k = 2$ e $k = 3$. Nel primo caso, data la soluzione corrente x, si tratta di cancellare due archi del ciclo, siano essi (i, j) e (h, k), e di sostituirli con altri due, ossia (i, h) e (k, j). Indicando con c_{ij} il costo dell'arco (i, j), chiaramente la nuova soluzione sarà conveniente se $c_{ih} + c_{kj} < c_{ij} + c_{hk}$. Poiche' gli archi in un ciclo hamiltoniano sono n, la dimensione del vicinato di una soluzione (e dunque il tempo necessario a esplorarlo) è $O(n^2)$. Tale complessità è ancora accettabile, e dunque un algoritmo di ricerca locale basato su questo vicinato risulta abbastanza efficiente.
Chiaramente, all'aumentare di k aumenta anche la grandezza del vicinato, e dunque il tempo di calcolo necessario a esplorarlo, che sarà ovviamente $O(n^k)$. Tuttavia, se la grandezza di un vicinato aumenta, si può sperare che aumenti pure l'efficacia del singolo passo di ricerca locale, ossia che, a parità di numero complessivo di passi, la soluzione cui si perviene sia buona. Trovare il miglior compromesso tra complessità dell'esplorazione del vicinato e qualità della soluzione è uno dei punti chiave di un approccio di ricerca locale. Nel caso del TSP, alcuni esperimenti fatti da Lin alla fine degli anni '50 hanno mostrato che, passando da $k = 2$ a $k = 3$ si ha un miglioramento sensibile nella qualità della soluzione trovata, che giustifica pienamente l'aumento di carico computazionale. Invece passando a $k = 4$ tale maggiore complessità appare abbastanza inutile, dal momento che i miglioramenti rispetto alla soluzione fornita da $k = 3$ risultano piuttosto piccoli. Perchè ciò accade non è ancora del tutto chiaro. Un altro interessante contributo di Lin è stato quello di utilizzare, come soluzioni di partenza, permutazioni generate casualmente. Dai suoi esperimenti condotti su un grafo con $n = 48$ nodi emergeva che, in questo modo, la soluzione ottenuta alla fine dell'algoritmo di ricerca locale, in 5 casi su 100 era ottima. Questo vuol dire che, iterando r volte la ricerca locale a partire da punti di partenza diversi, la probabilità di trovare la soluzione ottima è pari a $1 - 0.95^r$, il che, ad esempio, vuol dire che partendo da $r = 100$ punti iniziali diversi, si ha una probabilità di trovare l'ottimo pari a 0.99. Purtroppo, come osservato empiricamente, la probabilità di trovare l'ottimo diminuisce sensibilmente al crescere di n.

2.2 Ricerca locale per la partizione uniforme di grafi

Si consideri un grafo $G = (V, E)$ non orientato, con $|V| = 2n$ nodi, pesato sugli archi con pesi d_{ij}. Una partizione (A, B) dell'insieme dei nodi si dice *uniforme* se $|A| = |B|$. Il *costo* di una partizione uniforme è il peso complessivo degli archi che non sono interamente contenuti in una classe della partizione, ovvero degli archi che cadono *a cavallo* dei due insiemi. Trovare la partizione uniforme di costo minimo è stato dimostrato essere un problema NP–completo (si noti che risolvendo un problema di massimo flusso possiamo trovare un lower bound al valore della soluzione ottima). Vogliamo qui illustrare una

tecnica di ricerca locale (ideata da Lin e Kernighan) che definisce il vicinato di una soluzione in modo più raffinato rispetto a quanto visto prima.
Data una partizione uniforme (A, B), si consideri un nodo $a \in A$ e uno $b \in B$. Scambiandoli tra i due insiemi, si ottiene una nuova partizione uniforme. L'operazione che consiste nello scambiare due nodi prende il nome di *swap*. Appare abbastanza naturale allora definire il vicinato di una partizione (A, B) come l'insieme delle partizioni ottenibili per mezzo di uno swap, in tutti i modi possibili. La dimensione di questo vicinato è, evidentemente, $O(n^2)$. Questa scelta ha dato luogo a risultati abbastanza soddisfacenti. Se ora vogliamo ampliare la definizione di vicinato cercando di migliorare la qualità della soluzione trovata, la cosa più intuitiva potrebbe essere quella che consiste nello scambiare due nodi di A con due nodi di B, dando cosi' luogo a un vicinato che può essere esplorato in $O(n^4)$. Questo è però un pò dispendioso dal punto di vista computazionale. Lin e Kernighan hanno allora proposto di definire il vicinato in modo diverso e piuttosto articolato, come segue. Data una soluzione iniziale (A, B), si considerino tutte le soluzioni ottenibili da essa tramite un singolo swap. La migliore di esse sia quella ottenuta scambiando i nodi $a_1 \in A$ e $b_1 \in B$. Sia allora $A' = A - a_1 \cup b_1$ e $B' = B - b_1 \cup a_1$. Il costo di (A', B') può essere sia migliore che peggiore rispetto al costo di (A, B). *In ogni caso*, si consideri la nuova partizione (A', B'), e, *senza più scambiare i nodi* a_1 *e* b_1, a partire da questa si effettuino nuovamente tutti gli swap possibili (tra gli insiemi $A' - a_1$ e $B' - b_1$) alla ricerca della partizione migliore. Una volta trovata tale partizione (A'', B''), in corrispondenza dei nodi a_2 e b_2, si fissano anche questi nodi e si procede con i nuovi scambi, e cosi' via. Alla fine, si saranno individuate n partizioni (A', B'), (A'', B''),...,$(A^{(n)}, B^{(n)}$, dove chiaramente $(A^{(n)} = B$ e $B^{(n)} = A$. Tra tutte le partizioni cosi' generate si sceglie la migliore, e da essa si ricomincia l'intero procedimento. L'algoritmo si arresta quando, tra tutte le n partizioni individuate nei passi intermedi, nessuna risulta strettamente migliore di quella attuale. Si noti che la complessità del singolo passo di ricerca locale in questo caso è $O(n^3)$. La ricerca locale di Lin e Kernighan ha dato risultati sperimentali molto brillanti sul problema della partizione uniforme. Inoltre, essa è applicabile in modo abbastanza naturale a tutti quei problemi (anche non su grafi) in cui vi sono da allocare oggetti tra due risorse, come avviene in certi problemi di scheduling.
Si noti che, partendo da (A, B), può benissimo accadere che un singolo swap non porti alcun miglioramento, mentre nel secondo passo intermedio si giunga a individuare una partizione migliore. Questo meccanismo può essere dunque considerato un primo tentativo per evitare di rimanere imprigionati in un minimo locale.

2.3 Scelte realizzative nella ricerca locale

Benché ogni applicazione del concetto di ricerca locale a un problema abbia le sue peculiarità, alcuni aspetti sono abbastanza comuni a molte realizzazioni.

- La definizione del vicinato è in genere connessa al concetto di *perturbazione* di una soluzione ammissibile, nel senso che la ricerca avviene tra le soluzioni che si ottengono *perturbando* una soluzione iniziale. Nel caso del TSP visto sopra, come in molti problemi di scheduling, lo scambio di posizione tra due oggetti è spesso la scelta più naturale. Nel caso della partizione uniforme, come abbiamo visto, la scelta è molto meno scontata.

- Alcuni autori parlano di *forza* di un vicinato. Un vicinato definito in un certo modo è tanto più forte quanto più la qualità delle soluzioni prodotte dall'algoritmo è indipendente dalla bontà della soluzione di partenza. Ad esempio, 3-opt per il TSP è considerato un vicinato forte (almeno per grafi fino a 50 nodi): questo fatto implica che si può affrontare il problema senza perdere tempo a generare buone soluzioni iniziali. Anzi, conviene generare *molte* soluzioni iniziali casualmente, sperando cosi' di avere un campionamento rappresentativo dell'intera regione ammissibile.

- Un altro aspetto caratterizzante un approccio di ricerca locale è il *modo* in cui deve essere esplorato il vicinato di una soluzione. Due strategie antitetiche sono *first improvement* e *steepest descent*. Nel primo caso l'esplorazione del vicinato termina non appena si trova una soluzione migliore di quella corrente. Nel secondo, invece, lo si esplora comunque tutto cercando il massimo miglioramento che quel vicinato consente di ottenere. In genere si preferisce il primo approccio, ma non è una regola fissa.

Come già osservato, un grosso pregio della ricerca locale sta nella sua semplicità concettuale e realizzativa, pur consentendo tale metodo di ottenere risultati interessanti. Tuttavia, il criterio di arresto dell'algoritmo di ricerca locale visto finora appare in generale troppo rigido. A parte alcuni casi particolari (e fortunati) in cui la funzione obiettivo ha determinate caratteristiche di convessità, nella grande maggioranza dei problemi reali la funzione obiettivo da minimizzare presenta un grande numero di minimi locali, talora anche molto lontani dal minimo globale. In effetti, una delle fortunate eccezioni è il metodo del simplesso per la programmazione lineare. Il metodo del simplesso è infatti un algoritmo di ricerca locale, in cui si visitano le basi ammissibili di un sistema del tipo $Ax = b$, e il vicinato è dato da tutte le basi che differiscono per una sola variabile dalla

base corrente. La convessità della funzione obiettivo ci garantisce che il minimo locale trovato dal simplesso è anche globale.
A partire dalla metà degli anni '80, la ricerca si è indirizzata verso approcci euristici che generalizzano la ricerca locale cercando di ovviare ai suoi principali difetti.

3 Tabu search

Nella ricerca locale *classica*, ogni qual volta si esplora il vicinato di una soluzione, l'unica informazione relativa alla storia dell'algoritmo fino a quel momento è la migliore soluzione corrente e il corrispondente valore della funzione obiettivo. L'idea della *tabu search* è invece quella di mantenere una *memoria* di alcune informazioni sulle ultime soluzioni visitate, orientando la ricerca in modo tale da permettere di uscire da eventuali minimi locali. Più precisamente, vedremo che la struttura del vicinato di una soluzione *varia* da iterazione a iterazione.
Come in tutti gli algoritmi di ricerca locale, se, nell'esplorazione del vicinato $N(x)$ di una soluzione x, si scopre una soluzione migliore dell'ottimo corrente, si ha una transizione su quella soluzione e si inizia a esplorare quel vicinato. Tuttavia, se la soluzione corrente è un minimo locale, l'algoritmo di ricerca locale si fermerebbe. E' chiaro allora che, per poter sfuggire ai minimi locali, è necessario accettare mosse che possano portare a un *peggioramento* della soluzione corrente, pur di spostarsi sufficientemente lontano dal minimo locale.
Supponendo allora di trovarci in un minimo locale x, dobbiamo anzitutto scegliere su quale soluzione spostarci, dal momento che nessuna, in $N(x)$, sarà migliore di x. Negli algoritmi di tabu search si fa in genere la scelta – abbastanza logica – di spostarsi sulla soluzione $y \in N(x)$ per la quale è minimo il peggioramento della funzione obiettivo. A questo punto però nasce il problema che sta alla base dell'idea di tabu search. Se ci portiamo sulla soluzione y, ed esploriamo il suo vicinato, è assai probabile che la migliore soluzione in $N(y)$ risulti essere proprio x, cioè quella da cui vogliamo viceversa sfuggire. Più in generale, dal momento che stiamo permettendo anche mosse che non migliorano la funzione obiettivo, può sempre presentarsi il pericolo, anche dopo un certo numero di mosse, di tornare su una soluzione già visitata. Ecco allora che entra in gioco l'uso della memoria. L'idea è quella di utilizzare le informazioni sulle ultime mosse effettuate per evitare il pericolo di ciclaggio. Precisamente, mantenendo memoria delle ultime mosse che hanno portato alla soluzione attuale, sarà possibile *proibire* quelle mosse per un certo tempo al fine appunto di prevenire ricadute in punti già visitati[1].

[1] A proposito dell'esempio del metodo del simplesso, osserviamo che utilizzando la regola di Bland

Dunque, mentre nella ricerca locale si parla di vicinato $N(x)$ di una soluzione x, nella tabu search possono essere escluse, dall'esplorazione di $N(x)$, alcune soluzioni, a seconda dell'itinerario seguito per giungere in x. Diciamo allora che l'insieme delle soluzioni da esplorare dipende, oltre che dalla soluzione x, anche dall'iterazione k del metodo, ossia il vicinato diviene $N(x, k)$.

Per realizzare questo concetto, si fa uso di uno strumento chiamato *tabu list* T. La tabu list è una coda in cui vengono memorizzate le ultime $|T|$ mosse effettuate. Le mosse opposte a queste – che potrebbero dunque avere l'effetto di riavvicinarsi a soluzioni già visitate – vengono proibite nella situazione attuale, cioè vengono escluse dal vicinato della soluzione corrente tutte le soluzioni ottenibili per mezzo di una mossa tabu. Si noti che si potrebbe pensare di memorizzare direttamente le ultime $|T|$ soluzioni, ma questo può essere estremamente poco pratico per problemi di grandi dimensioni. Si preferisce allora memorizzare direttamente le mosse. Ad esempio, in un algoritmo per il TSP, anziché memorizzare $|T|$ cicli hamiltoniani, basta memorizzare le $|T|$ coppie di archi che sono state oggetto dello scambio (in un vicinato di tipo 2-opt). Se cioè abbiamo appena cancellato due archi (i, j) e (h, k) e aggiunto (i, h) e (k, j), questi archi non verranno scambiati tra di loro per un numero di mosse dato dalla *lunghezza* $|T|$ della tabu list.

La tabu list è una coda, nel senso che, a ogni passo, la mossa che era nella tabu list da maggior tempo viene cancellata, e torna a essere quindi ammessa. Anche se il ruolo della tabu list dovrebbe essere abbastanza chiaro, non è in generale ovvio stabilire quale debba essere la lunghezza più appropriata per la tabu list. Infatti, una tabu list troppo lunga potrebbe inutilmente vincolare il processo di ricerca anche quando la distanza percorsa da una determinata soluzione è già sufficientemente elevata da rendere improbabile un ritorno su tale soluzione. D'altro canto, una tabu list troppo corta può presentare invece il problema opposto, cioè potrebbe rendere possibile il ciclaggio. Il valore ottimale della lunghezza della tabu list varia tipicamente da problema a problema – anzi, talora addirittura da istanza a istanza – ma comunque raramente eccede il valore dieci[2].

Rispetto a un algoritmo di ricerca locale, va definito il criterio di arresto, dal momento che esso non è più dato dall'evento per cui la funzione obiettivo non migliora. In pratica si utilizzano i seguenti criteri di arresto:

---

si evita il pericolo del ciclaggio, senza bisogno di memorizzare le basi visitate. Tuttavia, un approccio forse meno sofisticato, consistente nel memorizzare le basi già visitate, in linea di principio sarebbe perfettamente lecito.

[2]Di fronte all'evidenza empirica che in molti casi il valore più efficace della lunghezza della tabu list è pari a 7, alcuni tra i creatori della tabu search hanno voluto ravvisare una corrispondenza tra tale valore e quello che indica la capacità della "memoria di brevissimo termine" degli umani, quella ad esempio che utilizziamo quando dobbiamo memorizzare per pochi secondi un numero di telefono senza avere carta e penna. Giustificare in questo modo il fatto che la tabu search funzioni bene quando $|T| = 7$ appare tanto suggestivo quanto arbitrario.

- il numero di iterazioni raggiunge un valore k_{max} prefissato;
- il numero di iterazioni dall'ultimo miglioramento della funzione obiettivo raggiunge un valore k'_{max} prefissato;
- è possibile certificare che l'ottimo corrente è l'ottimo globale.

Un ultimo *ingrediente* di un algoritmo di tabu search nasce dalla seguente osservazione. A un generico passo dell'algoritmo, si hanno un insieme di mosse tabu. Può però accadere che la soluzione in cui ci si porterebbe applicando una di queste mosse tabu abbia delle caratteristiche che la rendono in qualche modo interessante. In tal caso si esegue la mossa, nonostante fosse nella tabu list. Tali caratteristiche vengono chiamate *criteri di aspirazione*. Quello di gran lunga più utilizzato è quello per cui una mossa tabu può essere forzata (*overruled*) se la soluzione cui essa dà luogo ha un valore di funzione obiettivo migliore dell'ottimo corrente (dunque tra l'altro si tratta certamente di una soluzione mai visitata).
A questo punto possiamo riassumere la struttura di un algoritmo di tabu search (che come è evidente generalizza la ricerca locale):

Algoritmo di tabu search

1. Scegli una soluzione iniziale x; $k := 0$; inizializza la tabu list $T = \emptyset$;
2. Incrementa il contatore k e genera le soluzioni nel vicinato $N(i, k)$;
3. Trova la soluzione y per cui $f(y)$ (o un'altra funzione $\tilde{f}(y)$) è minima con $y \in N(x, k), y \neq x$ e che non viola nessun tabu oppure che soddisfa un criterio di aspirazione;
4. Se $f(y) < f(x)$, poni $x^* := y$;
5. aggiorna T inserendo la mossa che fa passare da x a y e poni $x := y$;
6. Se il criterio di arresto non è soddisfatto vai a 2, altrimenti STOP.

Si noti che al passo 3 si è menzionata una funzione $\tilde{f}$ che può essere considerata al posto della funzione obiettivo. Questo aspetto verrà chiarito nel prossimo paragrafo.
Un approccio di tabu search può comprendere molti altri "accessori", quali le *tabu list di lunghezza variabile* nel corso dell'algoritmo, nonché strategie di *diversificazione* e di *intensificazione* della ricerca. Per questi argomenti si rimanda a testi più specifici.

3.1 Scelte realizzative in un algoritmo di tabu search

La tabu search è applicabile, in linea di principio, pressoché a qualsiasi problema di ottimizzazione combinatoria. Tuttavia, alla scelta progettuale relativa al vicinato (come del resto già nel caso della ricerca locale) va aggiunta ora quella relativa alla definizione della mossa tabu e alla lunghezza della tabu list.

L'esperienza computazionale mostra che l'elemento più importante nell'efficienza di un algoritmo di tabu search è la scelta del vicinato. Una caratteristica importante di un vicinato è la *raggiungibilità* della soluzione ottima. Ci si chiede cioè se, data una qualunque soluzione iniziale, esiste un cammino di soluzioni che, passando da un vicinato all'altro, consente di raggiungere l'ottimo globale. Attenzione: non stiamo qui chiedendo che il nostro algoritmo sia in grado effettivamente di trovarlo (questo dipenderà da come vengono scelte le mosse tabu, dal criterio di arresto etc.), ma solo che con una certa definizione di vicinato questo sia possibile, almeno in teoria. Per capire meglio questa definizione, si consideri il problema della colorazione di grafi. Com'è noto, dato un grafo $G = (N, A)$, non orientato, non pesato, una soluzione ammissibile è una *colorazione*, cioè un assegnamento di colori ai nodi tale che nodi adiacenti hanno colori diversi. In effetti, si tratta di partizionare i nodi del grafo in un numero minimo di insiemi stabili (ciascuno corrispondente a un colore). Supponiamo allora di avere una soluzione iniziale ammissibile che fa uso di k colori. Potremmo definire il vicinato di una soluzione come costituito da tutte quelle colorazioni ottenute cambiando il colore di un nodo alla volta in tutti i modi possibili, in modo tale da mantenere l'ammissibilità della soluzione, e che non usano più di k colori.

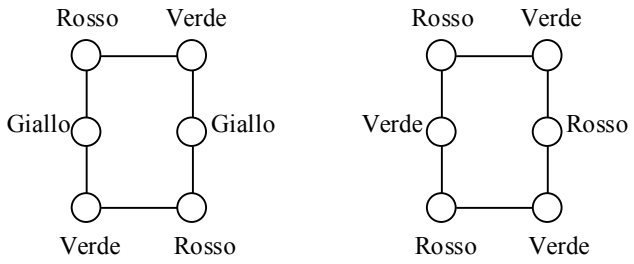

Figure 1: Una 3-colorazione e una 2-colorazione.

È facile vedere che con questa definizione la soluzione ottima può non essere raggiungibile. Si consideri infatti il semplice caso in figura 1, e supponiamo di prendere come soluzione iniziale quella a sinistra. Questa soluzione fa uso di 3 colori, mentre la soluzione ottima ne usa solo 2. In base alla definizione data sopra, la soluzione ottima non risulta raggiungibile a partire da quella iniziale; anzi, cambiando il colore di un solo nodo non è possibile nemmeno mantenere l'ammissibilità della soluzione. Questo fatto può indurre a ritenere che questa scelta del vicinato non sia la più conveniente.
Un altro aspetto molto rilevante è la scelta della funzione obiettivo. La scelta più naturale ovviamente è quella di usare nell'algoritmo di tabu search la funzione che si desidera minimizzare. Questa scelta però non necessariamente è quella più conveniente. Si consideri ancora il problema di graph coloring. In questi problemi, tipicamente il valore ottimo è un numero molto basso, e comunque non molto distante dal valore della soluzione iniziale, se questa è prodotta con una qualsiasi delle euristiche rapide presenti in letteratura. La conseguenza è che un numero enorme di soluzioni ammissibili hanno in realtà lo stesso numero di colori, e dunque la funzione obiettivo assai raramente avrà dei miglioramenti nel corso dell'algoritmo. La conseguenza è che l'algoritmo seguirà un cammino abbastanza casuale nella regione ammissibile, dal momento che un numero elevatissimo di direzioni di ricerca apparirebbero altrettanto appetibili. Si può allora utilizzare una funzione $\tilde{f}$ legata ovviamente a quella originaria, ma che consente di ovviare a questo problema. Torniamo al problema di colorazione di grafi, e supponiamo di rilassare il vincolo che due nodi adiacenti debbano sempre avere colori diversi. Possiamo allora sostituire la funzione obiettivo con una $\tilde{f}$ equivalente ma più *maneggevole*. Ad esempio: se noi consentiamo a due nodi adiacenti di avere lo stesso colore, il problema diviene quello di minimizzare il numero di archi che hanno ambedue gli estremi con lo stesso colore (archi *monocromatici*), sempre con il vincolo di non usare più di k colori (altrimenti banalmente assegneremmo a tutti i nodi colori diversi). Una soluzione con valore della $\tilde{f}$ pari a 0 rappresenta evidentemente una k-colorazione ammissibile. A questo punto, data una soluzione x (che ora è semplicemente un assegnamento di colori ai nodi, non una colorazione), possiamo definire il vicinato $N(x)$ nel seguente modo: $N(x)$ è costituito da tutte le soluzioni ottenute cambiando il colore di un nodo v, estremo di un arco monocromatico. Una mossa sarà quindi individuata dalla coppia (v, r), ove r è il colore assunto dal nodo v. Tale coppia di valori, che individua univocamente la mossa, entrerà nella lista tabu. Ossia, per $|T|$ mosse, sarà proibito far assumere al nodo v il colore r. Questo tipo di funzione obiettivo e di vicinato ha consentito empiricamente di ottenere risultati molto buoni per un problema particolarmente difficile e importante come quello della colorazione di grafi.

3.2 Tabu search: conclusioni

Concludiamo dicendo che la tabu search rappresenta forse più un approccio di tipo ingegneristico a problemi di ottimizzazione di grandi dimensioni piuttosto che un approccio matematicamente elegante. La tabu search è oggi da molti considerata la metaeuristica che consente di avere il miglior rapporto tra qualità delle soluzioni e sforzo computazionale (includendo in questo anche la complessità implementativa). I risultati ottenuti in certi campi, come ad esempio per i problemi di *job shop*, sono abbastanza evidenti.

4 Simulated Annealing

Il Simulated Annealing è una tecnica stocastica che si ispira alla meccanica statistica per trovare soluzioni per problemi di ottimizzazione sia continua che discreta. Ricercare il minimo globale di una funzione di costo con molti gradi di libertà è un problema molto complesso, se questa funzione ammette un gran numero di minimi locali. Uno degli obiettivi principali dell'ottimizzazione è proprio quello di evitare di rimanere intrappolati in un minimo locale. Questo è uno dei limiti più grandi delle tecniche di ricerca locale. I metodi basati sul Simulated Annealing applicano un meccanismo probabilistico che consente alla procedura di ricerca di fuggire da questi minimi locali. L'idea è quella di accettare, in certi casi, oltre alle transizioni che corrispondono a miglioramenti nella funzione obiettivo, anche quelle transizioni che portano a peggioramenti nel valore di questa funzione di valutazione. La *probabilità* di accettare tali deterioramenti varia nel corso del processo di ricerca, e discende lentamente verso zero. Verso la fine della ricerca, quando vengono accettati solo miglioramenti, questo metodo diventa una semplice ricerca locale. Tuttavia, la possibilità di transire in punti dello spazio di ricerca che deteriorano la soluzione ottima corrente, consente di abbandonare i minimi locali ed esplorare meglio l'insieme delle soluzioni ammissibili.

La strategia che è alla base del Simulated Annealing si ispira al processo fisico di annealing (raffreddamento). Se il sistema si trova all'equilibrio termico ad una data temperatura T, allora la probabilità $n_T(s)$ che esso sia in una data configurazione s dipende dall'energia dello stato in questione $E(s)$, e segue la distribuzione di Boltzmann:

$$n_T(s) = \frac{e^{-E(s)/kT}}{\sum_w e^{-E(w)/kT}}$$

dove k è la costante di Boltzmann e la sommatoria si estende a tutti gli stati possibili w. Fu per primo Metropolis nel 1953 a proporre un metodo per calcolare la distribuzione di un sistema di particelle all'equilibrio termico usando un metodo di simulazione al computer. In questo metodo, supposto che il sistema si trovi in una configurazione q avente energia

$E(q)$, si genera un nuovo stato r, avente energia $E(r)$, spostando una delle particelle dalla sua posizione: la nuova configurazione viene quindi confrontata con la vecchia. Se $E(r) \leq E(q)$ il nuovo stato viene accettato; se $E(r) > E(q)$ esso non viene respinto, bensi' viene accettato con una probabilità pari a:

$$e^{-(E(r)-E(q))/kT}$$

Secondo questo metodo, esiste dunque una probabilità non nulla di raggiungere stati di più alta energia, ossia di scavalcare le barriere di energia che separano i minimi globali dai minimi locali. Si noti che la funzione esponenziale esprime il rapporto tra la probabilità di trovarsi nella configurazione r e la probabilità di trovarsi in q. Kirkpatrick utilizzò lo schema del Simulated Annealing per problemi di ottimizzazione combinatoria. Per fare ciò, sostitui' l'energia con una funzione costo e gli stati di un sistema fisico di particelle con le soluzioni di un problema di minimizzazione. La perturbazione delle particelle con le soluzioni diventa allora equivalente ad uno spostamento nello spazio di ricerca verso un punto che si trova nelle vicinanze della soluzione corrente. La minimizzazione è raggiunta *riscaldando* dapprima il sistema, e poi raffreddandolo lentamente fino al raggiungimento della situazione di cristallizzazione in uno stato stabile. È da sottolineare il fatto che per raggiungere configurazioni di bassa energia non è sufficiente abbassare semplicemente la temperatura; è necessario, viceversa, attendere per ciascuna temperatura di transizione, un tempo sufficiente a raggiungere l'equilibrio termico del sistema. Altrimenti, la probabilità di raggiungere stati di bassa energia è sensibilmente ridotta.

4.1 Schema di principio del Simulated Annealing

La simulazione del processo di annealing applicato a problemi di ottimizzazione richiede diversi passi preparatori. Per prima cosa si devono identificare nel problema di ottimizzazione le analogie con i concetti fisici: l'energia diventa la funzione di costo, le configurazioni di particelle divengono le configurazioni di parametri (variabili decisionali) del problema da ottimizzare, ricercare uno stato di minima energia significa ricercare una soluzione che minimizza la funzione di costo, la temperatura diventa un parametro di controllo. Quindi, si deve scegliere un opportuno *schema di annealing* consistente nella regolazione dei parametri da cui dipende il processo di ottimizzazione: si tratta cioè di stabilire la legge di decadimento della temperatura e la durata del tempo necessario per il raggiungimento dell'equilibrio termico a ciascuna temperatura. Infine si deve introdurre un metodo di perturbazione del sistema che consenta di esplorare lo spazio di ricerca generando nuove configurazioni. L'algoritmo di annealing descritto da Kirkpatrick consiste nella esecuzione ripetuta dell'algoritmo di Metropolis per ciascuna temperatura di

transizione, fino al raggiungimento della configurazione ottima del sistema, ottenuta in prossimità della temperatura di cristallizzazione prefissata. L'algoritmo di Metropolis, che accetta configurazioni che incrementano il costo, fornisce il meccanismo probabilistico che consente di evitare l'intrappolamento nei minimi locali. Lo schema di principio di un algoritmo basato sul Simulated Annealing è riportato di seguito.

```
Algoritmo di Simulated Annealing
Inizializzazione di C ed X
While Not (condizione di uscita)
    Do Begin
    for i:=1 to L
        Do Begin
        Generazione di Y da X
        IF f(Y) ≤ f(X)
        Then X:=Y
        Else if e^{-(f(Y)-f(X))/C} > random(0,1)
            Then X:=Y
        End
    Riduzione del parametro C
End
```

L'algoritmo parte con l'inizializzazione di X, che rappresenta una possibile soluzione, e di C che costituisce il parametro di controllo dell'algoritmo stesso (equivale alla temperatura nel processo di annealing). Quindi si entra nel ciclo while da cui si esce solo quando risulterà vera la condizione di arresto: questa situazione si verifica quando si raggiunge una soluzione soddisfacente o in corrispondenza del raggiungimento della temperatura di congelamento prefissata per il sistema. Le fasi del ciclo while sono essenzialmente due; nella prima si entra in un ciclo for, durante la seconda si provvede a diminuire la temperatura del sistema secondo una legge prefissata che definisce lo schema di annealing. La prima fase corrisponde all'esecuzione dell'algoritmo di Metropolis e viene ripetuta L volte (questo è un altro parametro essenziale dello schema di annealing) per il raggiungimento dell'equilibrio termico; la seconda fase determina la transizione da una temperatura ad un'altra. Per quanto riguarda il problema della definizione dello schema di annealing, ovvero della definizione dei parametri di funzionamento, esistono in merito diverse soluzioni e una gran quantità di studi sono stati dedicati a questo problema.

5 Algoritmi genetici

Con il termine *weak methods* si indicano quei metodi di risoluzione di problemi che si basano su poche assunzioni (o conoscenze) relative alle particolari strutture e caratteristiche dei problemi stessi, motivo per il quale tali metodi sono applicabili ad una vasta classe di problemi. Gli Algoritmi Genetici rientrano pienamente in questa categoria, dal momento che essi sono in grado di compiere una efficiente ricerca anche quando tutta la conoscenza a priori è limitata alla sola procedura di valutazione che misura la qualità di ogni punto dello spazio di ricerca (misura data, ad esempio, dal valore della funzione obiettivo). Questa caratteristica conferisce a tali algoritmi una grande robustezza, ovvero una grande versatilità che li rende applicabili a diversi problemi, al contrario dei metodi convenzionali che, in genere, non trovano altra applicazione che quella relativa al problema per cui sono stati ideati.
Gli Algoritmi Genetici integrano l'abilità di semplici rappresentazioni (ad esempio stringhe di bit) a codificare strutture molto complicate con la potenza esibita da semplici trasformazioni (che agiscono su tali rappresentazioni) nel migliorare queste strutture. Questo miglioramento è l'obiettivo primario dell'ottimizzazione. Infatti, sotto l'azione di certe trasformazioni genetiche, si registrano miglioramenti, o, se si vuole, evoluzioni delle rappresentazioni in modo da imitare il processo di evoluzione di popolazioni di organismi viventi. In natura, il problema che ciascuna specie deve affrontare è quello della sopravvivenza, che coincide con il trovare un adattamento vantaggioso ad un ambiente mutevole e talvolta ostile. Le trasformazioni genetiche che consentono una evoluzione in tal senso sono quelle che alterano il patrimonio genetico della specie contenuto nei cromosomi di ciascun individuo: esse sono il *crossover* di gruppi di geni parentali e la *mutazione*. I cromosomi costituiscono l'equivalente delle stringhe di bit, in quanto codificano strutture biologiche molto complicate, mentre le trasformazioni genetiche costituiscono il mezzo attraverso il quale si generano nuove strutture biologiche più evolute delle precedenti.
L'idea di fondo degli algoritmi genetici è la seguente. Supponiamo di avere un certo insieme di soluzioni di un problema di ottimizzazione. Tra queste, ce ne saranno di più buone e di meno buone. La qualità di una soluzione è misurata da una funzione di merito, detta *fitness function*, che in genere coinciderà con la funzione obiettivo, ma non sempre (un po' come la $\tilde{f}$ nella tabu search). A questo punto, vogliamo generare nuove soluzioni, con la speranza ovviamente che fra queste ve ne siano sempre di migliori. L'idea è allora quella di "accoppiare" le soluzioni tra di loro in modo che "diano alla luce" nuove soluzioni. Allora, da una certa popolazione di individui (soluzioni), se ne ricava un'altra, che costituisce una nuova *generazione*, ossia quella dei figli della popolazione di partenza. Questa nuova generazione potrebbe essere molto più numerosa della precedente, e allora

quello che si fa è effettuare una *selezione*, cioè si escludono dalla popolazione tutte le soluzioni che hanno un valore di fitness function inferiore a una certa soglia. Con la popolazione cosi' selezionata, si ricomincia, generando quindi i nipoti delle soluzioni di partenza e cosi' via per un numero prefissato di generazioni.
Come si vede, in questo contesto cosi' generale vi sono diverse scelte da fare. Anzitutto, negli algoritmi genetici gioca un ruolo fondamentale il modo in cui vengono *rappresentate* le soluzioni ammissibili. Tipicamente, una soluzione sarà rappresentata da un insieme di stringhe di interi o binarie, che sono i *cromosomi*. Ciascun cromosoma a sua volta è composto da *geni* (i singoli bit). Dalla combinazione di due individui (*crossover*) nascerà una nuova soluzione del problema. I geni nei cromosomi del nuovo individuo dovranno – ovviamente – provenire da quelli dei due genitori. Un semplice modo di realizzare questo è il seguente. Se indichiamo con x_i e y_i l'i-esimo gene di un certo cromosoma dei due genitori rispettivamente, il corrispondente gene nel cromosoma-figlio può essere posto uguale all'uno o all'altro con probabilità p e $1-p$ rispettivamente. Si noti che in questo modo si introduce un elemento probabilistico nell'algoritmo. Peraltro, reiterando più volte il processo aleatorio di crossover, diverse soluzioni possono essere ottenute dalla stessa coppia di genitori.
È evidente che la scelta dell'operatore di crossover è particolarmente importante; tuttavia non c'e' solo questa. Un'altra scelta molto importante è quella relativa alla dimensione della popolazione: a ogni generazione, infatti, verranno escluse dall'evoluzione dell'algoritmo tutte le soluzioni con basso valore di funzione obiettivo. È una scelta comune quella di mantenere costante nei vari passi il numero di individui della popolazione.
È implicito in quanto detto finora che l'efficacia degli algoritmi genetici risiede nel fatto empirico che se si combinano soluzioni buone, si ottengono soluzioni ancora buone. Tuttavia, sperimentalmente si è potuto osservare che anche gli algoritmi genetici, dopo un certo numero di iterazioni, tendono a produrre soluzioni che non migliorano più. Per diversificare la ricerca, allora, si introduce un nuovo operatore, detto *mutazione*. Prima cioè di procedere alla successiva generazione, in alcuni degli individui della popolazione viene alterato un gene. Ovviamente, gli operatori di mutazione dipendono molto dal particolare problema in esame.

5.1 Un esempio: multiprocessor scheduling

A titolo di esempio, si consideri il problema NP-completo che va sotto il nome di *multiprocessor scheduling*. Questo problema consiste nell'assegnare n oggetti di peso $w_1, w_2, \ldots, w_n$ a m contenitori, con l'obiettivo di minimizzare il peso del contenitore più carico (per $m = 2$ questo problema coincide con PARTITION). In questo caso possiamo scegliere di rapp-

resentare la generica soluzione (assegnamento di oggetti a contenitori) per mezzo di una stringa (un unico cromosoma) di n interi in $\{1, \ldots, m\}$, che indicano il contenitore cui è assegnato ciascun oggetto. Cosi' ad esempio si consideri un'istanza costituita da 8 oggetti, di peso 2, 3, 4, 6, 7, 9, 10, 14. Consideriamo le due soluzioni ammissibili: $x = (1\ 3\ 3\ 2\ 3\ 3\ 2\ 1)$ $y = (3\ 2\ 1\ 3\ 2\ 1\ 2\ 3)$ di valore 23 e 22 di funzione obiettivo rispettivamente. Possiamo dunque combinarle nel modo indicato in precedenza per ottenere nuove soluzioni, ciascun gene delle quali coinciderà con quello di x o di y con probabilità p e $1-p$ rispettivamente. (In particolare, sarà identico a quello di ambedue i genitori qualora esso sia uguale nei due cromosomi, come il penultimo gene in questo esempio). Generando allora un certo numero di figli da x e y, tra di essi vi potrebbe essere: $z = (3\ 2\ 1\ 2\ 3\ 3\ 2\ 1)$ che è la soluzione ottima del problema, di valore pari a 19.
Si noti che in questo caso il meccanismo dell'algoritmo è particolarmente semplice. Infatti, la rappresentazione delle soluzioni è tale che scegliendo in qualunque modo il valore di un gene tra quelli dei due genitori, si ottiene sempre una soluzione ammissibile. Questo non è sempre vero, almeno con rappresentazioni semplici delle soluzioni ammissibili. Infine, è evidente che in questo caso una mutazione può consistere semplicemente nell'assegnare un oggetto a un contenitore diverso.

5.2 Schema di principio per un Algoritmo Genetico

Da quanto detto, un algoritmo genetico è fondamentalmente caratterizzato da cinque componenti:

1. una rappresentazione delle soluzioni generalmente sotto forma di stringhe di bit;
2. un criterio per generare la popolazione iniziale di soluzioni;
3. una funzione di valutazione che misura la qualità di ciascuna soluzione;
4. un insieme di operatori genetici che trasformano punti-soluzione in altri punti;
5. un insieme di valori per i parametri di controllo dell'algoritmo.

Lo schema di principio di un algoritmo genetico è quello che segue:

```
Algoritmo Genetico
    Generazione della popolazione iniziale
    While not (condizione di uscita)
        Do begin
        Selezione dei genitori
```

Generazione dei figli
Ricambio generazionale
End

L'algoritmo inizia con la creazione di una popolazione iniziale di soluzioni, generata di solito in modo casuale. Quindi prosegue con un ciclo la cui condizione d'uscita è determinata o dal raggiungimento di una soluzione soddisfacente, vale a dire caratterizzata da un valore di fitness giudicato buono, oppure dal raggiungimento di un numero prefissato di generazioni da elaborare.

Il ciclo che realizza la generica iterazione di un algoritmo genetico determina il passaggio da una generazione ad un'altra. Esso inizia con la selezione dei genitori dalla popolazione corrente, ovvero degli individui che partecipano alla fase di riproduzione. Una possibile implementazione del test di sopravvivenza consiste nell'assegnare a ciascun esemplare una probabilità di figurare nella lista dei genitori proporzionale al proprio valore di fitness. In questo modo, si seleziona il patrimonio genetico migliore di tutta la popolazione perché esso venga trasmesso alle generazioni successive.

Alla fase di selezione dei genitori segue quella di riproduzione, mediante la quale avviene la generazione di nuovi punti dello spazio di ricerca: ciascuna coppia di individui prescelti genera, con probabilità p_c (probabilità di crossover), una coppia di figli. Il processo di riproduzione simula il meccanismo biologico di crossover in cui si verifica uno scambio di gruppi di geni appartenenti ai due genitori. Nel caso più semplice viene selezionato casualmente un punto di rottura del cromosoma e quindi, in corrispondenza di questo taglio, si ha uno scambio delle parti omologhe. Tuttavia si possono adottare tecniche più sofisticate. L'altro operatore genetico responsabile di variazioni nei parametri codificati dal cromosoma è la mutazione, che consiste nell'effettuare, con una probabilità p_m, una complementazione del valore assunto da un bit in corrispondenza ad una posizione scelta in modo casuale. Questo operatore interviene sui nuovi nati.
La parte finale del ciclo comporta la fase di ricambio generazionale durante la quale si deve formare la popolazione da trasmettere alla generazione successiva, operando una selezione fra gli individui di cui si dispone (genitori e figli). Questa fase è necessaria per limitare la dimensione della popolazione di soluzioni che, altrimenti, crescerebbe esponenzialmente con l'evolvere dell'algoritmo. Esistono in merito diverse soluzioni. La prima costituisce il metodo di ricambio generazionale classico in cui i genitori lasciano il posto ai nuovi nati: alla popolazione dei genitori si sostituisce la popolazione dei figli. La seconda si basa ancora sul ricambio classico ma con l'accorgimento di preservare dalla scomparsa l'esemplare migliore in assoluto, qualora la nuova popolazione non dovesse contenere nes-

suna soluzione migliore di esso. La terza soluzione prevede una selezione naturale tra genitori e figli in base alla qualità di ciascuno di essi, ossia sopravvivono solo gli esemplari migliori.

5.3 Analisi del funzionamento di un Algoritmo Genetico: schemi e parallelismo implicito

Le prime interpretazioni teoriche sul funzionamento degli algoritmi genetici sono state proposte da John Holland attraverso l'introduzione del concetto di *schema*. Secondo la definizione di Holland, uno schema è una maschera di similarità che descrive un sottoinsieme di stringhe che possiedono similarità in corrispondenza di certe posizioni della stringa. Uno schema viene definito su un alfabeto di tre simboli $(0, 1, *)$, in cui il simbolo $*$ denota un don't care. Si consideri, per esempio, lo schema $H = 1 * 10 * *11$: una stringa costituisce un campione dello schema H se tutti i bit definiti nello schema trovano corrispondenza nella stringa; le posizioni in cui vi è un asterisco non sono significative per lo schema, perciò in quelle posizioni può esservi indifferentemente un simbolo 1 oppure un simbolo 0. Quindi le stringhe 1010111 e 1110001 appartengono allo schema H, non cosi' 1100111.
Secondo questa definizione, per alfabeti di cardinalità k ci sono $(k+1)^l$ schemi, essendo l la lunghezza del cromosoma, ovvero il numero di bit della stringa: nel caso di alfabeti binari si hanno, dunque, 3^l schemi. Ciascuna soluzione rappresenta contemporaneamente un numero grandissimo di schemi diversi: ad esempio, la stringa 1110111 costituisce un esemplare dello schema $H_1 = 11 * * * * *$, dello schema $H_2 = * * * * *11$ e di tantissimi altri ancora. Pur trattando un numero limitato di soluzioni, gli algoritmi genetici elaborano un numero grandissimo di schemi, precisamente tutti quelli rappresentati da ciascuna stringa della popolazione (D.E. Goldberg ha dimostrato che l'ordine di grandezza degli schemi elaborati è proporzionale al cubo della dimensione della popolazione). In ciò risiede quella caratteristica degli algoritmi genetici che Holland ha chiamato *parallelismo implicito*.
Si indica con *ordine* dello schema $o(H)$ il numero di bit definiti dallo schema (ad esempio $o(1 * 10 * *1) = 4$) e con *lunghezza di definizione* $\delta(H)$ la distanza tra i due bit estremi che sono fissati nello schema, ossia, che differiscono da "*" (ad esempio $\delta(*11 * 0 * *) = 4$).
Dopo queste premesse si può illustrare la cosiddetta legge degli schemi, che costituisce la base teorica del funzionamento degli algoritmi genetici.
Indichiamo con $m(H, T)$ il numero di stringhe della popolazione che, alla generazione $T-esima$, contengono lo schema H; con $f(H)$ la media dei valori di fitness delle stringhe che contengono H e con f_t la media dei valori di fitness dell'intera popolazione. Nel passaggio da una generazione alla successiva, considerando il solo processo di selezione naturale

(in cui si adotta un criterio di scelta che assegna probabilità di selezione proporzionali ai valori di fitness delle stringhe), il numero di stringhe in questione è una funzione di $m(H,T)$ e del rapporto $\frac{f(H)}{f_t}$:

$$m(H,T+1) = m(H,T)\frac{f(H)}{f_t}$$

.

Volendo considerare il contributo del processo di crossover, bisogna calcolare la probabilità che lo schema H venga distrutto dal processo di accoppiamento. Essa coincide con la probabilità che il punto di crossover cada all'interno della sottostringa delimitata dai bit estremi che definiscono lo schema, moltiplicata per la probabilità di crossover p_C (che è un parametro predefinito dell'algoritmo). Ad esempio nei seguenti casi:

a) $**1**\,|\,1*$

b) $**\,|\,1**1*$

il crossover risulta potenzialmente distruttivo per lo schema H nel caso a), mentre non lo è nel caso b). La probabilità di distruzione $\chi(H)$ per lo schema H è data dalla somma delle probabilità degli eventi distruttivi, ovvero dalla somma delle probabilità che il punto di crossover cada in corrispondenza di ciascuno dei bit interni allo schema, vale a dire:

$$\chi(H) = p_C\frac{\delta(H)}{l-1}$$

in cui $\delta(H)$ è la lunghezza di definizione dello schema ed $l-1$ sono i possibili punti di crossover.

Per la probabilità di sopravvivenza $\alpha(H)$ di uno schema H avremo quindi $\alpha(H) \geq 1 - \chi(H)$.

Infatti, occorre considerare l'esistenza di casi in cui il crossover, pur essendo potenzialmente distruttivo, non produce la scomparsa dello schema in questione. Si verifica immediatamente che ciò avviene, per esempio, quando si accoppiano due esemplari dello stesso schema H, oppure quando, dall'accoppiamento di un esemplare di H con uno che non appartiene ad H, si produce ancora un rappresentante di H. Per esempio, dato lo schema $H = *1***1*$, accoppiando le soluzioni 0100010 e 1100000, si ottengono (se il punto di rottura cade dopo il terzo bit) i discendenti 0100000 e 1100010, entrambi rappresentanti di H: come può vedersi il crossover in questo caso non è stato distruttivo.

Inoltre, un nuovo rappresentante di H può essere generato dall'accoppiamento di due individui non appartenenti ad H: con a definizione di H data sopra, le soluzioni 0100000

e 0000011 producono (se il punto di rottura cade dopo il terzo bit) le stringhe 0100011 e 0000000, in cui la prima stringa è un rappresentante di H.
La probabilità di sopravvivenza di un generico schema H dipende anche dall'operatore di mutazione, il quale interviene su ogni bit di ciascuna stringa con probabilità p_m. Tale probabilità di sopravvivenza coincide con la probabilità che non si abbia mutazione (pari a $1-p_m$) su ciascuno dei bit che definiscono lo schema (il numero di questi bit è pari a $o(H)$). Complessivamente la probabilità di sopravvivenza alla mutazione risulta $(1-p_m)^{o(H)}$. Questa probabilità, se $p_m \ll 1$ (come usuale) può essere approssimata con lo sviluppo in serie arrestato al primo ordine: $1 - p_m o(H)$.
La probabilità di sopravvivenza dello schema H è data dall'intersezione degli eventi descritti, sopravvivenza alla fase di selezione, al crossover e alla mutazione, pertanto si giunge alla relazione finale:

$$m(H, T+1) \geq m(H, T)\frac{f(H)}{f_t}\left(1 - p_C\frac{\delta(H)}{(l-1)} - o(H)p_m + p_C p_m o(H)\frac{\delta(H)}{(l-1)}\right)$$

dove si può trascurare il termine in $p_C p_m$ ottenendo:

$$m(H, T+1) \geq m(H, T)\frac{f(H)}{f_t}\left(1 - p_C\frac{\delta(H)}{(l-1)} - o(H)p_m\right)$$

È interessante stabilire quali schemi evolvono con un incremento nel numero di rappresentanti. Gli schemi in questione sono quelli caratterizzati da valori di fitness sopra la media, da piccoli valori di $\delta(H)$ (rispetto alla lunghezza del cromosoma) e di o(H): ad essi Holland ha dato il nome di *building blocks*.
Questi building blocks vengono iterativamente campionati e ricombinati dagli algoritmi genetici per formare stringhe migliori.
Dalla relazione ottenuta si nota che i parametri p_C e p_m regolano l'ampiezza della crescita esponenziale. Valori bassi per la probabilità di crossover assicurano un grande incremento per gli schemi che presentano valori di fitness sopra la media. Questa strategia sfrutta al limite la conoscenza di cui si dispone al momento dell'elaborazione di una nuova generazione, sacrificando l'esplorazione dello spazio di ricerca. Valori di p_C molto grandi, ovvero prossimi ad 1, riducono il fattore esponenziale a valori bassi; in questo caso, l'incremento del numero di esemplari di building blocks, ovvero l'utilizzazione della conoscenza disponibile, è limitato, a vantaggio di una robusta esplorazione dello spazio di ricerca. Per p_C tendente ad 1, si implementa una strategia di ricerca che assomiglia molto a quella random. Occorre scegliere opportunamente il valore di p_C per assicurare un buon compromesso fra le esigenze contrastanti cui si è accennato, vale a dire l'esplorazione dello spazio di ricerca e l'utilizzazione della conoscenza acquisita. Diversi studi e ricerche sono stati dedicati alla messa a punto dei parametri caratteristici degli algoritmi genetici, per quanto riguarda

p_C vengono suggeriti valori decrescenti con l'aumentare della popolazione. Intuitivamente ciò può spiegarsi con l'importanza del crossover nella prevenzione della convergenza verso minimi locali soprattutto nel caso di popolazioni piccole.
Infine, qualche chiarimento merita ancora l'operatore di mutazione che interviene nel processo di generazione di nuove popolazioni.
Durante l'evoluzione di una popolazione può verificarsi a causa degli effetti non deterministici della selezione naturale, un effetto detto di *assorbimento*, consistente nella perdita irreparabile di materiale genetico, ossia nella scomparsa di un determinato carattere della specie. In altre parole, in popolazioni di piccole dimensioni, dopo un certo numero di generazioni può verificarsi l'evento che tutti gli individui hanno lo stesso valore in corrispondenza alla stessa posizione nel cromosoma. La mutazione costituisce il rimedio contro questa irreparabile scomparsa, consentendo il recupero del carattere attraverso una casuale modifica dei geni. Si osservi che per questo problema il crossover non è d'aiuto. La mutazione consente quindi l'esplorazione di regioni dello spazio di ricerca che, altrimenti, sarebbero trascurate dall'algoritmo.
Per quanto riguarda la scelta di p_m, appare ragionevole dimensionare questo parametro secondo la probabilità di occorrenza del fenomeno dell'assorbimento; poiché la frequenza con cui si verifica questo fenomeno è tanto più alta quanto più piccola è la popolazione, allora la probabilità di mutazione può essere utilmente legata al reciproco della dimensione della popolazione.

6 Conclusioni

Gli approcci meta-euristici vanno intesi in un senso molto più flessibile di quanto visto qui. Molte scelte progettuali possono essere ridiscusse a seconda del particolare problema in esame, dando vita eventualmente ad approcci "ibridi" ma possibilmente più efficaci. Cosi' ad esempio, in un algoritmo genetico può essere raccomandabile, dopo aver creato ciascun nuovo individuo, effettuare una ricerca locale per migliorare la qualità delle singole soluzioni; oppure, in un algoritmo di tabu search, accettare o meno uno spostamento in una soluzione non migliorativa in base a un criterio probabilistico anziche' deterministico, avvicinando cosi' la tabu search al simulated annealing.
Il successo che tali algoritmi hanno avuto e continuano ad avere fa ritenere che il loro ruolo, nella soluzione di problemi di ottimizzazione combinatoria di grandi dimensioni, è destinato a essere ancora di primo piano.