Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Algoritmi evolutivi e programmazione genetica: strategie di progettazione e parallelizzazione

Gianluigi Folino

RT-ICAR-CS-03-17	Dicembre 2003

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Algoritmi evolutivi e programmazione genetica: strategie di progettazione e parallelizzazione

Gianluigi Folino[1]

Rapporto Tecnico N.:	*Data:*
RT-ICAR-CS-03-17	**Dicembre 2003**

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Cosenza , Via P. Bucci 41C, 87036 Rende(CS)

Abstract

Evolutionary algorithms are heuristics that mimics the processes of natural evolution in order to solve global search problems. Therefore, they can solve problems even with a few knowledge of the dominium. However, their diffusion is quite limited. Indeed, they usually need a tedious and expensive tuning phase and large populations to obtain a good convergence for hard problems. In order to improve the performance of evolutionary algorithms a number of strategies of design and techniques of parallelization can be used. This work surveys evolutionary algorithms, discussing the strategy of design to improve the tuning phase and the techniques of parallelization to reduce the execution time. Major regard is given to genetic programming, since it is relatively recent and not deeply discussed in literature.

Gli algoritmi evolutivi sono strategie euristiche che imitano i processi di evoluzione naturale per risolvere problemi di ricerca globale e si caratterizzano per la capacità di risolvere problemi anche con scarsa conoscenza del dominio. Tuttavia, la loro diffusione è limitata dalle difficoltà di progettazione dovute all'elevato numero di parametri da settare e dalla necessità di utilizzare una dimensione di popolazione elevata per ottenere una buona convergenza dell'euristica in problemi di una certa difficoltà. Ne conseguono tempi elevati sia in fase di tuning dei parametri che durante l'esecuzione dell'algoritmo. Per migliorare le prestazioni degli algoritmi evolutivi una serie di strategie di progetto e varie tecniche di parallelizzazione possono essere usate. Questo lavoro presenta una rassegna degli algoritmi evolutivi discutendone, in particolare, le strategie di progettazione per abbreviare la fase di tuning e le tecniche di parallelizzazione mirate alla riduzione dei tempi di esecuzione. Particolare rilievo è dato alla programmazione genetica, in quanto relativamente recente e ancora non molto trattata in letteratura.

1. Introduzione

Molti dei problemi fondamentali trattati dall'Intelligenza Artificiale possono essere ricondotti a quello di trovare un massimo o un minimo globale in uno spazio limitato di ricerca, considerando eventualmente alcuni vincoli sullo spazio delle soluzioni ammissibili. Le tecniche esatte di risoluzione non riescono spesso a trovare una soluzione in tempi accettabili; perciò si ricorre a metodi di tipo euristico.

Gli *algoritmi evolutivi* sono strategie euristiche che si ispirano all'evoluzione naturale, teorizzata da Darwin nel suo libro sull'evoluzione della specie, per risolvere problemi di ricerca globale. Questo tipo di algoritmi si basa sul principio darwiniano che gli elementi più “adatti” all’ambiente hanno maggiore possibilità di sopravvivere e di trasmettere le loro caratteristiche ai successori; in pratica, si ha una popolazione di individui che evolvono di generazione in generazione attraverso meccanismi simili alla riproduzione sessuale e alla mutazione dei geni. Questo meccanismo conduce ad una ricerca euristica che privilegia le zone dello spazio di ricerca dove maggiormente è possibile trovare soluzioni migliori, non trascurando altre zone a più bassa probabilità di successo in cui saranno impiegate un minor numero di risorse. Gli algoritmi evolutivi sono classificati fra i metodi di ricerca "deboli", così denominati perché si adattano a risolvere una grande varietà di problemi, incorporando poca conoscenza del domino particolare, in contrapposizione a quelli "forti" che sfruttano le conoscenze del dominio applicativo. Si è visto però che gli algoritmi evolutivi tendono a sviluppare un'intelligenza emergente che li porta a risolvere in modo efficace anche problemi con domini particolari.

All'interno degli algoritmi evolutivi si è soliti distinguere fra algoritmi genetici, programmazione genetica, strategie evolutive e programmazione evolutiva; in realtà soprattutto le prime due tipologie hanno avuto larga diffusione nella comunità scientifica. Un *algoritmo genetico* è costituito da una popolazione finita di individui di dimensione M, da una funzione di adattamento, detta fitness, che fornisce una misura della capacità dell’individuo (solitamente codificato come una stringa di bit) di adattarsi all’ambiente, cioè costituisce una stima della bontà della soluzione e un’indicazione sugli individui più adatti a riprodursi, da una serie di operatori (crossover, mutazione)

che trasformano l'attuale popolazione nella successiva e da un criterio di terminazione, che stabilisce quando l'algoritmo si deve fermare.

La *programmazione genetica*, introdotta da John R. Koza, è una tecnica di risoluzione di problemi per mezzo della quale programmi per computer si evolvono combinandosi, riproducendosi o mutando per dar luogo ad altri programmi che meglio si adattano a risolvere un determinato problema. Essa rappresenta un'estensione degli algoritmi genetici dove ogni elemento della popolazione rappresenta un programma per computer anziché una stringa di bit. Un programma è rappresentato come un albero in cui i nodi interni sono funzioni e le foglie costituiscono i simboli terminali del programma. Lo spazio di ricerca è costituito da tutti i programmi composti dai terminali e dalle funzioni definite per uno specifico problema. La popolazione dei programmi evolve utilizzando una funzione di *fitness*, che definisce la bontà di un programma a risolvere un determinato problema, e gli operatori genetici di crossover e mutazione adattati alla rappresentazione ad albero.

La programmazione genetica e gli algoritmi evolutivi in genere hanno bisogno di grandi popolazioni per ottenere una buona convergenza. Questo aspetto, unito al fatto che calcolare la fitness di un singolo individuo richiede la valutazione su un certo training set, porta ad uno sforzo computazionale notevole. Ancora, nel caso particolare della programmazione genetica l'albero del programma, può anche essere molto complesso; in questo caso il problema non riguarda solo la dimensione temporale, ma anche quella spaziale che può superare la capacità di memorizzazione di una singola macchina. Da qui la necessità di realizzare un'implementazione parallela di questo tipo di algoritmi su ambienti di tipo distribuito, per ovviare anche al problema della memoria insufficiente. Questo compito risulta facilitato dal fatto che la programmazione genetica è implicitamente parallela, in quanto la valutazione della fitness di ogni individuo, che è poi la fase più dispendiosa dell'algoritmo, è indipendente e può avvenire in parallelo. La fase problematica, invece, è la selezione degli individui nella popolazione che ha bisogno, nell'ipotesi di una distribuzione della popolazione, di accessi remoti che riducono la possibilità di avere speed-up lineari dell'algoritmo parallelo.

I metodi per parallelizzare un algoritmo evolutivo possono essere classificati in tre tipologie. Ricordiamo il *modello globale*, in cui si ha sempre una singola popolazione, come nella versione sequenziale, ma la valutazione della fitness degli individui è eseguita in parallelo, dividendo il carico fra i processori disponibili, mentre il crossover

e gli altri operatori sono applicati in sequenziale. La selezione è quella classica che agisce su tutta la popolazione, da qui il nome globale per il modello. Questo tipo di modello ha il vantaggio di preservare il comportamento dell'algoritmo sequenziale anche in parallelo, ma la fase di selezione richiede frequenti accessi all'intera popolazione e il conseguente overhead di comunicazione risulta inaccettabile sempre che non ci siano funzioni di fitness particolarmente pesanti da valutare. Un'alternativa al modello precedente è quella di utilizzare un diverso meccanismo di selezione limitato agli individui posti su un unico processore. Classico esempio di questa tecnica è il *modello ad isole*, nel quale la popolazione è suddivisa in sottopopolazioni, dette isole, che sono distribuite, usualmente, una per processore. L'implementazione di tale modello prevede la creazione di sottopopolazioni random all'interno di ogni singolo nodo, quindi i vari operatori genetici e la valutazione della fitness sono eseguiti a livello locale. In tal modo l'algoritmo presenta una scalabilità quasi lineare e risulta di facile implementazione (è sufficiente lanciare un processo dell'algoritmo per ogni nodo). Una variante di tale modello prevede la migrazione di elementi da una sottopopolazione all'altra, allo scopo di introdurre diversità nelle isole e garantire una convergenza migliore. Un aspetto ancora non ben compreso di questo modello riguarda la convergenza rispetto a quello sequenziale classico.

Nel *modello diffusivo*, ogni individuo k è sostituito nella successiva generazione da un nuovo elemento generato applicando l'operatore di crossover o mutazione ad elementi appartenenti al vicinato di k stesso. La selezione ristretta ad un vicinato locale rende l'implementazione parallela molto efficiente, dal momento che le comunicazioni sono limitate solo ai vicini e, di conseguenza, non c'è un overhead eccessivo. Ancora, soluzioni sub-ottime non si diffondono velocemente all'interno dell'intera popolazione, favorendo lo svilupparsi di nicchie in cui si esplorano diverse porzioni dello spazio di ricerca ed è evitata una prematura convergenza. Inoltre, rispetto al modello ad isole, si può notare una maggiore stabilità riguardo ai parametri di progettazione.

2. La ricerca evolutiva: dagli algoritmi alla programmazione genetica

2.1. La ricerca e l'evoluzione

"L'intelligenza artificiale è la capacita di un computer o un dispositivo controllato da computer di eseguire compiti comunemente associati con i più elevati processi intellettuali, caratteristici dell'uomo, quali l'abilità di ragionare, scoprire significati, generalizzare o imparare dalla passata esperienza." [Enciclopædia Britannica Online]. L'intelligenza artificiale si occupa, fra l'altro, di risolvere problemi utilizzando algoritmi che imitano o si ispirano all'intelligenza umana. L'utilizzo di metodologie di AI consente di affrontare problemi non facilmente risolvibili con le tecniche tradizionali.

Molti algoritmi di AI si riducono alla ricerca di un massimo o di un minimo globale in uno spazio finito, considerando alcuni vincoli sullo spazio delle soluzioni. Questo ricade tipicamente nel dominio dell'ottimizzazione; diamo, perciò, alcune definizioni formali della ricerca di un massimo:

> *consideriamo un elemento $X \in D$, dove e D è un particolare dominio, spesso chiamato spazio di ricerca. Se consideriamo D come uno spazio cartesiano, allora la cardinalità di D sarà uguale a n e X sarà un vettore. Data una funzione $f: D \rightarrow R$ detta funzione obbiettivo, allora la ricerca dell'ottimo globale corrisponde a trovare un X^* che massimizza tale funzione, cioè: $X^* \in D$ e $\forall X \in D : f(X) \leq f(X^*)$.*

La ricerca diventa molto difficoltosa se la funzione presenta più punti di massimo locale, vincoli sul dominio D, non linearità, ecc.. Gli algoritmi esatti di ricerca spesso non riescono a risolvere il problema in tempi accettabili, perciò sono utilizzati talvolta algoritmi di tipo euristico. Questi ultimi risolvono il problema con un certo grado d'incertezza oppure non assicurano la convergenza della ricerca alla soluzione se non in casi particolari, ma, d'altro canto, richiedono tempi di convergenza molto minori degli algoritmi esatti.

Fra gli algoritmi euristici è fatta distinzione fra due tipologie: i metodi "forti" e quelli "deboli". I primi sono metodi progettati allo scopo di risolvere un particolare

problema, includendo conoscenza del dominio particolare e sfruttando la rappresentazione interna del sistema in esame. In questo modo si ottengono buone prestazioni dell'algoritmo sul particolare compito da affrontare, ma esso stesso è difficilmente adattabile ad altri compiti o, comunque, ottiene risultati non soddisfacenti se applicato in ambiti diversi da quelli per i quali è stato progettato.

I metodi deboli [Rich] non incorporano o utilizzano poca conoscenza del dominio, ma sono adatti a risolvere una vasta gamma di problemi, non essendo specializzati per un problema ad hoc. Spesso sono stati ispirati dall'imitazione di metodi di risoluzione utilizzati dall'uomo o da analogie con il mondo naturale. Fra i più conosciuti si ricordano hill-climbing, depth-first e breadth-first-search.

Gli algoritmi genetici, la programmazione genetica, le strategie evolutive e la programmazione evolutiva sono tutti algoritmi di ricerca euristici, che si ispirano nel loro funzionamento alle teorie evolutive di Darwin. Essi sono, in genere, considerati metodi deboli; Angeline [Ang94], però, ha introdotto una nuova tipologia, i metodi deboli evolutivi, che meglio permette di classificare questo tipo di algoritmi. Essi sono, infatti, metodi che inizialmente hanno poca conoscenza del dominio, ma durante la loro evoluzione acquistano maggiore consapevolezza del problema, incorporando, in tal modo, alcune caratteristiche dei metodi forti. Si parla in questo caso di "intelligenza emergente", cioè si ha una comprensione del dominio che emerge durante l'evoluzione stessa.

2.2. Gli algoritmi genetici

Gli algoritmi genetici sono stati sviluppati basandosi sulle teorie evoluzionistiche di Darwin, presentate nel suo libro "On the Origin of Species by Means of Natural Selection" del 1859 e sono stati trattati per la prima volta da John Holland nel 1975. Questo tipo di algoritmi si basa sul principio darwiniano che gli elementi più "adatti" all'ambiente hanno maggiore possibilità di sopravvivere e di trasmettere le loro caratteristiche ai successori; in pratica, vi è una popolazione di individui che evolvono di generazione in generazione attraverso meccanismi simili alla riproduzione sessuale e alla mutazione dei geni. In tal modo si avrà una ricerca euristica che privilegia le zone dello spazio di ricerca dove maggiormente è possibile trovare soluzioni migliori, non

trascurando altre zone a più bassa probabilità di successo in cui saranno impiegate un minor numero di risorse.

Tipicamente un algoritmo genetico è costituito da una popolazione finita di individui di dimensione M, che rappresentano le soluzioni candidate a risolvere il problema, da una funzione di adattamento, detta fitness, che fornisce una misura della capacità dell'individuo di adattarsi all'ambiente, cioè costituisce una stima della bontà della soluzione e un'indicazione sugli individui più adatti a riprodursi, da una serie di operatori che trasformano l'attuale popolazione nella successiva, da un criterio di terminazione, che stabilisce quando l'algoritmo si deve fermare (siamo arrivati a una soluzione accettabile del problema, o abbiamo superato i limiti di tempo imposti dall'utente) e da una serie di parametri di controllo.

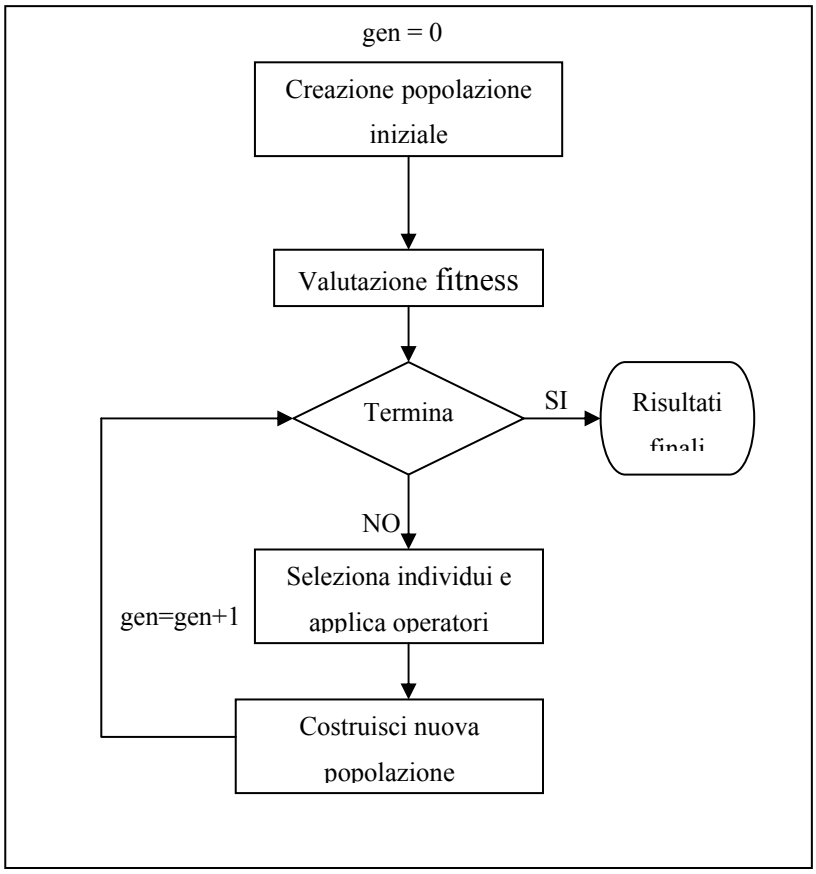

Figura 2.1. Uno schema classico di algoritmo genetico.

Uno schema semplice del funzionamento di un algoritmo genetico è illustrato in figura 2.1. Inizialmente si crea in modo del tutto casuale una popolazione di individui, dove ogni individuo è rappresentato da una stringa di lunghezza prefissata, tipicamente

binaria. Quindi, si valuta la fitness, cioè la funzione di adattamento di ognuno degli individui; si verifica se è soddisfatto il criterio di terminazione e in caso contrario si passa alla nuova generazione. La nuova popolazione sarà costruita, applicando alla vecchia gli operatori principali dell'algoritmo: il crossover, che dati due elementi selezionati nella popolazione, detti genitori, genera due "figli", cioè due individui con caratteristiche ereditate da entrambi i parenti, la mutazione che altera un singolo gene (bit) di un individuo e la riproduzione che copia un individuo inalterato nella nuova popolazione. Questi operatori, che insieme al numero di generazioni massime e alla dimensione della popolazione costituiscono i parametri fondamentali dell'algoritmo, sono applicati con diverse probabilità fino a che la nuova popolazione non ha raggiunto la dimensione desiderata.

2.2.1. La popolazione

Un elemento base degli algoritmi genetici è la popolazione che è costituita da un numero prefissato di individui ed è generata, all'inizio dell'algoritmo, in modo casuale.
Esistono diverse varianti di algoritmo genetico, dipendenti dalla gestione della popolazione, di cui citiamo le principali. Se l'intera popolazione è sostituita interamente dai nuovi elementi si parla di *generazionale*, in caso contrario si ha l'algoritmo genetico *steady-state* (a stato fissato). Ancora si ha un modello *elitarista* se l'elemento o gli elementi migliori sono conservati durante l'evoluzione della popolazione. Nel modello *a isole*, invece, si hanno una serie di popolazioni che evolvono in maniera autonoma, con occasionali migrazioni di individui da una "isola" all'altra.

Ogni individuo della popolazione è rappresentato, in genere, da una stringa di lunghezza prefissata di bit. La scelta di utilizzare prevalentemente un alfabeto binario è dovuta ad alcuni importanti risultati teorici, quali il teorema degli schemi di Holland [Holland], di cui si parlerà più in dettaglio più avanti, che indica in questa come una rappresentazione molto vicina all'ottimo. Recenti studi [Eshelman] [Janikow] [Radcliffe] indicano, però, in questa scelta alcuni svantaggi, quali aggiungere multimodalità e complessità alla funzione obiettivo, il che rende il problema più difficile da affrontare.

Continuando la similitudine con le teorie genetiche, la stringa di bit di un individuo rappresenta il *genotipo* dell'individuo, mentre il *fenotipo* indica il comportamento dell'individuo ed è completamente dipendente dal dominio. La funzione

di fitness genera un mapping dal genotipo al fenotipo; di conseguenza, due genotipi differenti potrebbero avere lo stesso valore di fitness, ma un'accurata progettazione dovrebbe evitare che ciò avvenga per due fenotipi diversi.

2.2.2. Gli operatori genetici

Negli algoritmi genetici si utilizzano principalmente tre operatori: la *riproduzione*, la *mutazione* e il *crossover*. I primi due metodi si applicano ad un solo individuo, mentre il crossover ha bisogno di due individui. Prima di applicare uno di questi operatori è necessario selezionare uno o due individui della popolazione, a seconda del caso. Fra i più popolari metodi di *selezione* si ricordano il *fitness-proportionate* e il *K-tournament*: il primo assegna ad ogni elemento della popolazione una probabilità di essere scelto, proporzionata al valore della sua fitness, mentre il secondo sceglie K elementi a caso nella popolazione, indice un torneo fra questi individui e quello che risulta vincente sarà quello selezionato. Ovviamente, con entrambi i metodi, gli individui con fitness migliore hanno maggiori probabilità di essere selezionati e, quindi, di trasmettere i propri geni alla generazione successiva, rispettando i meccanismi evolutivi di sopravvivenza dei più adatti.

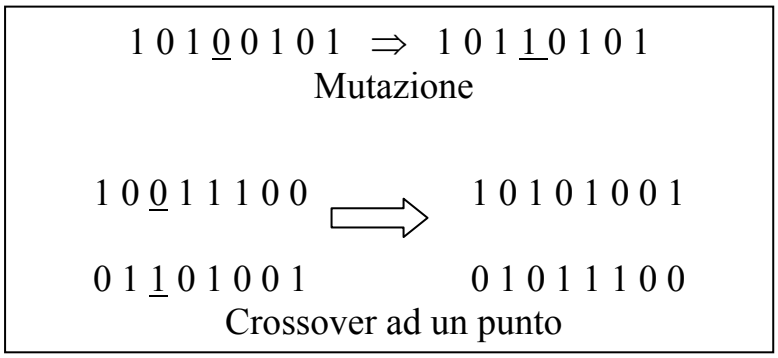

Figura 2.2. Due semplici operatori genetici

La riproduzione ricopia semplicemente l'individuo nella nuova popolazione, lasciando intatto tutto il suo patrimonio genetico. La mutazione inverte un bit, scelto casualmente con una distribuzione uniforme, del genotipo dell'individuo e inserisce in tal modo diversità nella popolazione, portando la ricerca verso nuovi spazi o recuperando allele che erano andati perduti in precedenza.

Il crossover combina i patrimoni genetici dei due genitori in modo da costruire due "figli", che possiedono metà dei geni di uno e metà dell'altro. Esistono diverse tipologie di crossover. Il crossover *ad un punto* sceglie in modo casuale una posizione della stringa e genera i figli nel modo seguente: supponendo che la stringa sia di lunghezza L, si sceglie K estraendolo a caso nell'intervallo 1..L, quindi il primo figlio avrà i geni da 1..K del primo genitore e da K+1 dell'altro e il secondo l'inverso. Altre tipologie largamente usate di crossover sono quello *a due punti*, in cui le stringhe sono scambiate, con procedimento analogo al precedente, fra due punti scelti a caso e il crossover uniforme, in cui ogni bit è scambiato con posizioni selezionate a caso.

Il crossover è la forza trainante dell'algoritmo genetico ed è l'operatore che maggiormente influenza la convergenza, anche se un suo utilizzo troppo spregiudicato potrebbe portare ad una convergenza prematura della ricerca su qualche massimo locale. I parametri che regolano la minore o maggiore influenza di un operatore su un altro sono la probabilità di crossover, di mutazione e di riproduzione; la loro somma deve essere uguale ad uno, anche se esistono varianti di algoritmo genetico che realizzano in ogni caso la mutazione, anche dopo il crossover per mantenere maggiori diversità nella popolazione. In realtà, la scelta dei parametri e del tipo particolare di operatore usato dipendono fortemente dal dominio del problema, perciò non è possibile, a priori, stabilire le specifiche di un algoritmo genetico. Addirittura sono stati sviluppati con buon successo algoritmi ibridi che sostituiscono la mutazione con un algoritmo fortemente dipendente dal dominio [Folino01].

2.2.3. Alcuni cenni teorici

Esiste pochissima teoria sul funzionamento effettivo degli algoritmi genetici; in particolare, Holland ha tentato di spiegare la distribuzione delle risorse nello spazio di ricerca con il suo famoso teorema degli schemi [Holland]. Si considera un alfabeto formato dai simboli 0,1,*; se si ha un genotipo binario di L elementi, uno **schema** è una stringa di L simboli appartenenti al suddetto alfabeto. Se * può indicare al contempo sia 0 che 1, allora uno schema rappresenta un insieme di stringhe e lo schema può essere più o meno adatto a sopravvivere nell'ambiente, a seconda che le stringhe che identifica abbiano una fitness più o meno elevata. Il crossover e la mutazione alterano gli schemi definiti e possono introdurne di nuovi.

Il teorema degli schemi dimostra che, utilizzando una selezione di tipo fitness-proportionate, la distribuzione degli schemi di ricerca, cioè l'aumento o la diminuzione di un particolare schema, avviene in modo molto vicino all'ottimo matematico ed è indipendente dal problema.

Numerose critiche sono state mosse a questo teorema, alcune delle quali sulla reale applicabilità in casi pratici; sono stati sviluppati, infatti, GA che non soddisfano le condizione del teorema, ma ottengono risultati simili all'algoritmo classico. Altre critiche sono state mosse sugli aspetti teorici, in particolare si è notato che il teorema degli schemi non esplica le correlazioni esistenti fra i padri e i figli. Il teorema di Price [Price][Alt] spiega meglio queste relazioni ed asserisce che il funzionamento e l'efficienza di un GA dipende maggiormente dalle correlazioni fra genitore e figlio che non dagli schemi stessi.

2.3. Le strategie e la programmazione evolutiva

Le strategie evolutive [Fogel97] sono state sviluppate all'Università di Berlino da Rechenberg, Bienert, Schwefel, Bäck, Hoffmeister. Diversamente dagli algoritmi genetici, la popolazione è costituita da vettori di lunghezza prefissata di reali e la popolazione iniziale è generata in modo casuale, in un certo intervallo e utilizzando una distribuzione uniforme. Una nuova popolazione è costituita dai migliori M individui di quella precedente e i rimanenti sono scelti applicando dei particolari operatori. I principali sono la *ricombinazione discreta*, che genera i figli con distribuzione uniforme rispetto ai genitori, la *ricombinazione intermedia*, che fa la media dei valori dei due genitori e la *ricombinazione intermedia casuale*, che determina in modo casuale i pesi da attribuire ai genitori. Esiste anche un operatore di *mutazione* che aggiunge un valore casuale, preso da una distribuzione gaussiana con varianza scelta dall'utente in modo appropriato a ogni elemento del vettore. Il principale vantaggio derivante dall'usare queste strategie è che elementi della popolazione con piccole diversità, dovrebbero presentare poca differenza nei comportamenti. Questo è vero, ma non sempre si riescono a generare figli con caratteristiche abbastanza simili ai genitori.

La programmazione evolutiva è stata introdotta da Fogel, Owens e Walsh [Fogel95]. Ogni individuo della popolazione costituisce una FSM (macchina a stati finiti), ed è formato da una serie di stati interni facenti parte di un alfabeto finito. La

caratteristica principale di una FSM è che riceve in input una serie di simboli e restituisce in output una serie di stati, basandosi solo sugli stati correnti e l'input. Ebbene, l'obbiettivo della programmazione evolutiva è predire la prossima configurazione del sistema. L'operatore usato è quello di mutazione che altera lo stato iniziale, modifica la transizione o cambia uno stato interno. La caratteristica fondamentale di questo tipo di algoritmi è che i figli hanno un comportamento simile a quello dei genitori.

2.4. La programmazione genetica

Gli algoritmi genetici sono un'applicazione dell'intelligenza artificiale che imita i processi di evoluzione naturali per far evolvere stringhe di bit alla ricerca della soluzione di un problema. Se ad evolversi non fossero delle semplici stringhe, ma complessi programmi per computer vi sarebbe una più stretta analogia fra l'evoluzione naturale e quella artificiale. Quest'approccio è stato sperimentato da John R. Koza [Koza92] che ha introdotto la programmazione genetica, per mezzo della quale programmi per computer si evolvono combinandosi, riproducendosi o mutando per dar luogo ad altri programmi che meglio si adattano a risolvere un determinato problema.

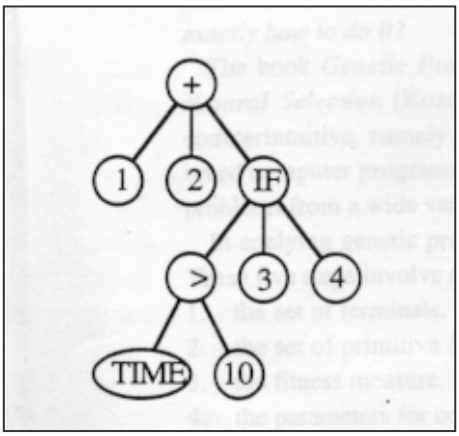

Figura 2.3. Una rappresentazione ad albero di un programma.

Lo spazio della rappresentazione utilizzato per i programmi è quello degli alberi, dal momento che questo tipo di rappresentazione risulta essere la più compatta è quella che si adatta meglio a essere sottoposta a vari operatori. Se si scrive un programma in un linguaggio di tipo funzionale, quale ad esempio il LISP [Pagan], allora sarà naturale trasformarlo in una rappresentazione ad albero, dove i nodi corrispondono alle funzioni

e le foglie ai simboli terminali. Ad esempio, l'espressione IF (TIME > 10) THEN return 1+2+3 ELSE return 1+2+4 può essere scritta in LISP come + (1 2 (IF (> TIME 10) 3 4)) e quindi trasferita in un albero come si vede nella figura 2.3, tratta dal libro di Koza. Da notare che l'IF ha tre argomenti, il primo è l'espressione da valutare, il secondo costituisce il ramo THEN e il terzo il ramo ELSE.

La funzione di fitness può essere valutata semplicemente facendo girare il programma su un set di istanze di validazione e calcolando l'errore, anche se può essere utile aggiungere a questa funzione altri membri che tengano conto anche della complessità dei programmi generati, dell'efficienza e di altri fattori.

2.4.1. I parametri principali

La programmazione genetica si presenta strutturalmente più complessa rispetto agli algoritmi genetici, dal momento che ci sono molti più parametri da considerare nella progettazione.
Di seguito elenchiamo le principali scelte che devono essere valutate:

- La generazione della popolazione iniziale.
- L'insieme di funzioni e terminali di base.
- Il tipo di selezione.
- La dimensione della popolazione e il numero massimo di generazioni.
- Il criterio di terminazione.

La popolazione iniziale è generata in modo random, cioè ogni individuo è creato scegliendo in modo casuale una funzione o un terminale dai rispettivi insiemi e poi gli eventuali argomenti della funzione sono costruiti con lo stesso meccanismo in maniera ricorsiva. Esistono principalmente tre metodi di generazione che danno luogo ad alberi di dimensione e forma differenti. Il metodo *full* genera alberi per i quali la lunghezza di ogni percorso dalla radice ad un qualsiasi terminale è pari a una costante di profondità massima. Con il metodo *growth*, gli alberi hanno una profondità che è variabile fino al limite di quella massima. Il metodo *ramped half-and-half* cerca di prendere il meglio dai due precedenti, combinandoli insieme; si sceglie un parametro variabile che va da 2 alla profondità massima e per ogni suo valore si genera una percentuale della popolazione con tale parametro usato come profondità massima (per metà di questi alberi è usato il metodo growth, per l'altra il metodo full). Usando questa tecnica si riesce a creare una

varietà di alberi per forma e dimensione, instaurando un buon grado di diversità nella popolazione. Infatti sperimentalmente, in un gran numero di casi, con questo metodo di generazione si sono ottenuti i risultati migliori che con gli altri due. Un'operazione aggiuntiva utile per aumentare la diversità nella popolazione è quella di controllare, durante la creazione, se un individuo esiste già e in caso affermativo sostituirlo con uno nuovo.

Un programma per computer è costituito da funzioni alle quali sono passati argomenti che possono essere altre funzioni o simboli terminali (costanti, variabili, numeri random, ecc..). Nella programmazione genetica la fase iniziale prevede la definizione dell'insieme di base di funzioni e terminali da cui creare la popolazione iniziale. La scelta di questi insiemi dipende fortemente dal dominio particolare del problema; per esempio se si vuole ricercare un'espressione matematica intera con valori da 0 a 9, i simboli di funzione potrebbero essere {+, *, - , \} e i terminali le cifre {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Le funzioni scelte devono soddisfare la proprietà di *chiusura*, cioè devono poter accettare come parametri qualsiasi valore restituito dall'insieme definito delle funzioni e qualsiasi valore che assumono i terminali, altrimenti si potrebbero generare programmi non corretti. Inoltre, devono, avere la proprietà di *sufficienza*, cioè è necessario che le funzioni unite ai simboli terminali siano in grado di generare una soluzione del problema. Perciò, la scelta di un insieme di funzioni e terminali appropriate ad un particolare dominio risulta essere un punto critico nella riuscita di un algoritmo di programmazione genetica; infatti, non sempre si conosce l'insieme di funzioni sufficienti alla risoluzione di un problema e, del resto, sceglierne un insieme ridondante può degradare in maniera significativa le prestazioni del sistema.

La selezione degli elementi su cui applicare gli operatori influenza la convergenza dell'algoritmo; infatti, una maggiore pressione selettiva comporta una più veloce convergenza dell'algoritmo, però può portare ad una perdita della diversità della popolazione e la soluzione ottima potrebbe non essere mai raggiunta. I metodi più usati sono: *fitness-proportionate selection*, in cui la probabilità che ogni individuo sia selezionato è pari alla sua fitness normalizzata, *rank selection*, in cui gli elementi sono ordinati e scelti in base alla loro fitness relativa e non assoluta, *tournament selection*, in cui un numero fissato di elementi è estratto in modo casuale dalla popolazione, e quello che possiede la fitness migliore viene selezionato. Fra questi metodi, il primo è quello che, generalmente, comporta una minor pressione selettiva.

La dimensione della popolazione è un altro dei parametri che influenzano la convergenza dell'algoritmo, insieme al numero massimo di generazioni; infatti un sottodimensionamento potrebbe portare al non raggiungimento dell'ottimo cercato e del resto al loro aumentare l'algoritmo diventa computazionalmente troppo espansivo.

2.4.2. Mutazione, crossover ed altri operatori

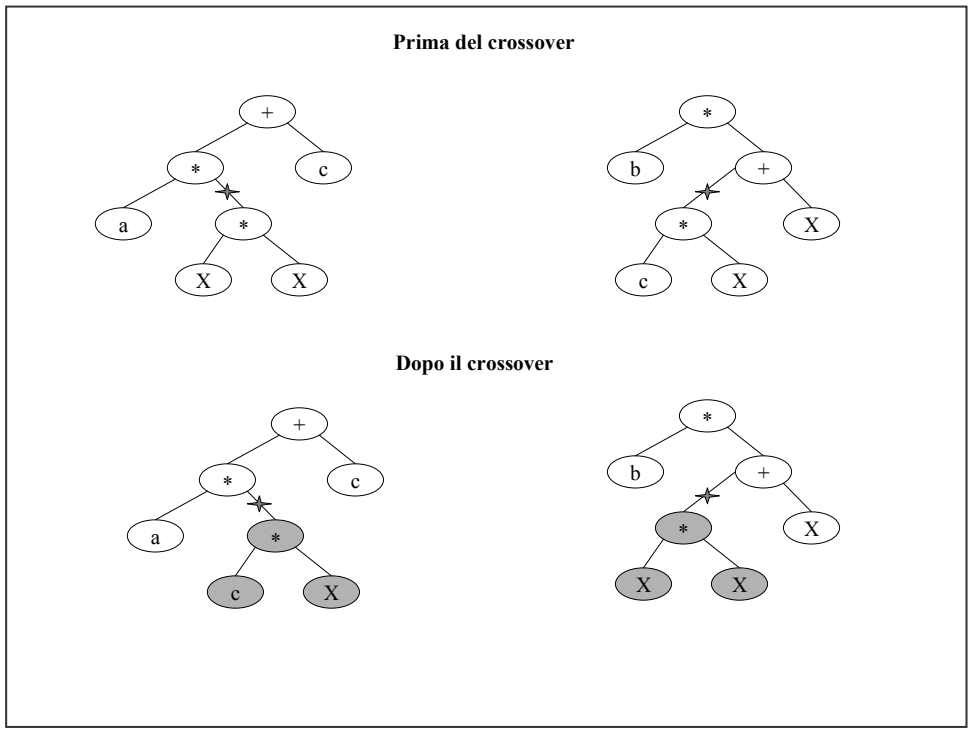

Figura 2.4. Un esempio di crossover.

Il passaggio da una generazione all'altra avviene attraverso l'applicazione di definiti operatori unari o binari ad elementi della popolazione selezionati con i metodi descritti nel precedente paragrafo. I suddetti operatori sono applicati in modo esclusivo con una certa probabilità; ovviamente la somma delle probabilità sarà uguale ad uno. Nella programmazione genetica la forza trainante dell'algoritmo è costituita dall'operatore di **crossover**, di gran lunga più usato rispetto agli altri operatori quali **mutazione**, **permutazione**, **editing**, **incapsulamento** e **decimazione**.

Il crossover è analogo a quello degli algoritmi genetici, con la sola differenza che è applicato a degli alberi e non a stringhe di bit. E' scelto un ramo a caso per ogni genitore, e le due parti sono combinate uno con l'altro, in modo da generare due alberi figli. Il procedimento è esplicato in figura 2.4, nella quale due alberi rappresentanti polinomi sono combinati per dare vita ad altri due polinomi differenti. Ancora esistono

dei parametri che fissano dei limiti sulla dimensione massima degli alberi della popolazione, perciò l'implementazione deve curare anche questo aspetto.

La mutazione è applicata ad un albero selezionato, scegliendo un punto a caso, spezzandolo l'albero e sostituendo la parte sottostante con un nuovo sottoalbero generato in modo casuale (vedi figura 2.5), rispettando sempre un parametro di lunghezza massima del sottoalbero.

La permutazione è attuata scegliendo un nodo interno (una funzione) a caso dell'albero; se la funzione ha k argomenti, è estratta una delle k! permutazioni e gli argomenti sono arrangiati di conseguenza. Ovviamente, se la funzione è commutativa l'operatore non ha nessun effetto. L'editing permette di semplificare un sottoalbero valutando l'espressione risultante: ad esempio, (OR X X) può essere sostituita con X. L'operazione può essere effettuata durante la fase di run con una certa frequenza (ogni generazione o più raramente) o sul risultato finale. Nel primo caso, il suo utilizzo potrebbe causare una prematura convergenza dato che le espressioni semplificate sono più soggette ad un effetto distruttivo del crossover, che sarà trattato più ampiamente nel paragrafo successivo.

Con l'incapsulamento, un sottoalbero che parte da un nodo interno di un albero selezionato è scelto e compilato; quindi tale sottoalbero va a far parte dei terminali e non è soggetto perciò all'effetto distruttivo del crossover. In tal modo sono salvate le funzioni più interessanti.

La decimazione seleziona una certa percentuale (ad esempio il 20%) della popolazione in una determinata generazione, in base ai valori della fitness, senza permettere duplicati e il resto della popolazione è cancellata. Questo risulta utile per mantenere la diversità in quei casi dove pochi elementi hanno una fitness molto più elevata degli altri e questi avrebbero il sopravvento in poche generazioni.

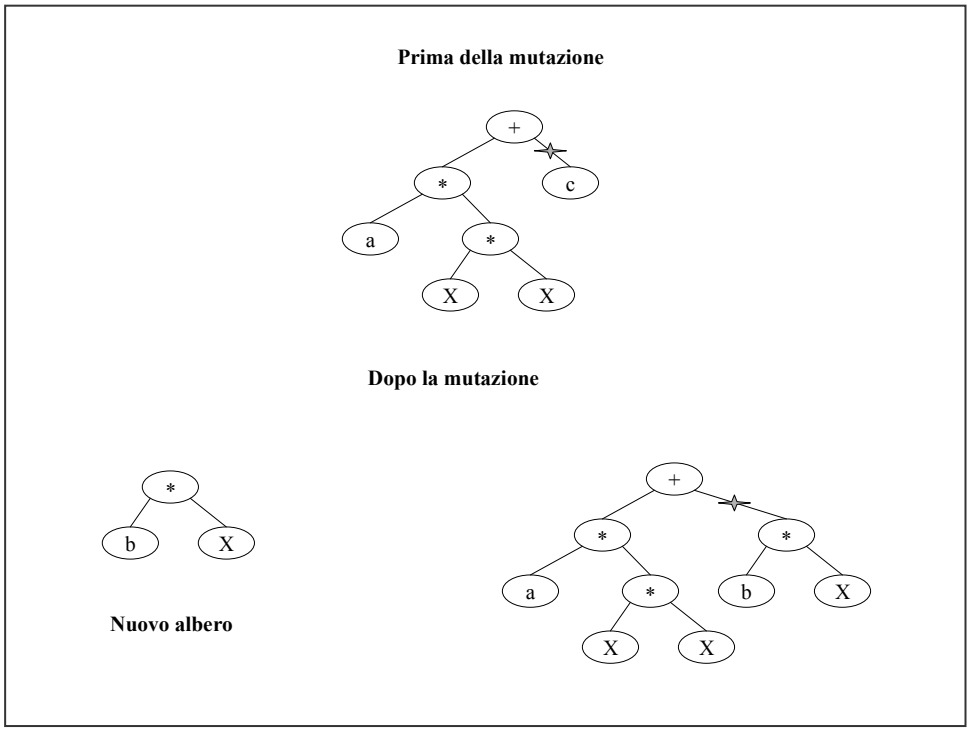

Figura 2.5. Un esempio di mutazione.

Nella programmazione genetica l'operatore più usato è senz'altro il crossover che, diversamente dal caso degli algoritmi genetici, basta da solo a garantire il mantenimento della diversità nella popolazione; in genere è usato in congiunzione con la mutazione che ha una probabilità minore; gli altri operatori sono usati solo in casi particolari, nei quali è possibile sfruttare uno dei vantaggi descritti in precedenza.

Gli operatori sopracitati garantiscono la correttezza sintattica degli alberi generati, grazie alla proprietà di chiusura, ma non la correttezza semantica. Sarà la fitness, penalizzando gli alberi che non rispettano le regole semantiche, a favorire la crescita di alberi semanticamente corretti.

2.4.3. Programmi più complessi lavorano meglio?

Se non si adottano strategie per limitare la complessità di un programma generato, essa cresce con il passare delle generazioni. L'aumento della complessità produce effetti negativi, in quanto il programma risulta di difficile interpretazione per un programmatore umano e non generalizza bene in accordo al principio del rasoio di Occam, che sostiene che le soluzioni più semplici sono da preferire in quanto evitano l'aggiunta del superfluo e, quindi, sono applicabili ad una casistica più generale.

Nella programmazione genetica, fra le cause dell'aumento della complessità degli alberi generati si riscontra un particolare fenomeno detto *bloating*, che indica l'utilizzo di una quantità di simboli in eccesso rispetto a quelli effettivamente necessari a esprimere un concetto (ad esempio (log e + log e + log e) al posto della semplice espressione (3 * log e). Fra le cause scatenanti del bloating hanno particolare rilievo i cosiddetti *introns*, che prendono il nome da quella parte del DNA che non viene trascritta nelle proteine. In GP, gli introns sono definiti come quella parte dell'albero del programma che non altera il fenotipo, cioè non è utilizzato nel calcolo della fitness, come ad esempio il blocco (X AND X) che non modifica la semantica del risultato. Eliminare questa parte superflua del codice potrebbe sembrare la soluzione migliore per ovviare ai problemi sopracitati, ma non sempre l'eliminazione di cose in apparenza inutili non comporta effetti collaterali negativi. Vari studi ed esperimenti [Ang94][Nordin95][Nordin96] hanno evidenziato un effetto positivo del bloating nel preservare soluzioni interessanti dagli effetti distruttivi del crossover. Infatti, se un programma contiene parti non utilizzabili per il calcolo della fitness, sarà più probabile che il crossover non alteri la fitness di tale programma. Perciò, l'eliminazione o meno degli introns deve essere attentamente considerata nel ridurre la complessità del sistema.

In genere, per ottenere soluzioni più semplici, si agisce sui parametri del sistema fissando la profondità massima degli alberi o si aggiunge alla fitness una funzione di parsimonia che favorisca gli alberi più corti o, ancora, si modificano gli operatori genetici in modo da ridurre le ridondanze. Limitare la profondità massima può essere difficoltoso se non si conosce la dimensione della soluzione del problema, rischiando così di escluderla perché fuori dal limite accettato. Ancora, aggiungere la funzione di penalizzazione alla fitness (parsimonia) non favorisce lo sviluppo naturale del sistema e può penalizzare la ricerca della soluzione ottima.

In definitiva, il problema di utilizzare alberi più o meno complessi resta un problema aperto, dal momento che la riduzione dell'albero se da una parte genera soluzioni più compatte e generali, dall'altra può portare ad una prematura convergenza e al non raggiungimento della soluzione ottimale e non può prescindere dal dominio particolare.

2.4.4. Differenze e analogie fra GP e GA

Il genotipo degli algoritmi genetici è, in genere, rappresentato con una stringa di lunghezza prefissata di bit, mentre nella programmazione genetica esso è costituito da un albero formato da funzioni e terminali, di lunghezza variabile, anche se, solitamente, è fissata una profondità massima che ne limita la dimensione. Niente vieterebbe, quindi, di rappresentare un albero di GP, sicuramente limitato in dimensione da una profondità massima o da necessità di implementazione, con una stringa di bit di lunghezza fissata pari a quella massima dell'albero, cioè di utilizzare un algoritmo genetico. Necessità di ordine pratico rendono più facile e naturale usare GP per far evolvere programmi per computer che non GA. Da non sottovalutare il fascino naturale che far evolvere programmi per computer invece che innaturali stringhe di bit senza significato esercita sull'uomo e rende più spontaneo la scelta della programmazione genetica.

L'operatore prevalente è, in entrambi i casi, quello di crossover, ma mentre in GP esso è sufficiente da solo a mantenere la diversità nella popolazione, in GA è la mutazione che si sobbarca questo compito, in modo da evitare una prematura convergenza, con l'introduzione di nuovi elementi per la ricerca o il recupero di elementi precedentemente persi. Nella programmazione genetica, il crossover fra più alberi può introdurre una nuova semantica negli alberi generati, spostando la ricerca verso un'altra zona, più di quanto possa fare l'analogo operatore degli algoritmi genetici rendendo superflua la mutazione.

Un altro fattore importante di differenza fra i due metodi è la sensibilità ai parametri che sono spesso di fondamentale importanza per la convergenza di un algoritmo genetico, mentre non influenzano di molto il raggiungimento di una soluzione ottima per la GP; in questo caso la fase di tuning dei parametri risulta più semplice e non necessita di molto sforzo.

Un altro punto da tenere in considerazione è che, riguardo agli algoritmi genetici, l'interpretazione di un allele dipende fortemente dalla sua posizione nella stringa di bit, mentre un istruzione per computer ha lo stesso significato indipendentemente dalla posizione anche se, ovviamente, può generare comportamenti semantici diversi; perciò questo aspetto facilita l'emergere di dinamiche naturali insite nel processo evolutivo della programmazione genetica.

3. Gli algoritmi evolutivi paralleli.

La programmazione genetica e gli algoritmi evolutivi in genere hanno bisogno di grandi popolazioni per ottenere una buona convergenza. Questo aspetto, unito al fatto che calcolare la fitness di un singolo individuo richiede la valutazione su un certo training set, porta ad uno sforzo computazionale notevole. Se, ad esempio consideriamo una popolazione M, un training set di dimensione T e facciamo girare il nostro algoritmo per un numero di generazioni G, allora il tempo totale di valutazione sarà dell'ordine di GxTxM per il tempo di valutazione di una singola fitness, il che può essere inaccettabile anche su workstation molto potenti. Ancora, nel caso particolare della programmazione genetica un singolo individuo non è più una stringa di bit, ma un albero di programma, che può anche essere molto complesso; in questo caso il problema non riguarda solo la dimensione temporale, ma anche quella spaziale che può superare la capacità di memorizzazione di una singola macchina. Da qui la necessità di realizzare un'implementazione parallela di questo tipo di algoritmi su ambienti di tipo distribuito, per ovviare anche al problema della memoria insufficiente.

3.1. I modelli per l'implementazione di algoritmi evolutivi paralleli

La programmazione genetica risulta implicitamente parallela, in quanto la valutazione della fitness di ogni individuo, che è poi la fase più dispendiosa dell'algoritmo, risulta indipendente da ogni altro elemento della popolazione e può avvenire in parallelo. La fase problematica, invece, è la selezione degli individui nella popolazione che ha bisogno, nell'ipotesi di una distribuzione della popolazione, di accessi remoti che riducono la possibilità di ottenere speed-up lineari dell'algoritmo parallelo.

In genere gli algoritmi paralleli evolutivi seguono una politica di selezione locale, utilizzando un modello di tipo diffusivo, detto anche cellulare, in cui la selezione è limitata solo ad elementi "vicini" e non a tutta la popolazione oppure dividendo la popolazione in sottopopolazioni, dette isole, e limitando la selezione all'interno di una stessa isola, con occasionali migrazioni da un'isola all'altra. Queste tecniche hanno il vantaggio di non diffondere troppo velocemente soluzioni sub-ottime e, di conseguenza, evitano una prematura convergenza dell'algoritmo. Nei successivi sottoparagrafi

saranno descritti in dettaglio i modelli principali per l'implementazione di algoritmi paralleli evolutivi.

3.1.1. Il modello globale

Erick Cantù-Paz [Cantù99b][Cantù97] indica fra i metodi per parallelizzare un algoritmo evolutivo quello *globale*, in cui è mantenuta sempre una singola popolazione, ma la valutazione della fitness degli individui è eseguita in parallelo, dividendo il carico fra i processori disponibili. Le fasi di crossover e mutazione potrebbero essere fatte in parallelo, ma il tempo necessario alla distribuzioni degli individui, annullerebbe i vantaggi ottenuti, così è preferibile eseguire queste operazioni in sequenziale. La selezione è quella classica che agisce su tutta la popolazione, da qui il nome globale per il modello. Cantu-Paz suggerisce di implementare il metodo su un'architettura distribuita utilizzando un modello di tipo master-slave, in cui il master mantiene la popolazione e la distribuisce ai vari slave al momento della valutazione della fitness.

Questo tipo di modello ha il vantaggio di preservare il comportamento dell'algoritmo sequenziale anche in parallelo, ma la fase di selezione richiede frequenti accessi all'intera popolazione e il conseguente overhead di comunicazione risulta inaccettabile a meno che non ci siano funzioni di fitness particolarmente pesanti da valutare.

3.1.2. Il modello a isole

Un'alternativa al modello precedente è quella di utilizzare un diverso meccanismo di selezione limitato agli individui posti su un unico processore. Classico esempio di questa tecnica è il modello ad isole, nel quale la popolazione è suddivisa in sottopopolazioni, dette isole, che sono distribuite, usualmente, una per processore. L'implementazione di tale modello prevede la creazione di sottopopolazioni random all'interno di ogni singolo nodo; l'esecuzione di tutti gli operatori e la valutazione della fitness viene eseguita a livello locale. In tal modo l'algoritmo presenta una scalabilità quasi lineare e risulta di facile implementazione (è sufficiente lanciare un processo dell'algoritmo per ogni nodo).

Una variante di tale modello prevede la migrazione di elementi da una sottopopolazione all'altra, allo scopo di introdurre diversità nelle isole e garantire una convergenza migliore. La migrazione non è, in genere, molto frequente ed è limitata a

pochi elementi per generazione, quindi la scalabilità si mantiene ancora pressoché lineare. In realtà, risulta abbastanza complesso fissare parametri quali la probabilità di migrazione, la sua frequenza e il criterio di scelta degli elementi da trasferire, visto che possono essere determinanti per la buona riuscita dell'algoritmo, come è evidenziato da alcuni studi [Cantù99][Cantù97b]. In letteratura sono conosciute, principalmente, due implementazioni riguardanti la parallelizzazione della programmazione genetica utilizzando un modello ad isole [Ouss97][Andre]. Un aspetto ancora non ben compreso di questo modello riguarda il comportamento della convergenza rispetto al sequenziale classico.

3.1.3. Il modello diffusivo

Nel modello diffusivo [Pettey], la popolazione è distribuita spazialmente, solitamente su una griglia bidimensionale. Ogni individuo k è sostituito nella successiva generazione da un nuovo elemento generato applicando l'operatore di crossover o mutazione ad elementi appartenenti al vicinato di k stesso, come si vede in figura 3.1.

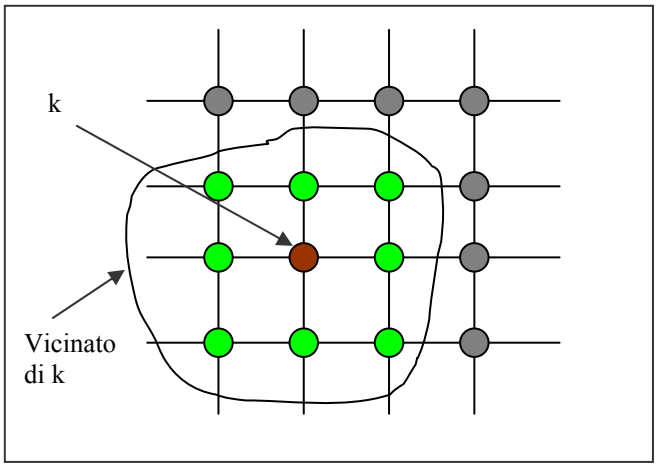

Figura 3.1. Un esempio di selezione locale

Dal momento che un modello di questo tipo è equivalente ad un automa cellulare, esso è detto anche modello cellulare.

La selezione ristretta ad un vicinato locale rende l'implementazione parallela molto efficiente, dal momento che le comunicazioni sono limitate solo ai vicini e, di conseguenza, non c'è un overhead eccessivo. Ancora, soluzioni sub-ottime non si diffondono velocemente all'interno dell'intera popolazione, favorendo lo svilupparsi di nicchie in cui si esplorano diverse porzioni dello spazio di ricerca e viene evitata una

prematura convergenza. Il modello cellulare è uno dei più promettenti paradigmi di programmazione degli ultimi tempi [Snipper]; infatti non solo si presta ad una naturale ed efficiente implementazione parallela, ma ottiene spesso una convergenza migliore grazie alle politiche di selezione locale che facilitano lo svilupparsi di nicchie che operano su spazi di ricerca differenti.

Sulle implementazioni parallele degli algoritmi genetici esistono ampie rassegne in letteratura. Nel successivo paragrafo si darà una carrellata, invece, sulle implementazioni parallele esistenti riguardanti la programmazione genetica, sottolineandone limiti e sviluppi futuri.

3.2. La programmazione genetica parallela: lo stato dell'arte

Andre e Koza [Andre] [Koza95] hanno sviluppato un'implementazione ad isole, con il supporto della migrazione occasionali di individui da un'isola all'altra, su un network di transputer, ottenendo speedup quasi lineari relativi al codice implementato e super-lineari in rapporto a problemi risolti con la programmazione genetica classica. Juillé e Pollack [Juillé] hanno realizzato un'implementazione SIMD che precompila gli alberi per una macchina virtuale a stack. Stoffel e Spector [Stoffel] hanno sviluppato un sistema ad alte prestazioni per la programmazione genetica che manipola programmi lineari per una macchina virtuale a stack; il sistema gira su macchine di tipo MIMD e raggiunge speedup quasi lineari, nonostante supporti una strategia di selezione globale. Niwa e Iba hanno implementato un ambiente di programmazione genetica distribuito su un architettura parallela di tipo MIMD, utilizzando un modello ad isole con migrazione, come Koza; il loro studio si differenzia per aver sperimentato il modello su varie topologie e per aver valutato la diversità strutturale dei vari approcci. Oussaidène e altri [Ouss96][Ouss97] hanno realizzato un'implementazione parallela, seguendo un modello ad isole con migrazione, in cui però la fitness è valutata seguendo il paradigma master-slave ed è presente, inoltre, un algoritmo di bilanciamento del carico dinamico che ovvia al problema di una diversa complessità degli alberi della popolazione. Sahli ed altri [Sahli] hanno implementato un tool per la programmazione genetica orientata alla regressione simbolica, che combina sia il modello classico ad isole con migrazione che quello a nicchie; per alcuni problemi di regressione hanno riscontrato speed-up superlineari. Kent e Dracopoulus hanno utilizzato il modello BSP (Bulk Synchronous

Parallel) per implementare la programmazione genetica in parallelo su un cluster di workstation Sun e hanno sfruttato un paradigma di tipo master-slave, in cui gli slave supportano il master nella valutazione della fitness; quindi hanno realizzato un modello per valutare lo speedup del sistema e hanno riportato valori di questo indice quasi lineari. Folino ed altri [Folino01bis] hanno realizzato un implementazione scalabile che utilizza il modello cellulare su architetture distribuite utilizzando le librerie MPI ed hanno verificato che la convergenza dell'algoritmo risulta migliore rispetto al modello sequenziale canonico e ad alcune implementazioni ad isole esistenti in letteratura.

4. Conclusioni

Questo lavoro ha passato in rassegna le varie tipologie di algoritmi evolutivi. Il gran numero di parametri da progettare in tali algoritmi può essere ridotto da una conoscenza del dominio applicativo. Diversamente dagli algoritmi genetici, la programmazione genetica risulta essere abbastanza robusta alla maggior parte dei parametri e quindi più facilmente progettabile una volta che siano fissate le funzioni e i terminali dipendenti dal dominio applicativo.

Per risolvere problemi di una certa difficoltà è necessario aumentare la dimensione della popolazione e/o il numero massimo di generazioni. Questo rende la parallelizzazione indispensabile per poter risolvere problemi reali in tempi accettabili e con memorie sufficienti. Inoltre, una caratteristica da rilevare nella parallelizzazione della programmazione genetica è che, differentemente dagli algoritmi genetici, gli individui della popolazione non sono stringhe di bit di lunghezza prefissata, ma programmi per computer di lunghezza variabile. Questo costituisce il punto critico da valutare; infatti ogni programma non solo esegue istruzioni differenti ma presenta anche differente dimensione e complessità e perciò richiede differenti tempi di esecuzione. Perciò, tecniche di bilanciamento del carico o l'utilizzo di algoritmi asincroni deve essere impiegato per ottenere implementazioni efficienti.

5. Bibliografia

[Alt] L. Altenberg, “The Schema Theorem and Price’s Theorem”, Foundations of Genetic Algorithms 3, Morgan Kaufmann. Estes Park, Colorado, 1995.

[Andre] A. David and J. R. Koza, “Exploiting the fruits of parallelism: An implementation of parallel genetic programming that achieves super-linear performance”, Information Science Journal, Elsevier, 1997.

[Ang93] P. J. Angeline, “Evolutionary Algorithms and Emergent Intelligence”, Ph.D. thesis, The Ohio State University, Department of Computer and Information Science, 1993.

[Ang94] P. J. Angeline, “Genetic Programming and Emergent Intelligent”, Advances in Genetic Programming, K. Kinnear, Cambridge MA: MIT Press, pp. 75-96.

[Back] T. Bäck, U. Hammel and H. Schwefel, “Evolutionary Computation: Comments on the History and Current State”, IEEE Transactions on Evolutionary Computation, Vol. 1, N. 1, April 1997.

[Cantú97] E. Cantu-Paz, “Designing Efficient Master-Slave Parallel Genetic Algorithms”, Technical Report IlliGAl 9704, Dept. Comp. Science, University of Illinois at Urbana-Champaign, 1997.

[Cantú97b] E. Cantú-Paz and D. E. Goldberg, “Modeling Idealized Bounding Cases of Parallel Genetic Algorithms”, Genetic Programming 1997, Proceedings of the Second Annual Conference, MIT Press, Stanford University, July 1997.

[Cantú99] E. Cantú-Paz, ”Migration Policies and Takeover Times in Parallel Genetic Algorithms”, GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, San Francisco, July 1999.

[Cantú99b] E. Cantu-Paz, “Migration policies, selection pressure, and parallel evolutionary algorithms”, Technical Report IlliGAl 99015, Dept. Comp. Science, University of Illinois at Urbana-Champaign, 1999.

[Deakin] A. G. Deakin and D. F. Yates, “Genetic Programming Tools Available on the Web: A First Encounter”, Genetic Programming 1996, Proceedings of the First Annual Conference, MIT Press, Stanford University, July 1996.

[Draco] D. C. Dracopoulos and S. Kent, “Speeding up Genetic Programming: A Parallel BSP implementation”, Genetic Programming 1996, Proceedings of the First Annual Conference, MIT Press, Stanford University, July 1996.

[Eshelman] L. J. Eshelman and J. D. Shaffer, “Real-coded genetic algorithms and interval-schemata”, Foundations of Genetic Algorithms 2, Morgan Kaufmann, San Mateo CA, 1993.

[Fogel95] D. B. Fogel, “Evolutionary Computation – Towards a New Philosophy of Machine Intelligence”, IEEE Press, 1995.

[Fogel97] D. B. Fogel, “Evolutionary Programming and Evolutionary Strategies Tutorial”, Genetic Programming 1997, Proceedings of the Second Annual Conference, MIT Press, Stanford University, July 1997.

[Folino01] G. Folino, C. Pizzuti,G. Spezzano, "Parallel Hybrid Method for SAT that Couples Genetic Algorithms and Local Search″, IEEE Transactions on Evolutionary Computation, vol. 5 N. 4, August 2001.

[Folino01bis] G. Folino, C. Pizzuti, G. Spezzano, ”CAGE: A Tool for Parallel Genetic Programming Applications”, EuroGP2001: 4th European Conference on Genetic Programming, Springer-Verlag, Lake Como, April 2001.

[Koza92] J. R. Koza, “Genetic Programming: On the Programming of Computers by means of Natural Selection”, MIT Press, Cambridge, 1992.

[Koza94] J. R. Koza, “Genetic Programming II: Automatic Discovery of Reusable Programs”, MIT Press, Cambridge, 1994.

[Koza95] J. R. Koza and D. Andre, “Parallel Genetic Programming on a Network of Transputers”, Technical Report CS-TR-95-1542, Dept. Comp. Science, University of Stanford, 1995.

[Kumar] V. Kumar, A. Grama, A. Gupta, G. Karypis, “Introduction to Parallel Computing: Algoritm Design and Analysis”, Benjamin Cummings/ Addison Wesley, Redwod City, 1994.

[Holland] J. Holland, “Adaptation in Natural and Artificial Systems”, University of Michigan Press, 1975.

[Horn] J. Horne, “The Nature of Niching: Genetic Algorithms and the Evolution of Optimal, Cooperative Populations”, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1997.

[Janikow] C. Z. Janikow and Z. Michalewicz, “An experimental comparison of binary and floating point representations in genetic algorithms”, Proceedings 4th International Conference on Genetic Algorithms”, Morgan Kaufmann, San Mateo CA, 1991.

[Juillé95] H. Juillé, J. B. Pollack, “Parallel Genetic Programming on Fine-Grained SIMD Architectures”, Working Notes of the AAAI-95 Fall Symposium on Genetic Programming”, AAAI Press.

[Juillé96] H. Juillé, J. B. Pollack, “Massively Parallel Genetic Programming”, Advances in Genetic Programming: Volume 2, P. Angeline and K. Kinnear, MIT Press, Cambridge, 1996.

[Niwa] T. Niwa and H. Iba, “Distribuited Genetic Programming – Empirical Study and Analisys –”, Genetic Programming 1996, Proceedings of the First Annual Conference, MIT Press, Stanford University, July 1996.

[Nordin95] P. Nordin and W. Banzhaf, “Complexity Compression and Evolution”, Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers Inc., 1995.

[Nordin96] P. Nordin, F. Francone and W. Banzhaf, “Explictly Defined Introns and Destructive Crossover in Genetic Programming”, Advances in Genetic Programming: Volume 2, P. Angeline and K. Kinnear, MIT Press, Cambridge, 1996, pp. 111-134.

[Ouss96] M. Oussaidène, B. Chopard, O. Pictet and M. Tommasini, “Parallel Genetic Programming: an Application to Trading Model Evolution”, Genetic Programming 1996, Proceedings of the first annual conference, MIT Press, Stanford University, July 1996.

[Ouss97] M. Oussaidène, B. Chopard, O. Pictet and M. Tommasini, “Parallel Genetic Programming and its Application to Trading Model Induction”, Parallel Computing , vol. 23, n. 2, September 1997.

[Pagan] F. Pagan, “Formal Specification of Programming Languages”, Prentice-Hall, 1981.

[Pettey] C. C. Pettey, “Diffusion (cellular) models”, Handbook of Evolutionary Computation, Institute of Phisics Publishing and Oxford University Press, 1997, C6.4.

[Price] G. R. Price, “Selection and Covariance”, Nature, vol. 227, 1970.

[Punch] W. F. Punch, “How Effective are Multiple Populations in Genetic Programming”, Genetic Programming 1998, Proceedings of the Third Annual Conference, MIT Press, University of Winsconsin, July 1998.

[Radcliffe] N. J. Radcliffe, “Equivalence class analysys of genetic algorithms”, Complex Systems, vol.5, n. 2, 1991.

[Rich] E. Rich, “Artificial Intelligence”, McGraw-Hill, New York, 1983.

[Salhi] A. Salhi, H. Glaser and D. De Roure, “Parallel Implementation of a Genetic-Programming based Tool for Symbolic Regression”, Technical Report DSSE-TR-97-3, Dept. Comp. Science, University of Southampton, 1997.

[Snipper] M.Snipper, “The Emergence of Cellular Computing”, Computer, IEEE Computer Society, July 1999.

[Stoffel] K. Stoffel and L. Spector, “High-Performance Stack-Based Genetic Programming”, Genetic Programming 1996, Proceedings of the First Annual Conference, MIT Press, Stanford University, July 1996.

[Tackett94] W.A. Tackett, “Recombination, Selection and the Genetic Construction of Computer Programs”, Ph.D. thesis, University of Southern California, Department of Electrical Engineering Systems, 1994.