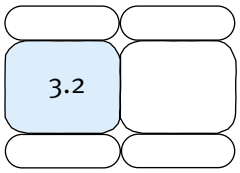
ALGORITMI EVOLUTIVI: CONCETTI E APPLICAZIONI

Andrea G. B. Tettamanzi

Gli algoritmi evolutivi sono una famiglia di tecniche stocastiche per la risoluzione di problemi che fa parte della più ampia categoria dei "modelli a metafora naturale". Essi trovano la loro ispirazione in biologia e, in particolare, si basano sull'imitazione dei meccanismi della cosiddetta "evoluzione naturale". Nel corso degli ultimi 25 anni queste tecniche sono state applicate ad un elevato numero di problemi di grande rilevanza pratica ed economica. L'articolo presenta una rassegna di queste tecnologie e alcuni esempi di applicazione.



1. CHE COSA SONO GLI ALGORITMI EVOLUTIVI?

Guardando agli esseri viventi, tra cui l'Uomo e ad alcuni dettagli dei loro organi, alla loro complessità e perfezione, viene quasi da chiedersi come sia possibile che soluzioni tanto sofisticate possano essersi evolute autonomamente. Eppure esiste una teoria, proposta inizialmente da Charles Darwin e perfezionata in seguito da numerosi altri naturalisti, biologi e genetisti, che è in grado di spiegare in modo soddisfacente la maggior parte di tali fenomeni biologici, partendo dallo studio dei meccanismi di adattamento delle specie ad ambienti mutevoli e complessi. Questa teoria è supportata da importanti evidenze e non è stata ancora falsificata da alcun dato sperimentale. Secondo la teoria darwiniana, tali prodotti mirabili altro non sarebbero che il risultato di un processo evolutivo che procede senza uno scopo, guidato da una parte da una componente casuale e, dall'altra, dalla legge della sopravvivenza del più adatto: l'evoluzione naturale.
Se un processo simile è stato capace di produrre degli artefatti tanto sofisticati come l'occhio, il sistema immunitario e il nostro stesso cervello, pare soltanto logico cercare di imitarlo, simulandolo con gli elaboratori elettronici di cui disponiamo, al fine di risolvere i problemi più complicati che la realtà ci pone. È questa l'idea che sta alla base dello sviluppo degli algoritmi evolutivi.

1.1. La metafora di fondo

Gli algoritmi evolutivi sono dunque delle tecniche informatiche ispirate dalla biologia che si basano su una metafora, illustrata schematicamente nella tabella 1: come un individuo di una popolazione di organismi deve essere adattato all'ambiente che lo circonda per sopravvivere e riprodursi, così una possibile soluzione deve essere adatta a risolvere il suo problema. Il problema è l'ambiente in cui una soluzione vive, all'interno di una popolazione di altre possibili soluzioni; le soluzioni differiscono tra loro per qualità, cioè per costo o merito, che si riflettono nella valutazione della funzione obiettivo, così come gli individui di una popolazione di organismi differiscono

TABELLA 1

Illustrazione schematica della metafora su cui si basano gli algoritmi evolutivi

Evoluzione	Problem Solving
Ambiente	Problema da risolvere
Individuo	Possibile soluzione
Adattamento	Qualità della soluzione

Un po' di storia

L'idea di usare selezione e mutazione casuale per un compito di ottimizzazione risale almeno agli anni cinquanta, con il lavoro dello statistico George E. P. Box, noto per la massima "tutti i modelli sono sbagliati, ma alcuni sono utili", che tuttavia non fece uso dell'elaboratore elettronico. Box giunse a formulare una metodologia statistica che sarebbe divenuta di largo uso nell'industria e che egli battezzò *evolutionary operation* [1]. Più o meno negli stessi anni, altri studiosi concepirono l'idea di simulare l'evoluzione sull'elaboratore elettronico: Barricelli e Fraser utilizzarono simulazioni al calcolatore per studiare i meccanismi dell'evoluzione naturale, mentre al biomatematico Hans J. Bremermann va dato il credito di avere per primo riconosciuto nell'evoluzione biologica un processo di ottimizzazione [2].

Come spesso accade per molte idee pionieristiche, questi primi sforzi incontrarono uno scetticismo considerevole. Ciononostante, i tempi evidentemente erano maturi perché queste idee, che ormai erano nell'aria, venissero sviluppate. Probabilmente un fattore determinante per cui ciò avvenne fu l'aumento, oltre una certa soglia critica, della potenza computazionale degli elaboratori elettronici, allora disponibili nelle migliori università, che rese finalmente possibile la messa in pratica del calcolo evoluzionistico. Gli algoritmi evolutivi, in quelle che oggi riconosciamo come le loro varianti originarie, furono inventati indipendentemente e praticamente allo stesso tempo, a metà degli anni Sessanta, nel seno di tre distinti gruppi di ricerca: in America, Lawrence Fogel e colleghi, dell'Università di California a San Diego, posero le basi della programmazione evolutiva (*evolutionary programming*) [3], mentre presso l'Università del Michigan ad Ann Arbor John Holland proponeva i primi algoritmi genetici (*genetic algorithms*) [4]; in Europa, invece, furono Ingo Rechenberg e colleghi, allora studenti presso il Politecnico di Berlino, a ideare quelle che battezzarono "strategie evolutive" (*Evolutionsstrategien*) [5]. Per i successivi 25 anni questi tre filoni si svilupparono essenzialmente ciascuno per conto suo, finché nel 1990 non venne messo in atto uno sforzo organizzato per farli convergere: la prima edizione del congresso PPSN (*Parallel Problem Solving from Nature*), che si tenne quell'anno a Dortmund. Da allora i ricercatori interessati al calcolo evoluzionistico (*evolutionary computation*) formano un'unica, anche se articolata, comunità scientifica.

tra di loro per grado di adattamento all'ambiente, chiamato dai biologi *fitness*. Se la selezione naturale permette a una popolazione di organismi di adattarsi all'ambiente che la circonda, sarà anche in grado, applicata a una popolazione di soluzioni a un problema, di far evolvere soluzioni sempre migliori ed eventualmente, con il tempo, ottime.

In base a questa metafora, il modello computazionale prende in prestito dalla biologia alcuni concetti e i relativi termini: ogni soluzione è codificata in uno o più *cromosomi*; i *geni* sono i pezzi della codifica responsabili di uno o più *tratti* di una soluzione; gli *alleli* sono le possibili configurazioni che un gene può assumere; lo scambio di materiale genetico tra due cromosomi si chiama *crossover*, mentre ci si riferisce alla perturbazione della codifica di una soluzione con il termine *mutazione*.

Sebbene il modello computazionale introduca delle semplificazioni drastiche rispetto al mondo naturale, gli algoritmi evolutivi si sono rivelati capaci di far emergere strutture sorprendentemente complesse e interessanti. Ogni individuo può essere la rappresentazione, secondo un'opportuna codifica, di una particolare soluzione di un problema, di una strategia per affrontare un gioco, di un piano, di un'immagine o addirittura di un semplice programma per calcolatore.

La **storia** e il **funzionamento** degli algoritmi genetici sono riassunti nei due riquadri.

1.2. Gli ingredienti di un algoritmo evolutivo

Fatta questa premessa di tipo concettuale, vediamo ora in che cosa consiste, praticamente, un algoritmo evolutivo.

Un algoritmo evolutivo è una tecnica stocastica di ottimizzazione che procede in modo iterativo, mantenendo una popolazione (che in questo contesto significa un multiinsieme, ovvero una collezione di elementi non necessariamente tutti distinti tra loro) di individui che rappresentano possibili soluzioni per il problema che deve essere risolto (il problema oggetto) e facendola evolvere mediante l'applicazione di un certo numero, di solito abbastanza ridotto, di operatori stocastici: *mutazione, ricombinazione e selezione*.

La mutazione può essere qualsiasi operatore che perturbi casualmente una soluzione; gli operatori di ricombinazione decompongono due o più individui distinti e quindi mescolano le loro parti costitutive per formare un certo numero di nuovi individui; la selezione crea delle repliche degli individui che rappresentano le soluzioni migliori all'interno della popolazione ad un tasso proporzionale alla loro *fitness*.

La popolazione iniziale può provenire da un campionamento casuale dello spazio delle soluzioni oppure da un nucleo di soluzioni iniziali trovate da semplici procedure di ricer-

Come funziona un algoritmo genetico

Proviamo ad osservare da vicino il funzionamento di un algoritmo genetico servendoci di un esempio. Supporremo di dover risolvere un problema, che chiameremo *maxuno*, il quale consiste nel cercare, tra tutte le stringhe binarie di lunghezza *l*, quella che contiene il numero massimo di "1". A prima vista questo potrebbe sembrare un problema banale, per il semplice motivo che conosciamo in anticipo la soluzione: la stringa di tutti "1". Tuttavia, se si immaginasse di dover compiere *l* scelte binarie per risolvere un problema e che la qualità della soluzione fosse proporzionale al numero di scelte corrette effettuate, ecco che avremmo un problema di difficoltà equivalente e per nulla facile. Il fatto di supporre che le scelte corrette corrispondano tutte a un "1" è solo un artificio per rendere l'esempio più facile da seguire. Definiamo dunque la *fitness* di una soluzione come il numero di "1" presenti nella sua codifica binaria, fissiamo $l = 10$, che è un numero abbastanza piccolo da essere gestibile, e proviamo ad applicare a questo problema l'algoritmo genetico.

Per prima cosa dobbiamo stabilire la dimensione della popolazione: una buona scelta, tanto per cominciare, potrebbe essere 6 individui. A questo punto, è necessario generare una popolazione iniziale: lo faremo lanciando 60 volte (6 individui per 10 cifre binarie) una monetina non truccata e scrivendo 0 se esce "testa" e 1 se esce "croce". La popolazione iniziale così ottenuta è quella della tabella A. Notiamo che la media della *fitness* nella popolazione iniziale è di 5,67.

Numero	Individuo	Fitness
1	1111010101	7
2	0111000101	5
3	1110110101	7
4	0100010011	4
5	1110111101	8
6	0100110000	3

TABELLA A *La popolazione iniziale dell'algoritmo genetico per risolvere il problema maxuno, con la fitness corrispondente a ciascun individuo*

Il ciclo evolutivo ora può cominciare: per applicare la selezione proporzionale alla *fitness* il metodo più semplice consiste nel simulare il lancio di una pallina in una *roulette* speciale, con tanti settori quanti sono gli individui della popolazione (in questo caso 6) , ciascuno avente un'ampiezza che sta alla circonferenza quanto la *fitness* dell'individuo corrispondente sta alla somma delle *fitness* di tutta la popolazione (in questo caso 34). Perciò, quando lanceremo la pallina, questa avrà una probabilità di 7/34 di fermarsi nel settore dell'individuo 1, 5/34 di fermarsi nel settore dell'individuo 2, e così via. Dovremo compiere esattamente 6 lanci per formare una popolazione intermedia di 6 stringhe per la riproduzione. Supponiamo che i lanci diano il seguente esito: 1, 3, 5, 2, 4 e ancora 5. Significa che verranno usate per la riproduzione due copie dell'individuo 5 e una copia degli altri individui ad eccezione dell'individuo 6, che non lascerà discendenti. Il successivo operatore ad essere applicato è la ricombinazione: si formano le coppie, il primo estratto con il secondo, il terzo con il quarto, e così via; per ciascuna delle coppie, decidiamo con una certa probabilità, diciamo 0,6, se effettuare il *crossover*. Supponiamo che il *crossover* venga effettuato solo sulla prima e sull'ultima coppia, con punti di taglio scelti a casi rispettivamente dopo la seconda cifra e dopo la quinta.

Per la prima coppia, avremo:

11.11010101 che diventa 11.10110101
11.10110101 " 11.11010101

Notiamo che, siccome le parti a sinistra del punto di taglio sono identiche, il *crossover* non ha alcun effetto. L'eventualità è più comune di quanto si possa immaginare, specialmente quando, avanti con le generazioni, la popolazione sia piena di individui tutti buoni e quasi identici tra di loro.

Invece, per la terza coppia, avremo:

01000.10011 che diventa 01000.11101
11101.11101 " 11101.10011

Non resta che applicare la mutazione alle sei stringhe risultanti dalla ricombinazione, decidendo per esempio con probabilità di 1/10 per ogni cifra binaria se invertirla. Avendo in tutto 60 cifre binarie, ci aspetteremo in media 6 mutazioni distribuite casualmente in tutta la popolazione. La nuova popolazione, dopo l'applicazione di tutti gli operatori genetici, potrebbe essere quella mostrata nella tabella B, dove le cifre binarie mutate sono state evidenziate in grassetto.

Numero	Individuo	Fitness
1	11101**0**0101	6
2	1111**1**1010**0**	7
3	11101**0**111**1**	8
4	0111000101	5
5	0100011101	5
6	11011000**0**1	6

TABELLA B *La popolazione dell'algoritmo genetico per risolvere il problema maxuno dopo una generazione, con la fitness corrispondente a ciascun individuo*

In una generazione, la *fitness* media della popolazione è passata da 5,67 a 6,17, con un incremento dell'8,8%. Iterando lo stesso processo più volte, si arriva ben presto a un punto in cui fa la sua comparsa un individuo di tutti "1", la soluzione ottima del problema.

ca locale, se disponibili, o determinate da un esperto umano.
Gli operatori stocastici, applicati e composti secondo le regole che definiscono il particolare algoritmo evolutivo, determinano un operatore stocastico di trasformazione di popolazioni, in base al quale è possibile modellare il funzionamento di un algoritmo evolutivo come una catena di Markov i cui stati sono le popolazioni. È possibile dimostrare che, se sono soddisfatte alcune ipotesi tutto sommato ragionevoli, tale processo stocastico converge in probabilità all'ottimo globale del problema [16].
Si parla spesso, con riferimento agli algoritmi evolutivi, di *parallelismo implicito*. Questo termine si riferisce al fatto che ciascun individuo può essere pensato come un rappresentante di una moltitudine di *schemi* di soluzione, cioè di soluzioni parzialmente specificate, di modo che, elaborando un singolo individuo, l'algoritmo evolutivo starebbe in realtà elaborando implicitamente allo stesso tempo (cioè in parallelo) tutti gli schemi di soluzione di cui quell'individuo è un rappresentante. Non bisogna confondere questo concetto con il *parallelismo inerente* degli algoritmi evolutivi, derivante dal fatto che essi conducono una ricerca basata su una popolazione, il quale fa sì che, sebbene il loro funzionamento sia esprimibile per comodità mediante una descrizione sequenziale, la loro realizzazione su hardware parallelo risulti particolarmente naturale e vantaggiosa.

1.3. Algoritmi genetici

Il modo migliore per capire come funzionano gli algoritmi evolutivi consiste nel considerare una delle loro versioni più semplici: gli algoritmi genetici [6]. Negli algoritmi genetici, le soluzioni sono rappresentate come stringhe binarie di lunghezza fissa. Questo tipo di rappresentazione è sicuramente il più generale, tuttavia, come vedremo più avanti, non sempre è anche il più conveniente: infatti, qualsiasi struttura dati, per quanto complessa e articolata, sarà sempre codificata in alfabeto binario nella memoria dell'elaboratore elettronico. Ora, una sequenza di due simboli, 0 e 1, da cui è possibile ricostruire una soluzione, ricorda moltissimo un filamento di DNA, costituito da una sequenza di quattro basi, A, C, G e T, da cui è possibile ricostruire un organismo vivente! In altre parole, possiamo considerare una stringa binaria come il DNA di una soluzione del problema oggetto.
Un algoritmo genetico è composto di due parti:
1. una procedura che genera (casualmente o utilizzando qualche euristica) la popolazione iniziale;
2. un ciclo evolutivo, che, ad ogni iterazione (o *generazione*), crea una nuova popolazione applicando gli operatori genetici alla popolazione precedente.

Il ciclo evolutivo degli algoritmi genetici può essere schematizzato mediante lo pseudocodice di tabella 2. A ciascun individuo viene assegnato un particolare valore di *fitness*, che dipende dalla qualità della soluzione che esso rappresenta. Il primo operatore ad essere applicato è la selezione, il cui scopo è simulare la legge darwiniana della sopravvivenza del più adatto. Nella versione originale degli algoritmi genetici, questa legge è implementata per mezzo della cosiddetta selezione proporzionale alla *fitness*: per creare una nuova popolazione intermedia di *n* individui "genitori", vengono effettuate *n* estrazioni indipendenti di un individuo dalla popolazione esistente, con probabilità per ogni individuo di essere estratto direttamente proporzionale alla sua *fitness*. Di conseguenza, gli individui al di sopra della media verranno in media estratti più volte, mentre quelli al di sotto della media andranno incontro all'estinzione.
Una volta estratti gli *n* genitori come descritto, gli individui della generazione successiva

```
generation = 0
Inizializza popolazione
while not <condizione di terminazione>
do
        generation = generation + 1
        Calcola fitness di ciascun individuo
        Selezione
        Crossover(p_cross)
        Mutazione(p_mut)
end while
```

TABELLA 2
Pseudocodice che illustra un tipico algoritmo genetico semplice

saranno prodotti mediante l'applicazione di un certo numero di operatori di riproduzione, i quali possono coinvolgere un solo genitore (simulando quindi una sorta di riproduzione asessuata), nel qual caso si parla di *mutazione*, o più di un genitore, normalmente due (riproduzione sessuata), nel qual caso si parla di *ricombinazione*. Negli algoritmi genetici sono utilizzati due operatori di riproduzione: crossover e mutazione.

Per applicare il *crossover*, gli individui genitori vengono accoppiati a due a due; quindi, con una certa probabilità p_{cross}, chiamata "tasso di *crossover*", che è un parametro dell'algoritmo, ciascuna coppia subisce il *crossover* vero e proprio, che consiste nell'allineare le due stringhe binarie, tagliarle in un punto estratto a caso, e scambiarne le metà destre, ottenendo così due nuovi individui, che ereditano parte dei loro caratteri da un genitore e parte dall'altro.

Dopo il *crossover*, tutti gli individui subiscono la mutazione, il cui scopo è quello di simulare l'effetto di errori casuali di trascrizione che possono avvenire con una probabilità molto bassa p_{mut} ogniqualvolta un cromosoma venga duplicato, e che consiste nel decidere di invertire ciascuna singola cifra binaria, indipendentemente dalle altre, con probabilità p_{mut}. In altre parole, ciascuno zero ha una probabilità p_{mut} di diventare un uno e viceversa.

Il ciclo evolutivo, per come è concepito, potrebbe andare avanti all'infinito. Nella pratica, però, bisogna decidere quando arrestarlo, in base a qualche criterio di terminazione specificato dall'utente. Alcuni esempi di criteri di terminazione possono essere:

- il passaggio di un numero prefissato di generazioni o di una certa quantità di tempo;
- il rinvenimento di una soluzione soddisfacente secondo qualche misura;
- la mancanza di miglioramenti per un certo numero prefissato di generazioni.

1.4. Strategie evolutive

Le strategie evolutive affrontano l'ottimizzazione di una funzione obiettivo reale di variabili reali in uno spazio a l dimensioni. La rappresentazione utilizzata per le variabili indipendenti della funzione (la soluzione) è quella più diretta, cioè un vettore di numeri reali. Oltre a codificare le variabili indipendenti, tuttavia, le strategie evolutive includono nell'individuo anche delle informazioni sulla distribuzione di probabilità da utilizzare per la loro perturbazione (operatore di mutazione): a seconda delle versioni, queste informazioni possono andare dalla semplice varianza, valida per tutte le variabili indipendenti, all'intera matrice di varianza e covarianza $\boldsymbol{C}$ di una distribuzione normale congiunta; in altre parole, la dimensione di un individuo può andare da $l + 1$ a $l\,(l + 1)$ numeri reali.

Nella sua forma più generale, l'operatore di mutazione perturba un individuo in due passi:

1. perturba la matrice $\boldsymbol{C}$ (o, meglio, una matrice equivalente di angoli di rotazione da cui la matrice $\boldsymbol{C}$ può essere agevolmente calcolata) con una distribuzione di probabilità identica per tutti gli individui;
2. perturba il vettore dei parametri che rappresenta la soluzione al problema di ottimizzazione con una probabilità normale congiunta con media $\mathbf{0}$ e matrice di varianza e covarianza la $\boldsymbol{C}$ perturbata.

Questo meccanismo di mutazione permette all'algoritmo di far evolvere autonomamente i parametri della sua strategia di ricerca mentre va in cerca della soluzione ottima: il processo che ne deriva, denominato *autoadattamento*, è uno degli aspetti più potenti e interessanti di questo tipo di algoritmo evolutivo.

La ricombinazione nelle strategie evolutive può assumere diverse forme: quelle utilizzate più di frequente sono la ricombinazione *discreta* e quella *intermedia*. Nella ricombinazione discreta, ciascuna componente dell'individuo figlio è presa da uno dei genitori a caso; nella ricombinazione intermedia, invece, ciascuna componente è ottenuta mediante combinazione lineare, con un parametro casuale, delle componenti corrispondenti dei genitori.

Esistono due schemi di selezione alternativi, che definiscono due classi di strategie evolutive: (n, m) e $(n + m)$. Nelle strategie (n, m), da una popolazione di n individui vengono prodotti $m > n$ individui figli e gli n migliori vanno a costituire la popolazione della generazione successiva. Nelle strategie $(n + m)$, invece, anche gli n individui genitori partecipano alla selezione e, di questi $n + m$ individui, soltanto i migliori n entrano a far parte della popolazione della generazione successiva. Si noti che,

in entrambi i casi, la selezione è deterministica e funziona "per troncamento", cioè scartando gli individui peggiori: in questo modo, non è necessario definire una *fitness* non negativa per gli individui e l'ottimizzazione considera direttamente la funzione obiettivo, che può essere, a seconda dei casi, massimizzata o minimizzata.

1.5. Programmazione evolutiva

L'evoluzione, naturale o artificiale, in sé non ha nulla di "intelligente" nel senso letterale del termine: infatti non capisce quello che sta facendo, né deve capirlo. L'intelligenza, invece, ammesso che si possa definire, può essere un fenomeno "emergente" dell'evoluzione, nel senso che l'evoluzione può giungere a produrre organismi o soluzioni dotati di una qualche forma di "intelligenza".
La programmazione evolutiva nasce come approccio all'intelligenza artificiale, alternativo rispetto alle tecniche basate sul ragionamento simbolico. Il suo scopo è di far evolvere, piuttosto che definire a priori, comportamenti intelligenti, rappresentati per mezzo di automi a stati finiti. Nella programmazione evolutiva, quindi, il problema oggetto determina l'alfabeto di ingresso e di uscita di una famiglia di automi a stati finiti, e gli individui sono opportune rappresentazioni di automi a stati finiti che operano su tali alfabeti. La rappresentazione naturale di un automa a stati finiti consiste nella matrice che definisce le due funzioni di transizione di stato e di uscita. La definizione degli operatori di mutazione e ricombinazione è leggermente più complessa che nel caso degli algoritmi genetici o delle strategie evolutive, in quanto deve tenere conto della struttura degli oggetti che tali operatori devono manipolare. La *fitness* di un individuo può essere calcolata mettendo alla prova su un insieme di casi del problema l'automa a stati finiti che esso rappresenta: per esempio, se si desidera far evolvere individui capaci di modellare una serie storica, si selezioneranno un certo numero di pezzi della serie passata, li si daranno in ingresso a un individuo e se ne osserveranno i simboli prodotti, interpretandoli come previsioni e confrontandoli con i dati effettivi per misurarne l'accuratezza.

1.6. Programmazione genetica

La programmazione genetica [7] è una branca degli algoritmi evolutivi relativamente nuova, che si pone come obiettivo un vecchio sogno dell'intelligenza artificiale: la programmazione automatica. In un problema di programmazione, una soluzione è rappresentata da un programma in un dato linguaggio di programmazione. Nella programmazione genetica, quindi, gli individui rappresentano programmi.
Qualsiasi linguaggio di programmazione, almeno in linea di principio, potrebbe essere adottato; tuttavia, la sintassi della maggior parte dei linguaggi renderebbe la definizione di operatori genetici che la rispettassero particolarmente goffa ed onerosa, ragione per cui i primi sforzi in questa direzione trovarono in una sorta di LISP ristretto un mezzo ideale di espressione. Il LISP ha il pregio di possedere una sintassi particolarmente semplice, oltre a permettere di manipolare dati e programmi in modo uniforme. In pratica, per ogni problema di programmazione che si voglia risolvere, si stabilisce un insieme di variabili, costanti e funzioni adatto a risolverlo, limitando così lo spazio di ricerca che altrimenti sarebbe spropositato. Le funzioni scelte saranno quelle che *a priori* si riterranno utili ai bisogni; inoltre, di solito, si cerca di fare in modo che ciascuna funzione accetti come argomenti i risultati prodotti da ognuna delle altre, così come ogni variabile e costante predefinita. Di conseguenza, lo spazio dei possibili programmi, all'interno del quale si cerca quello che risolve il problema, sarà costituito da tutte le possibili composizioni di funzioni che possano essere formate ricorsivamente a partire dall'insieme delle funzioni, delle variabili e delle costanti predefinite.
Per semplicità, e senza perdita di generalità, si possono considerare gli individui della programmazione genetica come alberi di *parsing* di altrettanti programmi, come illustrato nella figura 1. La ricombinazione di due programmi, in questo contesto, consiste nell'estrarre a caso un nodo dell'albero di ciascuno dei due genitori e nello scambiare i sottoalberi che hanno tali nodi come radici, come illustrato schematicamente nella figura 2. L'operatore di muta-

zione riveste un'importanza limitata nella programmazione genetica, in quanto la ricombinazione da sola è in grado di generare una diversità sufficiente a far procedere l'evoluzione.
Il calcolo della *fitness* di un individuo procede in un modo non troppo dissimile dal *testing* di un programma: deve essere dato, come parte integrante della descrizione del problema da risolvere, un insieme di casi di *test*, cioè di coppie (dati di ingresso, risultato corretto), che verranno usati per testare i programmi generati dall'algoritmo in questo modo: per ciascun caso, il programma viene eseguito sui dati di ingresso; il risultato ottenuto viene confrontato con quello corretto, e l'errore misurato; infine, la *fitness* è ottenuta in funzione dell'errore totale accumulato.
Un approccio ancora più recente alla programmazione genetica è costituito dalla cosiddetta *evoluzione grammaticale* [8], la cui idea di fondo è semplice ma potente: data la grammatica di un linguaggio di programmazione (che questa volta è completamente arbitrario, senza limitazioni derivanti dalla particolare sintassi), costituita da un certo numero di regole di produzione, un programma in questo linguaggio è rappresentato da una stringa di cifre binarie. La rappresentazione viene decodificata partendo dal simbolo non terminale obiettivo della grammatica e leggendo le cifre binarie da sinistra a destra, ogni volta in numero sufficiente a scegliere quale delle regole di produzione applicabili debba essere effettivamente applicata; si considera la stringa come se fosse circolare, in modo che il processo di decodifica non si trovi mai a corto di cifre binarie; il processo termina quando nessuna regola di produzione è più applicabile e si è ottenuto quindi un programma ben formato, che può essere compilato ed eseguito in ambiente controllato.

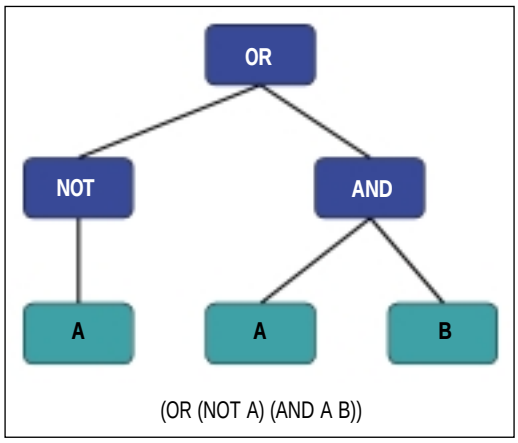


FIGURA 1
Un esempio di programma LISP con il suo albero di parsing associato

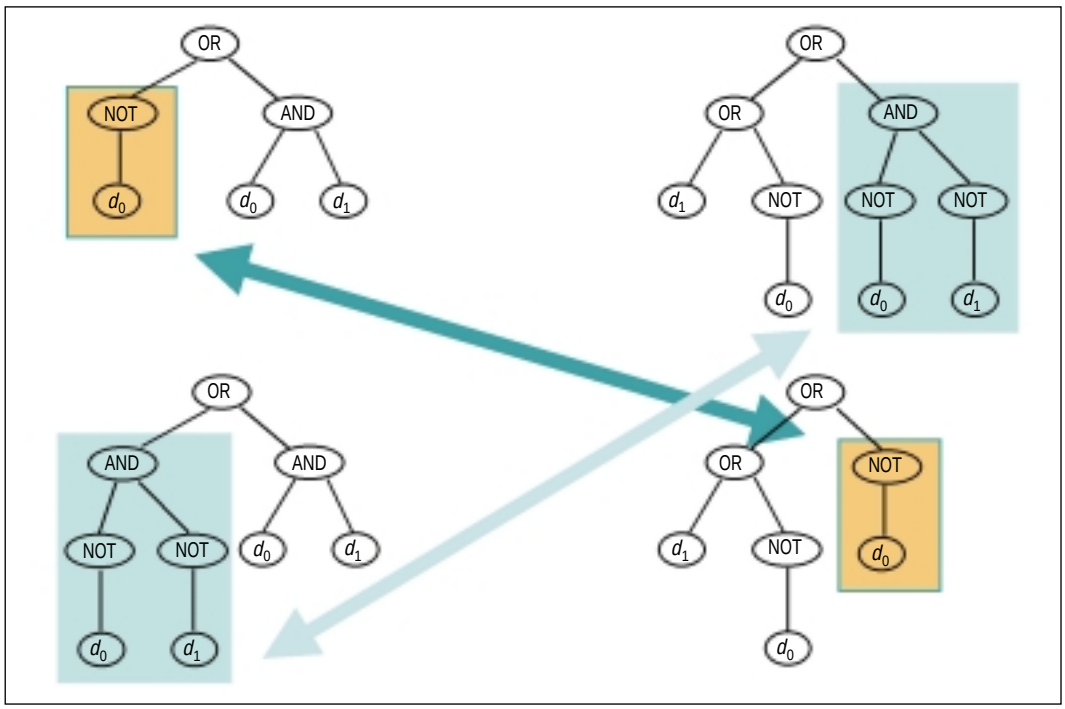


FIGURA 2
Illustrazione schematica della ricombinazione nella programmazione genetica

2. ALGORITMI EVOLUTIVI "MODERNI"

Dagli inizi degli anni ‘80 ad oggi gli algoritmi evolutivi sono stati applicati con successo a molti problemi del mondo reale, studiati dalla ricerca operativa e difficili o impossibili da trattare con metodi esatti e si sono guadagnati un posto di tutto rispetto nella cassetta degli attrezzi del risolutore di problemi. Chiaramente, questo quarto di secolo ha assistito a una maturazione delle varie tecniche evolutive e a una loro fertilizzazione incrociata, oltre che a una progressiva ibridazione con altre metodologie.

Se dovessimo identificare una linea di tendenza principale in questo processo di sviluppo, potremmo senza dubbio riconoscere un progressivo distacco dalle rappresentazioni eleganti, a stringhe binarie, dei primi algoritmi genetici, così suggestivamente vicine alla fonte d’ispirazione biologica e una maggiore propensione ad adottare rappresentazioni più vicine alla natura del problema oggetto, che si mappino in modo più diretto sugli elementi costitutivi delle soluzioni, permettendo così di sfruttare tutte le informazioni a disposizione per “aiutare”, per così dire, il processo evolutivo a trovare la strada migliore [9].

Adottare rappresentazioni più vicine al problema significa anche necessariamente progettare operatori di mutazione e ricombinazione che manipolano gli elementi di una soluzione in modo esplicito, in modo informato; da un lato questi operatori finiscono per essere meno generali, ma dall’altro i vantaggi in termini di prestazioni sono spesso notevoli e ripagano del maggior sforzo di progettazione.

Chiaramente, la richiesta di soluzioni efficienti fa perdere di vista l’unitarietà del modello genetico.

2.1. Gestione dei vincoli

I problemi del mondo reale, quelli cioè che si incontrano nell’industria, nel commercio, nella finanza e nella pubblica amministrazione, la cui risoluzione ha spesso un impatto economico rilevante e che formano l’obiettivo principale della ricerca operativa, hanno tutti in comune la caratteristica di avere vincoli complessi e difficili da trattare.

Nei primi lavori sul calcolo evoluzionistico, non era molto chiaro come affrontare nella maniera ottimale la problematca della gestione dei vincoli. Con il tempo, gli algoritmi evolutivi da una parte hanno cominciato ad essere apprezzati come metodi approssimati per la ricerca operativa e, dall’altra, hanno potuto beneficiare di tecniche ed accorgimenti messi a punto nell’ambito della ricerca operativa per altri metodi approssimati. Da questa fertilizzazione incrociata sono emerse tre tecniche principali che all'occorrenza possono essere combinate tra di loro, per tenere conto di vincoli non banali in un algoritmo evolutivo:

- l’uso di funzioni di penalizzazione;
- l’uso di algoritmi di decodifica o di riparazione;
- la progettazione di codifiche e operatori genetici specializzati.

Le funzioni di penalizzazione sono associate a ciascun vincolo del problema e misurano il grado di violazione del loro vincolo da parte di una soluzione. Come suggerisce il loro nome, queste funzioni vengono combinate con la funzione obiettivo in modo da penalizzare la *fitness* degli individui che non rispettano qualche vincolo. Sebbene le funzioni di penalizzazione costituiscano un approccio molto generale e facile da applicare a qualsiasi problema, il loro utilizzo non è privo di insidie: se le funzioni di penalizzazione non sono accuratamente pesate, l’algoritmo potrebbe sprecare la maggior parte del suo tempo elaborando soluzioni non ammissibili, o addirittura finire per convergere su un ottimo apparente che in realtà non è realizzabile. Per esempio, in un problema di trasporto, in cui sono dati *n* stabilimenti di produzione ed *m* acquirenti a cui una data quantità di merce deve essere consegnata, ed è noto quanto costi trasportare un’unità di merce tra ciascuno stabilimento e ciascun acquirente, una soluzione che minimizza il costo complessivo in modo imbattibile è quella che non trasporta assolutamente nulla! Se la violazione dei vincoli che impongono che ad ogni acquirente sia consegnato il quantitativo di merce che ha ordinato non viene penalizzata a sufficienza, la soluzione assurda di non consegnare alcuna merce potrebbe risultare migliore di tutte quelle che soddisfano gli ordini degli acquirenti. In alcuni problemi, chiamati *problemi di ammissibilità*,

trovare una soluzione che non viola alcun vincolo è tanto difficile quanto trovare la soluzione ottima, o poco meno: in questo genere di problemi, le funzioni di penalizzazione devono essere progettate con cura, altrimenti l'evoluzione non riuscirà mai a trovare alcuna soluzione ammissibile.
Gli algoritmi di decodifica, o decodificatori, sono algoritmi di ottimizzazione basati su un'euristica parametrizzabile che cercano di costruire una soluzione ottima partendo da zero e compiendo un certo numero di scelte. L'idea allora, qualora uno di questi algoritmi sia disponibile, è quella di codificare negli individui trattati dall'algoritmo genetico i parametri dell'euristica, piuttosto che direttamente la soluzione, e utilizzare il decodificatore per ricavare dai parametri la soluzione corrispondente. Si ha cioè quella che potremmo chiamare una rappresentazione *indiretta* delle soluzioni.
Gli algoritmi di riparazione sono operatori che, in base a qualche euristica, prendono una soluzione non ammissibile e la "riparano" forzando il rispetto prima di un vincolo violato, poi di un altro, fino ad ottenere una soluzione ammissibile. Applicati al risultato degli operatori genetici di mutazione e ricombinazione, questi riparatori possono garantire che l'algoritmo evolutivo elabori in ogni momento solamente soluzioni ammissibili. Tuttavia, l'applicabilità di questa tecnica è limitata, in quanto per molti problemi la complessità computazionale del riparatore è tale da vanificare i vantaggi derivanti da una sua eventuale adozione.
La progettazione di codifiche e operatori genetici specializzati sarebbe la tecnica ideale, ma anche quella più complicata da applicare in tutti i casi. L'idea di fondo consiste nel cercare di progettare una rappresentazione delle soluzioni che, per costruzione, sia in grado di codificare solo tutte le soluzioni ammissibili e operatori specifici di mutazione e ricombinazione che preservino l'ammissibilità delle soluzioni a cui sono applicati. Come si può immaginare, questo esercizio, al crescere della complessità e del numero dei vincoli, diventa ben presto formidabile ed eventualmente impossibile. Tuttavia, quando questa strada può essere percorsa, è sicuramente quella ottimale, in quanto garantisce che l'algoritmo evolutivo elabori soltanto soluzioni ammissibili e quindi, di fatto, riduce lo spazio di ricerca al minimo indispensabile.

2.2. Combinazione con altre tecniche di soft computing

Gli algoritmi evolutivi fanno parte, insieme alla logica *fuzzy* e alle reti neurali, di quello che potremmo chiamare *soft computing*, per contrapposizione alla computazione tradizionale, *hard*, basata su criteri come la precisione, il determinismo e il contenimento della complessità. Il *soft computing* si distingue dalle tecniche convenzionali (*hard computing*) per il fatto di tollerare l'imprecisione, l'incertezza e le verità parziali. Il suo principio guida è quello di sfruttare questa tolleranza per ottenere trattabilità, robustezza e contenimento delle risorse computazionali necessarie per risolvere i problemi affrontati.
Il *soft computing* non è semplicemente una mistura degli ingredienti che lo compongono, ma una disciplina nella quale ciascuna metodologia completa le altre intervenendo sull'aspetto del problema che meglio le si adatta [10]. Così gli algoritmi evolutivi possono essere impiegati per progettare e ottimizzare sistemi *fuzzy*, come insiemi di regole *fuzzy* o alberi di decisione *fuzzy*, ma anche per migliorare le caratteristiche di apprendimento delle reti neurali, arrivando anche a determinarne la topologia ottimale; d'altro canto, la logica *fuzzy* può essere utilizzata per controllare il processo evolutivo agendo in modo dinamico sui parametri dell'algoritmo, in modo da accelerare la convergenza all'ottimo globale e sfuggire dagli ottimi locali, ma anche per "*fuzzificare*" alcuni elementi dell'algoritmo, come la *fitness* degli individui o la loro codifica, mentre le reti neurali possono essere affiancate a un algoritmo evolutivo per ottenere una stima approssimata della *fitness* degli individui per problemi in cui il calcolo della *fitness* richieda simulazioni molto pesanti dal punto di vista computazionale, riducendo in tal modo il tempo macchina e migliorando le prestazioni.
Le combinazioni degli algoritmi evolutivi con altre tecniche di *soft computing* costituiscono un campo di ricerca affascinante e una

delle grandi promesse di questa gamma di tecniche computazionali.

3. LE APPLICAZIONI

Gli algoritmi evolutivi sono stati applicati con successo a problemi in un grande numero di domini. A puro scopo illustrativo e senza pretendere di proporre una classificazione significativa, potremmo dividere il campo di applicazione di queste tecniche in cinque vasti domini:

- pianificazione, che include tutti quei problemi in cui si richiede di scegliere, tra diversi modi alternativi di impiegare un insieme finito di risorse, quello a minor costo o a più alte prestazioni: fanno parte di questo dominio la pianificazione di rotta di una flotta dei veicoli, il problema del trasporto, la pianificazione della traiettoria di un robot, la pianificazione della produzione di un impianto industriale, la confezione di orari, la determinazione del carico ottimale di un mezzo di trasporto ecc.;
- progettazione, che include tutti quei problemi in cui si richiede di determinare una disposizione ottimale di elementi (componenti elettroniche o meccaniche, elementi architettonici ecc.) al fine di soddisfare una serie di requisiti funzionali, estetici e di robustezza: ricadono in questo dominio, quindi, vari problemi di disegno di circuiti elettronici, di strutture ingegneristiche, di progettazione di sistemi informativi ecc.;
- simulazione e identificazione, che consiste, dato un progetto o un modello di un sistema, nel determinare come tale sistema si comporterà: in alcuni casi ciò deve essere fatto perché non si è sicuri del comportamento del sistema, altre volte il comportamento è noto ma si vuole valutare l'accuratezza di un modello. I sistemi studiati possono essere chimici (determinazione della struttura tridimensionale di una proteina, dell'equilibrio di una reazione chimica), economici (simulazione delle dinamiche della concorrenza in un'economia di mercato), medici, e così via;
- controllo, che include tutti quei problemi in cui è richiesto di stabilire una strategia di controllo per un dato sistema;
- classificazione, modellazione e apprendimento automatico, dove, a partire da un insieme di osservazioni, si richiede di costruire un modello del fenomeno sottostante: a seconda dei casi questo modello può consistere nella semplice determinazione di appartenenza di ciascuna osservazione a una di due o più classi, oppure nella costruzione (o apprendimento automatico) di un modello più o meno complesso, spesso da utilizzare per scopi di previsione. Fa parte di questo dominio anche il *data mining*, che consiste nello scoprire regolarità invisibili "ad occhio nudo" in mezzo a enormi quantità di dati.

Naturalmente i confini tra questi cinque domini applicativi non sono nettamente definiti e i domini stessi possono in alcuni casi sovrapporsi in qualche misura. Tuttavia si noterà che essi comprendono tutta una serie di problemi di grande rilievo economico, oltre che di enorme difficoltà.

Nel seguito, cercheremo di dare un'idea di che cosa significhi applicare algoritmi evolutivi a problemi di rilevanza pratica descrivendo tre esempi di applicazioni in domini molto diversi tra loro: la confezione di orari scolastici, la progettazione di circuiti elettronici e la modellazione del comportamento di un cliente.

3.1. Confezione di orari scolastici

Il problema dell'orario (*timetable problem*) consiste nella pianificazione di un certo numero di incontri (esami, lezioni, partite ecc.) che coinvolgano un gruppo di persone (studenti, insegnanti, giocatori ecc.) per un certo periodo e richiedano un insieme di risorse (aule, laboratori, campi di gioco ecc.) secondo le loro rispettive disponibilità e rispettando tutta una serie di altri vincoli accessori. Questo problema è noto essere NP-completo: questo è il motivo principale per cui non può essere affrontato in modo soddisfacente (dal punto di vista delle prestazioni) con algoritmi di tipo esatto ed è da tempo una palestra per tecniche alternative, come gli algoritmi evolutivi. Il problema di confezionare orari scolastici, in particolare per le scuole medie superiori italiane, molte delle quali distribuite su più sedi, è ulteriormente complicato dalla presenza di vincoli molto stringenti, che lo rendono molto prossimo a un vero e proprio problema di ammissibilità.

Un'istanza di questo problema è costituita dalle seguenti entità e dalle loro relazioni:

- aule, caratterizzate per tipo, capacità e localizzazione;

- materie di insegnamento, identificate dal tipo di aula che richiedono;
- insegnanti, caratterizzati dalle materie che insegnano e dai loro orari di disponibilità;
- classi, cioè gruppi di studenti che seguono lo stesso *curriculum*, assegnate a una data sede, e con un orario di presenza a scuola;
- lezioni, una relazione <*t*, *s*, *c*, *l*>, dove *t* è l'insegnante, *s* è la materia, *c* è la classe e *l* è la durata temporale espressa in *periodi* (per esempio, ore); in certi casi, ad una lezione possono partecipare più di un insegnante e più di una classe, nel qual caso si parlerà di lezioni *raggruppate*.

I vincoli del problema sono molteplici, divisi in vincoli rigidi e vincoli flessibili, ma lo spazio ristretto non ci consente di passarli in rassegna; fortunatamente, chiunque abbia frequentato una scuola media superiore in Italia dovrebbe averne almeno un'idea.
Questo problema è stato affrontato mediante un algoritmo evolutivo che sta alla base di un prodotto commerciale, *EvoSchool* [11]. L'algoritmo adotta una rappresentazione "diretta" delle soluzioni, che consiste in un vettore le cui componenti corrispondono alle lezioni che devono essere programmate, e il valore (intero) di ciascuna componente indica il periodo in cui la lezione corrispondente deve avere inizio. La funzione che associa ad ogni orario la sua *fitness*, uno dei punti critici dell'algoritmo, consiste in pratica solo in una combinazione di funzioni di penalizzazione ed assume la seguente forma:

$$f(\mathbf{x}) = 1/\Sigma_i\, \alpha_i\, h_i + \gamma/\Sigma_j\, \beta_j\, s_j$$

dove h_i è la penalizzazione associata alla violazione dell'*i*-esimo vincolo rigido (*hard*), s_j è la penalizzazione associata alla violazione del *j*-esimo vincolo flessibile (*soft*), mentre le α_i e β_j sono opportuni pesi associati a ciascun vincolo; infine, γ è un indicatore che vale uno quando tutti i vincoli rigidi sono soddisfatti e zero altrimenti. In sostanza, ciò vuol dire che i vincoli flessibili vengono presi in considerazione solo dopo che tutti i vincoli rigidi siano stati soddisfatti.
Tutti gli altri ingredienti dell'algoritmo evolutivo utilizzato sono abbastanza standard, tranne la presenza di due operatori di perturbazione, mutuamente esclusivi, chiamati dall'operatore di mutazione ciascuno con la sua probabilità:

- mutazione intelligente;
- operatore di miglioramento.

La mutazione intelligente, pur mantenendo una natura stocastica, è rivolta ad effettuare modifiche che non diminuiscano la *fitness* dell'orario a cui sono applicate; in particolare, se l'operatore agisce sull'*i*-esima lezione, propagherà la sua azione anche a tutte le altre lezioni che coinvolgono la stessa classe, insegnante o aula. La scelta del "raggio d'azione" di questo operatore è casuale, con una certa distribuzione di probabilità. In pratica, l'effetto di questo operatore è di muovere casualmente alcune lezioni collegate tra loro in modo da ridurre le violazioni di vincoli.
L'operatore di miglioramento, invece, compie una ristrutturazione radicale di un orario, scegliendo a caso una lezione di partenza e concentrandosi sugli orari parziali della classe, dell'insegnante e dell'aula in essa coinvolti. La ristrutturazione consiste nel compattare le lezioni presenti, liberando uno spazio sufficiente per sistemare senza conflitti la lezione selezionata.
L'interazione accuratamente bilanciata di questi due operatori è il segreto dell'efficacia di questo algoritmo evolutivo, che si è dimostrato in grado di generare in poche ore, su PC non particolarmente potenti come quelli in uso nei licei e negli istituti tecnici, orari di alta qualità per complessi scolastici con migliaia di lezioni da programmare in diverse sezioni staccate disperse sul territorio.

3.2. Progettazione di circuiti elettronici digitali

Uno dei problemi che hanno ricevuto considerevole attenzione da parte della comunità internazionale del calcolo evoluzionistico è la progettazione di filtri digitali con risposta all'impulso finita. Questo interesse è giustificato dalla presenza di questo tipo di componente in un gran numero di dispositivi elettronici presenti su altrettanti prodotti di largo consumo, come telefoni cellulari, dispositivi di rete ecc.
Le metodologie tradizionali per la progettazione di circuiti elettronici hanno come criterio principale la minimizzazione dei *transistor* impiegati e dunque del costo di produzione. Tuttavia, un altro criterio molto signifi-

TABELLA 3

Operazioni primitive per la rappresentazione di filtri digitali. Il formato delle primitive è fisso. Gli interi n e m si riferiscono agli ingressi al ciclo t – n e t – m rispettivamente

Operazione	Codice	Operando 1	Operando 2	Descrizione
Ingresso	I	non usato	non usato	Copia l'ingresso
Ritardo	D	*n*	non usato	Ritardo di *n* cicli
Scorrimento a sx	L	*n*	*p*	moltiplica per 2^p
Scorrimento a dx	R	*n*	*p*	divide per 2^p
Sommatore	A	*n*	*m*	somma
Sottrattore	S	*n*	*m*	differenza
Complemento	C	*n*	non usato	inverte l'ingresso

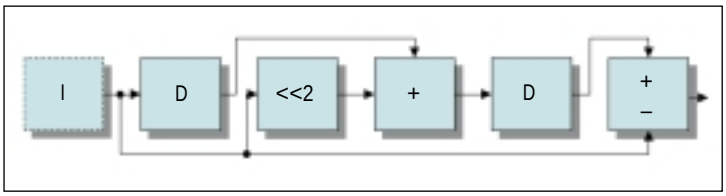


FIGURA 3

Diagramma schematico di un esempio di circuito ottenuto per composizione di 6 operazioni primitive

cativo è quello dell'assorbimento di potenza, che è funzione del numero di transizioni logiche subite dai nodi del circuito. La progettazione di filtri digitali ad assorbimento minimo di potenza è stata affrontata con successo mediante un algoritmo evolutivo [12].
Un filtro digitale può essere rappresentato come composizione di un numero molto ridotto di operazioni elementari, come le primitive elencate in tabella 3. Ciascuna operazione primitiva è codificata per mezzo del suo codice (un carattere) e due numeri interi, che rappresentano l'*offset* relativo (calcolato rispetto alla posizione corrente) dei due operandi. Quando tutti gli *offset* sono positivi, il circuito non presenta retroazioni e la struttura risultante è quella di un filtro con risposta a impulso finito. Per esempio, l'individuo

(I 0 2) (D 1 3) (L 2 2) (A 2 1) (D 1 0) (S 1 5)

corrisponde al diagramma schematico di figura 3.
La funzione di *fitness* è a due stadi: nel primo stadio, penalizza le violazioni delle specifiche della risposta in frequenza del filtro, rappresentate mediante una "maschera" nel grafico della risposta in frequenza; nel secondo stadio, che entra in azione nel momento in cui la risposta in frequenza è in maschera, la *fitness* è inversamente proporzionale all'attività del circuito, la quale dal canto suo è direttamente proporzionale all'assorbimento di potenza.
L'algoritmo evolutivo che risolve questo problema richiede molta potenza computazionale; per questo motivo è stato realizzato in modo distribuito, su un *cluster* di elaboratori elettronici, secondo un modello a *isole*, in cui la popolazione è divisa in un certo numero di isole, residenti su macchine distinte, che evolvono indipendentemente salvo scambiarsi, di tanto in tanto, degli individui "emigranti", che permettono di realizzare una circolazione di materiale genetico mantenendo allo stesso tempo la banda di comunicazione necessaria al funzionamento dell'algoritmo piccola a piacere.
Un risultato sorprendente di questo approccio evolutivo alla sintesi di circuiti elettronici è stato che i filtri digitali scoperti dall'evoluzione, oltre ad avere, come richiesto, un assorbimento di potenza nettamente inferiore rispetto ai filtri corrispondenti ottenuti con tecniche di sintesi tradizionali, mostrano una riduzione dal 40% al 60% del numero di elementi logici, e quindi anche dell'area e della velocità. La diminuzione dei consumi, in altre parole, non è stata ottenuta a spese del costo di produzione e della velocità, ma si è accompagnata con un generale aumento di efficienza rispetto ai metodi tradizionali di progettazione.

3.3. Data mining

Un fattore critico di successo per ogni azienda è la sua capacità di utilizzare le informazioni (e la conoscenza che se ne può estrarre) in modo efficace. L'uso strategico dei dati può derivare dalle opportunità offerte dalla

scoperta di fatti nascosti, precedentemente non riscontrati e spesso estremamente preziosi relativi ai consumatori, ai rivenditori e ai fornitori, all'andamento delle attività commerciali. Conoscendo queste informazioni, un'organizzazione è in grado di formulare strategie efficaci di marketing e vendita, focalizzare le azioni promozionali, scoprire e penetrare nuovi mercati e ottenere sul mercato una posizione di vantaggio rispetto ai concorrenti. L'attività di vaglio delle informazione al fine di ottenere un siffatto vantaggio competitivo è nota come *data mining* [13]. Da un punto di vista tecnico, il *data mining* può essere definito come la ricerca di correlazioni, schemi e tendenze non percepibili "ad occhio nudo" per mezzo del vaglio approfondito di grandi masse di dati immagazzinati in grandi banche dati e *data warehouse*, sfruttando metodi statistici, di intelligenza artificiale, di apprendimento automatico e di *soft computing*. Sono molte le grandi aziende e organizzazioni, come banche, assicurazioni, catene di grande distribuzione ecc., che dispongono di un'enorme quantità di informazioni sul comportamento dei loro clienti. La possibilità di sfruttare queste informazioni per inferire modelli del comportamento dei propri clienti attuali e futuri in relazione a prodotti specifici o a classi di prodotti è una prospettiva molto attraente per queste organizzazioni. I modelli così ottenuti potranno poi essere utilizzati per prendere decisioni strategiche e per meglio focalizzare le azioni di *marketing*, a patto che essi siano accurati, comprensibili e informativi.

Chi scrive ha partecipato, negli ultimi cinque anni, alla definizione, messa a punto e validazione di un potente motore per il *data mining*, sviluppato da Genetica Srl e da Nomos Sistema SpA (oggi un'azienda del gruppo Accenture) in collaborazione con l'Università degli Studi di Milano, nel quadro di due progetti Eureka finanziati dal Ministero dell'Istruzione e dell'Università (ex M.U.R.S.T.), che si basa sull'utilizzo di un algoritmo genetico per la sintesi di modelli predittivi del comportamento dei clienti, espressi mediante insiemi di regole *fuzzy* di tipo SE ... ALLORA. Questo approccio, tra l'altro, è un chiaro esempio dei vantaggi che possono essere conseguiti mediante la combinazione degli algoritmi evolutivi con la logica *fuzzy*, di cui si è parlato sopra.

L'approccio parte dalla disponibilità di un *data set*, cioè di un insieme, grande a piacere, di record che rappresentano osservazioni o registrazioni dei comportamenti passati di clienti. A dire il vero, il campo di applicabilità sarebbe ancor più vasto, considerando questi record come osservazioni puntuali di un qualche fenomeno, non necessariamente economico o commerciale, come per esempio la misurazione mediante radar di elettroni liberi nella ionosfera [14].

Un record è composto da m attributi, cioè valori di variabili che descrivono il cliente. Tra questi, si suppone che ne esista almeno uno che misura l'aspetto del comportamento dei clienti che si desidera modellare. Senza perdita di generalità, si può assumere che esista un solo attributo di questo genere: infatti, se fossimo interessati a modellare diversi aspetti del comportamento, potremmo sviluppare altrettanti modelli distinti. Potremmo chiamare questo attributo "predittivo", perché ci serve per predire il comportamento di un cliente. In questo quadro concettuale, un modello è una funzione con $m - 1$ parametri che esprime il valore dell'attributo predittivo in base ai valori degli altri attributi.

Il modo in cui si sceglie di rappresentare questa funzione è critico: l'esperienza dimostra che l'utilità e l'accettabilità di un modello non derivano soltanto dalla sua accuratezza, che, beninteso, è una condizione necessaria, ma anche e soprattutto dalla sua intellegibilità da parte dell'esperto che necessariamente lo dovrà valutare prima di autorizzarne l'utilizzo. Una rete neurale o un programma LISP, che altri autori hanno scelto come "linguaggi" per esprimere i modelli, può fornire risultati imbattibili quanto ad accuratezza, ma qualsiasi organizzazione sarà riluttante a "fidarsi" dei suoi risultati se non potrà comprendere e spiegare come essi siano stati ottenuti. Questa è la motivazione di fondo che ha determinato l'adozione degli insiemi di regole *fuzzy* SE ... ALLORA come linguaggio per l'espressione dei modelli. Tali insiemi di regole, infatti, costituiscono probabilmente quanto di più prossimo esista al modo intuitivo di esprimere la propria conoscenza da parte di

un esperto, grazie all'utilizzo di regole che esprimono relazioni tra variabili linguistiche (aventi cioè valori linguistici del tipo BASSO, MEDIO, ALTO). Inoltre, le regole *fuzzy*, sfumate, possiedono la proprietà molto desiderabile di comportarsi in modo *interpolativo*, cioè non saltano mai da una conclusione a quella opposta per colpa di un piccolo cambiamento nel valore di una condizione, come invece succede per le regole di tipo netto.
La codifica adottata per rappresentare un modello all'interno dell'algoritmo genetico è abbastanza complicata, ma rispecchia da vicino la struttura logica di un insieme di regole *fuzzy* e consente la definizione di operatori specifici di mutazione e ricombinazione che operano in modo informato sui suoi blocchi costitutivi. In particolare, l'operatore di ricombinazione è progettato in modo tale da preservare la legalità sintattica dei modelli; un modello figlio si ottiene combinando le regole dei due modelli genitori: ogni regola del modello figlio può essere ereditata dall'uno o dall'altro genitore con eguale probabilità e, quando viene ereditata, porta con sé tutte le definizioni dei valori linguistici (insiemi *fuzzy*) presenti nel genitore da cui proviene, che contribuiscono a determinarne la semantica.
I modelli sono valutati applicandoli a una parte del *data set* e ottenendo un valore di *fitness* che misura la loro accuratezza; la parte restante del *data set* viene utilizzata, come è prassi nell'apprendimento automatico, per monitorare le capacità di generalizzazione dei modelli ed evitare così il fenomeno dell'*overfitting*, che si ha quando un modello apprende uno per uno gli esempi che ha "visto", invece di catturare le regole generali che possono essere applicate anche a casi mai visti in precedenza.
Il motore che utilizza questo approccio è stato applicato con successo alla valutazione del credito in ambito bancario, alla stima della redditività dei clienti in campo assicurativo [15] e al recupero dei crediti al consumo.

4. CONCLUSIONI

Questa breve rassegna sugli algoritmi evolutivi ha cercato di fornire una panoramica completa, anche se per ovvi motivi di spazio non esaustiva, sui vari filoni tradizionali in cui si dividono (algoritmi genetici, strategie evolutive, programmazione evolutiva e programmazione genetica). Ha inoltre fornito alcuni elementi sulle problematiche più significative che riguardano l'applicazione pratica del calcolo evoluzionistico a problemi di rilevanza industriale ed economica, come la rappresentazione delle soluzioni e la gestione dei vincoli, per le quali la ricerca ha fatto negli ultimi anni sostanziali progressi. Infine, ha completato la trattazione con un'illustrazione più approfondita, pur se non appesantita da eccessivi dettagli tecnici, di tre esempi di applicazioni a problemi "del mondo reale", selezionati in domini il più possibile distanti tra loro, in modo tale da fornire tre visioni complementari sulle criticità e sulle problematiche che si incontrano nel realizzare, a partire dall'idea di fondo, un sistema *software* che "funzioni", ma anche da far apprezzare al lettore la versatilità e le enormi potenzialità di queste tecniche che, a quasi quarant'anni dalla loro prima introduzione, si trovano ancora nella loro adolescenza. Se c'è una lacuna in questa rassegna, è nei fondamenti teorici del calcolo evoluzionistico, che comprendono la teoria degli schemi (con l'ipotesi cosiddetta dei *building block*) e la teoria della convergenza, argomenti che sono stati volutamente tralasciati perché avrebbero richiesto un livello di formalismo inopportuno per un lavoro di rassegna: il lettore interessato potrà tuttavia colmare questa lacuna autonomamente, facendo riferimento alla bibliografia citata. Un altro aspetto che è stato trascurato perché non rappresenta una vera e propria "applicazione", ma che nondimeno riveste un grande interesse scientifico è quello dell'impatto che il calcolo evoluzionistico ha avuto sullo studio dell'evoluzione stessa e dei sistemi complessi in generale: si veda per esempio il lavoro di Axelrod sull'evoluzione spontanea di comportamenti cooperativi in un mondo di agenti egoisti [18].
Il lettore che volesse avvicinarsi al calcolo evoluzionistico può consultare alcuni eccellenti libri introduttivi [6, 9, 17, 19] o trattazioni più approfondite [20, 21], nonché visitare i siti Internet indicati nel riquadro **"Algoritmi evolutivi in Internet"**.

Algoritmi evolutivi in Internet
I seguenti sono alcuni siti web selezionati in cui il lettore può trovare materiale introduttivo o avanzato sugli algoritmi evolutivi:
- "http://www.isgec.org/": portale della International Society for Genetic and Evolutionary Computation;
- "http://evonet.lri.fr/": portale della rete di eccellenza europea sugli algoritmi evolutivi;
- "http://www.aic.nrl.navy.mil/galist/": GA Archives, nati come archivi della listi di distribuzione "GA-List", che oggi si chiama "EC Digest"; contiene informazioni aggiornate sugli eventi più importanti nel campo e link ad altre pagine web;
- "http://www.fmi.uni-stuttgart.de/fk/evolalg/index.html": EC Repository, mantenuto presso l'Università di Stoccarda.

Bibliografia

[1] Box George E.P., Draper N.R.: *Evolutionary Operation: Statistical Method for Process Improvement*. John Wiley & Sons, 1969.

[2] Bremermann Hans J.: *Optimization through Evolution and Recombination*. In: Yovits M.C., Jacobi G.T., Goldstein G.D. (a cura di), *Self-Organizing Systems 1962*. Spartan Books, Washington D. C., 1962.

[3] Fogel Lawrence J., Owens A. J., Walsh M.J.: *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York, 1966.

[4] Holland John H.: *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[5] Rechenberg Ingo: *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, Stoccarda, 1973.

[6] Goldberg David E.: *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[7] Koza John R.: *Genetic Programming*. MIT Press, Cambridge, Massacusets, 1992.

[8] O'Neill Michael, Ryan Conor: *Grammatical Evolution*. Evolutionary automatic programming in an arbitrary language. Kluwer, 2003.

[9] Michalewicz Zbigniew: Genetic *Algorithms + Data Structures = Evolution Programs*. 3rd Edition. Springer, Berlino, 1996.

[10] Tettamanzi Andrea G.B., Tomassini Marco: *Soft Computing. Integrating evolutionary, neural, and fuzzy systems*. Springer, Berlino, 2001.

[11] Di Stefano Calogero, Tettamanzi Andrea G.B.: *An Evolutionary Algorithm for Solving the School Time-Tabling Problem*. In: Boers E. et al., *Applications of Evolutionary Computing*. EvoWorkshops 2001, Springer, 2001, p. 452-462.

[12] Erba Massimiliano, Rossi Roberto, Liberali Valentino, Tettamanzi Andrea G.B.: *Digital Filter Design Through Simulated Evolution*. Atti di ECCTD'01 - European Conference on Circuit Theory and Design, 28-31 agosto 2001, Espoo, Finlandia.

[13] Berson Alex, Smith Stephen J.: *Data Warehousing, Data Mining & OLAP*. McGraw Hill, New York, 1997.

[14] Beretta Mauro, Tettamanzi Andrea G.B.: *Learning Fuzzy Classifiers with Evolutionary Algorithms*. In: Bonarini A., Masulli F., Pasi G., (a cura di), *Advances in Soft Computing*. Physica-Verlag, Heidelberg, 2003, p. 1-10.

[15] Tettamanzi Andrea G.B., et al.: *Learning Environment for Life-Time Value Calculation of Customers in Insurance Domain*. In: Deb K., et al. (a cura di), Proceedings of the Genetic and Evolutionary Computation Congress (GECCO 2004), S. Francisco, 26–30 giugno 2004, p. 1251-1262.

[16] Rudolph Günter: Finite Markov Chain Results in Evolutionary Computation: A Tour d'Horizon. *Fundamenta Informaticae*, Vol. 35, 1998, p. 67-89.

[17] Mitchell Melanie: *An Introduction to Genetic Algorithms*. Bradford, 1996.

[18] Axelrod Robert: *The Evolution of Cooperation*. Basic Books, 1985.

[19] Fogel David B.: *Evolutionary Computation: Toward a new philosophy of machine intelligence*. 2nd Edition. Wiley-IEEE Press, 1999.

[20] Bäck Thomas: *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996.

[21] Bäck Thomas, Fogel David B., Michalewicz Zbigniew (a cura di): *Evolutionary Computation* (2 volumi). IoP, 2000.

Andrea Tettamanzi è professore associato presso il Dipartimento di Tecnologie dell'Informazione dell'Università degli Studi di Milano. Laureato in Scienze dell'Informazione nel 1991, ha conseguito il Dottorato di Ricerca in Matematica Computazionale e Ricerca Operativa nel 1995, anno in cui ha fondato Genetica Srl, un'azienda specializzata nelle applicazioni industriali degli algoritmi evolutivi e del *soft computing*. Attivo nella ricerca sugli algoritmi evolutivi e sul *soft computing*, ha sempre cercato di coniugare gli aspetti teorici con quelli pratici e applicativi.
andrea.tettamanzi@unimi.it