Ottimizzazione, apprendimento automatico ed algoritmi basati su metafora naturale

Marco Dorigo
PM-AI & R
Dipartimento di Elettronica
Politecnico di Milano
P.za Leonardo Da Vinci 32
20133 Milano - Italia
e-mail: dorigo@ipmel2.elet.polimi.it

Tesi di Dottorato in
Ingegneria Elettronica dell'Informazione e dei Sistemi
IV Ciclo (1988-91)

Relatore: Professor Alberto Colorni
Correlatore: Professor Alberto Bertoni

Sommario

In questo lavoro vengono studiati alcuni algoritmi appartenenti alla classe degli "algoritmi naturali". Questi algoritmi hanno finora trovato applicazione prevalentemente nei settori dell'ottimizzazione - combinatoria e non - e dell'apprendimento automatico.

A questa classe appartengono algoritmi che presentano almeno alcune delle seguenti caratteristiche:

(i) l'idea fondamentale sulla quale si basa la strutture dell'algoritmo è stata ispirata dall'osservazione di un qualche fenomeno naturale;
(ii) l'algoritmo proposto è stocastico;
(iii) l'algoritmo proposto è intrinsecamente parallelo (cioè la sua implementazione più naturale è quella sotto forma di molti agenti cooperanti per la soluzione del problema);
(iv) l'algoritmo è adattativo: cioè utilizza il risultato delle decisioni effettuate nel passato per dirigere il processo di ricerca [81];
(v) l'algoritmo è autocatalitico ovvero utilizza una retroazione positiva per rendere più veloce il processo di ricerca di una soluzione.

È bene sottolineare che lo scopo della ricerca presentata in questa tesi è quello di mostrare come algoritmi che si basano sulla cooperazione-competizione di molti agenti semplici possano presentare caratteristiche molto desiderabili, quali robustezza, adattatività, velocità nel trovare soluzioni buone, ampio spettro di applicabilità.

Nel primo capitolo di questa tesi vengono date brevemente alcune chiavi di lettura del lavoro.

Nel secondo capitolo vengono introdotti alcuni algoritmi naturali, tra i quali quello delle "formiche" proposto dall'autore: in questa introduzione si tenta di classificare tali algoritmi lungo alcune dimensioni - vedi le caratteristiche di cui sopra - e di mettere in luce i pregi e i limiti di ognuno degli approcci. Nei capitoli seguenti vengono presentati i risultati ottenuti applicando alcuni di questi algoritmi a diversi problemi, tratti sia dal settore dell'ottimizzazione combinatoria che dell'apprendimento automatico.

Ringraziamenti

Voglio per prima cosa ringraziare il professor Alberto Colorni, relatore del presente lavoro, per avere collaborato nello sviluppo di molte delle idee presentate in questa tesi e per il tempo speso per la lettura della stessa nelle sue varie versioni, e il professor Alberto Bertoni, controrelatore, che ha trovato il tempo per leggere, discutere e giudicare la stesura finale del manoscritto.

Un sentito ringraziamento va al professor Marco Somalvico, che avrebbe dovuto essere mio relatore per una tesi sull'apprendimento automatico, per avermi consigliato durante i tre anni di durata della ricerca in qualità di tutore e per aver accettato di buon grado la mia scelta di porre maggior enfasi di quanto lui avesse desiderato sugli aspetti di ottimizzazione. Voglio inoltre ringraziare tutto il gruppo del Progetto di Intelligenza Artificiale e Robotica per avere contribuito, sia tramite stimolanti conversazioni che per mezzo degli strumenti di calcolo a sua disposizione, allo svolgimento delle ricerche di questo lavoro che più hanno a che fare con l'apprendimento automatico.

Voglio inoltre ringraziare in modo particolare Vittorio Maniezzo per avere condiviso la mia passione per gli algoritmi stocastici e l'apprendimento automatico, Uwe Schnepf per avere a lungo discusso con me delle possibili applicazioni di queste tecniche alla robotica, Hughes Bersini, Andrea Bonarini e Jean-Pierre Nordvik per le discussioni riguardo l'applicabilità a problemi di controllo adattativo. Un sentito ringraziamento va anche al professor Francesco Maffioli per avermi per primo introdotto ai segreti della matematica discreta e della NP-completezza; e al professor Christian Freksa per avermi accolto come visitatore presso la sua unità di ricerca presso l'università tecnica di Monaco.

Al professor Marco Colombetti, con il quale ho intrattenuto lunghe e piacevoli discussioni su ogni argomento che anche lontanamente risuonasse con le idee presentate in questa tesi, vanno i miei più grati ringraziamenti e la speranza di poter continuare la insolitamente stimolante collaborazione.

Infine un ringraziamento al professor Crespi-Reghizzi e a tutti coloro i quali, con il loro lavoro, hanno permesso uno svolgimento regolare e senza intoppi dell'intero periodo dedicato al dottorato.

Parte del lavoro qui presentato è stato possibile grazie ai finanziamenti dei seguenti progetti CNR:

Progetto finalizzato robotica - Sottoobiettivo 2 - Tema: ALPI
Progetto finalizzato sistemi informatici e calcolo parallelo - Sottoprogetto 2 - Tema: Processori dedicati

Indice

1 Introduzione 1

1.1 Chiavi di lettura 2
1.2 Nota sull'evoluzione del lavoro 4

2 Algoritmi basati su metafora naturale 5

2.1 Introduzione 6

2.2 Il sistema formiche 7
2.2.1 Origine del modello 7
2.2.2 Caratteristiche generali 9
2.2.3 Caratteristiche del modello e risultati attesi 11
2.2.4 Caratteristiche naturali del sistema formiche 13

2.3 Algoritmi genetici 14
2.3.1 Gli operatori genetici 15
2.3.2 Ammissibilità dei cromosomi 16
2.3.3 Risultati teorici fondamentali 17
2.3.4 Considerazioni conclusive 23
2.3.5 Caratteristiche naturali degli algoritmi genetici 24

2.4 Sistemi a classificatori 25
2.4.1 Un problema 25
2.4.2 Il sistema di performance 27
2.4.3 Il sistema di valutazione delle regole 29
2.4.4 Gerarchie di default 31
2.4.5 La generazione di nuove regole: l'algoritmo genetico applicato ai sistemi a classificatori 32
2.4.6 La funzione di valutazione 33
2.4.7 Caratteristiche naturali dei sistemi a classificatori 33

2.5 Strategie evolutive 34
2.5.1 Gli operatori 35
2.5.2 Caratteristiche naturali delle strategie evolutive 37

2.6 Reti immunitarie 38
2.6.1 Componenti del Sistema Immunitario 38
2.6.2 Il meccanismo di arruolamento come strumento di ottimizzazione 40
2.6.3 Caratteristiche naturali delle reti immunitarie 42

3 Il sistema "formiche" e il problema del commesso viaggiatore 43

3.1 Il sistema formiche 44

3.2 Gli algoritmi Ant-density e Ant-quantity 46

3.3 L'algoritmo Ant-cycle 48

3.4 La taratura dei parametri 50

3.5 Ulteriori indagini su Ant-cycle 54
3.5.1 Alcune proprietà generali del modello Ant-cycle 54
3.5.2 Numero di formiche 56
3.5.3 Come posizionare le formiche nell'istante iniziale? 58
3.5.4 Strategia elitista 59
3.5.5 Probabilità di transizione con rumore 60
3.5.6 Tempo necessario per trovare la soluzione ottima 61
3.5.7 Paragoni con altri algoritmi specializzati per il TSP 62

3.6 Applicazioni ad altri problemi 63

3.7 Conclusioni e lavoro futuro 64

4 L'algoritmo genetico e il problema dell'orario scolastico 65

4.1 Introduzione 66

4.2 Il problema dell'orario scolastico 67

4.3 Gli algoritmi genetici e il problema dell'orario scolastico 68
4.3.1 Gli operatori genetici 69

4.4 La funzione obiettivo e la fitness function 71

4.5 Complessità dell'algoritmo 73

4.6 Risultati 79

4.7 Conclusioni e sviluppi futuri 82

5 I sistemi a classificatori e il problema del robot autonomo 83

5.1 Introduzione 84

5.2 Quale architettura software? 85
5.2.1 Il modello di Tinbergen 85
5.2.2 Il modello su cui si basa ALECSYS 87

5.3 Modifiche apportate al sistema a classificatori standard 90
5.3.1 L'algoritmo Message-based Bucket Brigade 90
5.3.2 Innovazioni in ALECSYS 95
5.3.2.1 L'operatore *mutespec* 97
5.3.2.2 Valutazione del raggiungimento della situazione di regime 99
5.3.2.3 Numero di regole da sostituire ad ogni fase di genetica 101
5.3.2.4 Utilizzo di un insieme ridotto di regole 103
5.3.2.5 Lunghezza ottima della lista dei messaggi interni 106
5.3.2.6 Utilità dei messaggi interni 108
5.3.2.7 Utilità dei diversi operatori genetici 108

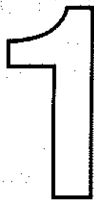
5.4 ALECSYS: un sistema a classificatori parallelo 110
5.4.1 Parallelizzazione di basso livello di un SC 110
5.4.1.1 I sistemi di performance e di valutazione delle regole .. 111
5.4.1.2 Il sistema di generazione delle regole 113
5.4.2 Parallelizzazione di alto livello di un SC 114

5.5 Valutazione sperimentale di ALECSYS 116
5.5.1 Esperimenti in un ambiente bidimensionale 116
5.5.2 Caratteristiche generali del robot simulato 117
5.5.3 Esperimento (a): inseguimento di una luce 118
5.5.4 Esperimento (b): un robot *camaleonte* 121
5.5.4.1 Descrizione delle componenti del sistema parallelo ... 123
5.5.4.2 Descrizione delle componenti del sistema distribuito .. 124
5.5.4.3 Risultati delle simulazioni 125
5.5.5 Esperimento (c): seguire una luce evitando oggetti pericolosi .. 126
5.5.6 Esperimento (d): seguire due sorgenti luminose (stando alla minor distanza possibile dalle due) 128

5.6 Lavori correlati 130

5.7 Conclusioni e sviluppi futuri 132

6 Bibliografia 133



Introduzione

1.1 Chiavi di lettura
1.2 Nota sull'evoluzione del lavoro

1.1 Chiavi di lettura

Questa tesi tenta di presentare in modo unitario alcuni algoritmi che definiamo appartenere alla classe degli algoritmi naturali. Le chiavi di lettura del lavoro sono molteplici.

Da un lato c'è l'interesse per una delle caratteristiche principali dei processi naturali: l'***adattatività***. In questa tesi abbiamo definito adattativo un sistema che sia in grado di migliorare le sue prestazioni tramite l'uso di retroazione (feedback)[1]. Questa retroazione può essere generata dalla interazione con l'ambiente, come nel caso presentato nel capitolo 5, o da una funzione di valutazione, come nei casi esposti nei capitoli 3 e 4. È chiaro che l'utilità di utilizzare la retroazione risiede nella mancata disponibilità di meccanismi di decisione su come procedere che siano più potenti. Un approccio adattativo molto promettente e attualmente molto studiato sembra essere quello delle reti neurali. In questa tesi abbiamo voluto esaminare le caratteristiche di adattatività di algoritmi che, come le reti neurali, prendono spunto dall'osservazione della natura, ma che, a differenza delle reti neurali, sono stati finora meno studiati.

Un altro aspetto fondamentale è quello della ricerca di algoritmi che siano "veloci". Questo obiettivo porta con sé due caratteristiche progettuali dei suddetti algoritmi: la ***stocasticità*** e il ***parallelismo***. La stocasticità permette di limitare il problema dei minimi locali consentendo al contempo di trovare soluzioni "buone" in tempi sufficientemente brevi e di mantenersi pronti ad esaminare nuove soluzioni nel caso in cui le caratteristiche del problema si dovessero modificare (ed è perciò strettamente legata anche alla caratteristica di adattatività). Il parallelismo è in primo luogo una risposta al bisogno di avere molta potenza di calcolo per trovare soluzioni buone in tempi brevi. Gli algoritmi naturali sono intrinsecamente paralleli; essi infatti prendono ispirazione da fenomeni naturali che avvengono in parallelo, e ne conservano questa desiderabile caratteristica che permette una loro spesso immediata implementazione su calcolatori paralleli.

Inoltre vi è l'interesse per un fenomeno di sapore olistico che abbiamo riscontrato in almeno alcuni di questi algoritmi: spesso il risultato dell'interazione di tanti agenti semplici è qualitativamente diverso dalla somma dei risultati ottenuti dai singoli agenti separatamente. In alcuni casi è possibile osservare un fenomeno che ho chiamato di *ricerca autocatalitica*: il sistema impara molto velocemente una soluzione molto buona perché utilizza l'esperienza passata, utilizzando un processo retroazionato positivamente (per maggiori dettagli si veda il capitolo 3).

Infine c'è l'interesse per la valutazione della effettiva *utilizzabilità* delle tecniche proposte: alcuni capitoli della tesi sono dedicati all'analisi ed alla soluzione di problemi difficili utilizzando alcune delle tecniche presentate nei capitoli precedenti.

In questa breve introduzione vorrei mettere in risalto un'ulteriore, forse meno canonica, ma secondo me non meno interessante, chiave di lettura. Per far ciò vorrei partire da quello che è il principio di ottimalità. Questo principio è stato utilizzato con successo da molte discipline scientifiche (per esempio: economia, fisica, chimica, biologia) e si potrebbe enunciare nel modo seguente:

[1] Per una definizione più formale si veda [81].

Il principio di ottimalità è invocato nella modellizzazione di un comportamento o di un fenomeno fisico quando è possibile far dipendere la bontà del risultato da un processo di massimizzazione o minimizzazione di una qualche funzione obiettivo.

Purtroppo questo principio, che pervade la cultura scientifica (vedi [77]), è in qualche modo fuorviante qualora si voglia trovare una soluzione al problema di progettare dei sistemi che presentino un comportamento intelligente. Io credo che una caratteristica fondamentale dell'intelligenza sia la "non ottimalità"[2]: il comportamento ottimo è possibile solo in presenza di problemi semplici, statici e ben capiti (caratteristiche che purtroppo non si riscontrano spesso nei problemi risolti dagli esseri viventi).

Prendiamo ad esempio il problema del commesso viaggiatore, problema molto semplice da definire: date n città si deve determinare il cammino più breve che, partendo da una qualsiasi città, vi ritorni dopo avere toccato tutte le altre. La versione di decisione di questo semplice problema è NP-completa [12], [64], [49] ed è perciò altamente improbabile che esista un algoritmo che sia in grado di trovare il cammino minimo in tempo polinomiale nelle dimensioni dei dati di input. Questo implica che, al crescere del numero delle città, il tempo richiesto per trovare la soluzione ottima col miglior algoritmo conosciuto cresce superpolinomialmente e che, per quanto sia potente il calcolatore utilizzato, esistono istanze del problema di dimensioni limitate che possono richiedere tempi di calcolo non ammissibili [49] (l'importanza del concetto di NP-completezza risiede nel fatto che a questa classe appartengono una miriade di problemi che sono polinomialmente isomorfi [12], [5], [65]: qualora si dovesse trovare una soluzione in tempo polinomiale ad uno di essi la si sarebbe trovata per tutti gli altri).

In realtà molti dei problemi risolti dagli esseri viventi presentano un tipo di difficoltà differente, legato al fatto che l'informazione disponibile è parziale, rumorosa e costosa [81]. Dati i vincoli sulle risorse disponibili, l'obiettivo degli essere viventi non può essere quello dell'ottimizzazione, ma diviene quello della sopravvivenza.

Abbiamo quindi due prospettive differenti quando studiamo algoritmi che si pongono come obiettivo la soluzione di problemi difficili: una è quella dell'ottimizzatore (o del matematico), interessato a trovare soluzioni ottime, l'altra è quella del progettista di sistemi intelligenti, interessato a fornire il sistema intelligente *in fieri* di un risolutore di problemi che fornisca al contempo soluzioni sufficientemente buone (affinché sia garantita la sopravvivenza), e sufficientemente rapide, cioè ottenute in un tempo commensurato alla scala temporale nella quale il sistema si muove (quindi in tempi che siano sufficientemente brevi da permettere al sistema di reagire sensatamente alla dinamica dell'ambiente e di fronteggiare modifiche inattese alla struttura del problema da risolvere).

Se da un lato questa seconda prospettiva può apparire di più semplice soluzione, perché rimuove il vincolo di ottimalità, d'altra parte non è a tutt'oggi chiaro come l'introduzione delle nuove caratteristiche cui deve soddisfare l'algoritmo influenzi la complessità della progettazione dell'algoritmo stesso.

Spero di avere contribuito con questa tesi, seppur limitatamente, ad una migliore comprensione di quest'ultimo aspetto.

[2] Con ciò indichiamo il fatto che il comportamento degli esseri intelligenti prescinde dal principio di ottimalità e non che gli esseri intelligenti non siano interessati a comportamenti ottimi.

1.2 Nota sull'evoluzione del lavoro

Il lavoro presentato in questa tesi sviluppa una serie di temi diversi intorno ad alcuni aspetti unificanti - algoritmi naturali, ottimizzazione autocatalitica, sistemi adattativi, parallelismo. Può forse avere una sua importanza il conoscerne l'evoluzione logico-cronologica.

Il mio primo interesse per gli algoritmi naturali nacque per caso in seguito alla lettura di un libro, all'epoca il primo, sugli algoritmi genetici. Non essendovi, a mia conoscenza, nessuno in Italia che lavorasse nel settore, ho cercato nel "vicinato" europeo, trovando interesse e competenze nel gruppo del professor Freksa presso l'università tecnica di Monaco in Germania, dove ho passato sette mesi durante i quali ho approfondito le mie conoscenze su algoritmi genetici e loro applicazioni in apprendimento automatico e ottimizzazione combinatoria (nonché della lingua tedesca!). In quel periodo ebbi anche il mio primo incontro/scontro con i transputer, che sono poi diventati un importante strumento di sviluppo per le mie ricerche. È sempre del periodo passato a Monaco il mio primo contatto con il formalismo che va sotto il nome di reti neurali.

Un seminario da me tenuto a Milano dopo il ritorno dalla Germania ha causato l'inizio della collaborazione con il professor Colorni, poi divenuto mio relatore. Primo lavoro nato da questo incontro è stato l'applicazione degli algoritmi genetici ad un problema noto per la sua particolare difficoltà: il problema dell'orario scolastico, del quale si parla diffusamente nel capitolo 4.

Nel contempo alcuni contatti insorti quasi per caso con un giovane ricercatore del GMD[3] di Bonn - Uwe Schnepf - mi portarono all'ideazione di un progetto ambizioso, di cui si parla diffusamente nel capitolo 5: il progetto e la realizzazione di un robot autonomo il cui sistema di controllo fosse un sistema di apprendimento automatico basato su algoritmi genetici.

È sempre casualmente che, durante un workshop su sistemi intelligenti distribuiti organizzato a Bonn da Uwe Schnepf, sono venuto a conoscenza delle ricerche condotte da Goss (allievo di Prigogine) e Denebourg sul comportamento delle colonie di formiche. Il passo che mi conduce al tentativo di formalizzare un nuovo algoritmo naturale è breve e, dopo un primo tentativo sfortunato, riesco, con la collaborazione degli ormai affiatati Alberto Colorni e Vittorio Maniezzo, a definire ed implementare un primo sistema - chiamato *sistema formiche* - del quale si discute diffusamente nel capitolo 3.

Il motivo per cui il "sistema formiche" viene presentato prima delle applicazioni basate su algoritmi genetici, non rispettando pertanto l'ordine cronologico, è che esso rappresenta l'aspetto più innovativo introdotto da questa tesi e quello più suscettibile di sviluppi futuri.

Un ulteriore aspetto interessante è presentato nel capitolo 2, nel quale definisco la classe degli algoritmi naturali e presento le caratteristiche di alcuni di essi.

---

[3] Il GMD - Gesellschaft für Mathematik und Datenverarbeitung - è un grosso centro di ricerca situato nelle vicinanze di Bonn.

2

Algoritmi basati su metafora naturale

2.1 Introduzione
2.2 Il sistema formiche
2.3 Algoritmi genetici
2.4 Sistemi a classificatori
2.5 Strategie evolutive
2.6 Reti immunitarie

2.1 Introduzione

In questo capitolo verranno presentati alcuni algoritmi che fanno parte della classe degli algoritmi naturali. Nel seguito riportiamo le caratteristiche che ci paiono discriminanti per questa classe. Non tutti gli algoritmi soddisfanno a tutte le caratteristiche: in questo senso possiamo dire che alcuni algoritmi appartengono alla classe degli algoritmi naturali "più di altri". Le suddette caratteristiche sono riportate in ordine di presunta importanza.

A) L'algoritmo prende ispirazione da un processo che si trova in natura.
B) L'algoritmo è intrinsecamente parallelo (nel senso che utilizza una popolazione di semplici agenti cooperanti per risolvere il problema).
C) L'algoritmo è stocastico.
D) L'algoritmo è adattativo (cioè è in grado di migliorare le sue prestazioni tramite l'uso di retroazione).
E) L'algoritmo fa uso di retroazione positiva (cioè quando in una data situazione una decisione porta ad un effetto positivo, allora viene aumentata la probabilità che, qualora nel futuro si ripresenti la stessa situazione, venga presa di nuovo la stessa decisione).
F) L'algoritmo fa uso del concetto di evoluzione e selezione.

Riportiamo di seguito una tabella riassuntiva nella quale per ogni algoritmo sono mostrate le caratteristiche presenti a pieno titolo, quelle presenti solo parzialmente e quelle non presenti. Per una spiegazione più dettagliata della tabella si vedano le sezioni di questo capitolo dedicate ai vari algoritmi.

Tabella 2.1 - Caratteristiche dei diversi algoritmi naturali
● = caratteristica presente
O = caratteristica presente solo parzialmente
• = caratteristica assente

	A	B	C	D	E	F
Sistema formiche	●	●	●	●	●	•
Algoritmi genetici	●	●	●	●	●	●
Sistemi a classificatori	O	●	●	●	O	O
Strategie evolutive	●	●	●	●	●	●
Reti immunitarie	●	●	●	●	•	●
Annealing simulato	●	•	●	●	•	•
Apprendimento su reti neurali	●	O	O	●	•	•

Le indicazioni della tabella sono ovviamente soggettive e andrebbero forse motivate: per i primi cinque tipi di algoritmi ciò verrà fatto nel seguito di questo capitolo.

2.2 Il sistema formiche

Il regno animale presenta molti casi di sistemi sociali nei quali le capacità possedute dai singoli individui sono molto limitate se confrontate con le capacità organizzative e di comportamento del sistema sociale nel suo complesso. Questi sistemi si possono osservare in animali che si posizionano a diversi stadi del processo evolutivo: dai batteri [79] alle formiche [57], dai bruchi [46] ai molluschi ed alle larve. Inoltre gli stessi processi che causano alcuni di questi comportamenti aggregati sono conservati in alcune specie più evolute, quali ad esempio i pesci, gli uccelli ed i mammiferi. Queste specie utilizzano mezzi di comunicazione vieppiù sofisticati, ma che possono condurre, in particolari situazioni, a comportamenti analoghi a quelli presentati dagli esseri meno evoluti prima citati.

Tutto ciò suggerisce che questi meccanismi si siano dimostrati vincenti da un punto di vista evoluzionistico e che pertanto la strategia di risoluzione di problemi per mezzo della distribuzione della capacità risolutiva ad un insieme di agenti semplici e cooperanti per mezzo di un qualche meccanismo non direttamente sotto il controllo di alcuno di essi possa essere un interessante alternativa da esplorare.

Uno dei sistemi sociali più conosciuti e studiati è quello delle colonie di formiche [29]: nella sezione seguente presentiamo le caratteristiche principali dei modelli finora sviluppati per spiegare un interessante comportamento delle formiche che ci servirà per definire nel seguito un algoritmo di ottimizzazione distribuito che applicheremo al problema del commesso viaggiatore.

2.2.1 Origine del modello

L'osservazione da cui muoviamo è la seguente: le formiche, insetti quasi privi dell'organo della vista, riescono spesso a trovare il cammino più breve tra il formicaio e la sorgente di cibo. Come è possibile?

Il mezzo utilizzato per trasmettere informazione riguardo al cammino seguito ed utilizzato nel processo decisionale è una sostanza, detta feromone, che le formiche depositano quando camminano. Una formica deposita una quantità variabile di feromone sul terreno, marcando in questo modo il cammino seguito per mezzo della traccia di questa sostanza. Mentre una formica isolata si muove essenzialmente in modo casuale, una formica che incontri un cammino seguito in precedenza da un altra formica può scegliere di seguire il cammino già tracciato (e la probabilità che la scelta avvenga effettivamente dipende dall'intensità della traccia percepita), a sua volta lasciando del nuovo feromone che si somma a quello preesistente. Il comportamento collettivo emergente è una forma di comportamento *autocatalitico* – o *allelomimesis* o anche a *retroazione positiva* – nel quale più sono le formiche che seguono un determinato cammino, più quel cammino diviene attraente per le prossime formiche che dovessero incontrarlo. Il processo è pertanto caratterizzato da una retroazione positiva, data dal fatto che la probabilità con la quale una formica sceglie un determinato cammino aumenta con il numero delle formiche che hanno scelto quello stesso cammino precedentemente.

In Fig.2.1 presentiamo un esempio di come questo processo autocatalitico possa molto rapidamente condurre un gruppo di formiche all'identificazione del cammino più breve intorno ad un ostacolo.

L'esperimento mostrato in Fig.2.1 è il seguente: un gruppo di formiche sta percorrendo in entrambi i sensi un determinato sentiero (per esempio in Fig.2.1a potrebbe essere il cammino tra una sorgente di cibo in A e il formicaio in E). Improvvisamente un ostacolo interrompe il flusso delle formiche in entrambe le direzioni. Questa nuova situazione impone alle formiche che raggiungono il punto D venendo da E (e a quelle che raggiungono il punto B venendo da A) di operare una scelta tra il cammino di destra e quello di sinistra (vedi Fig.2.1b). La scelta è influenzata dalla intensità del feromone lasciato dalle formiche precedenti: un livello di feromone più elevato sul cammino di destra causa uno stimolo più forte e quindi una più alta probabilità che la formica scelga di svoltare a destra.

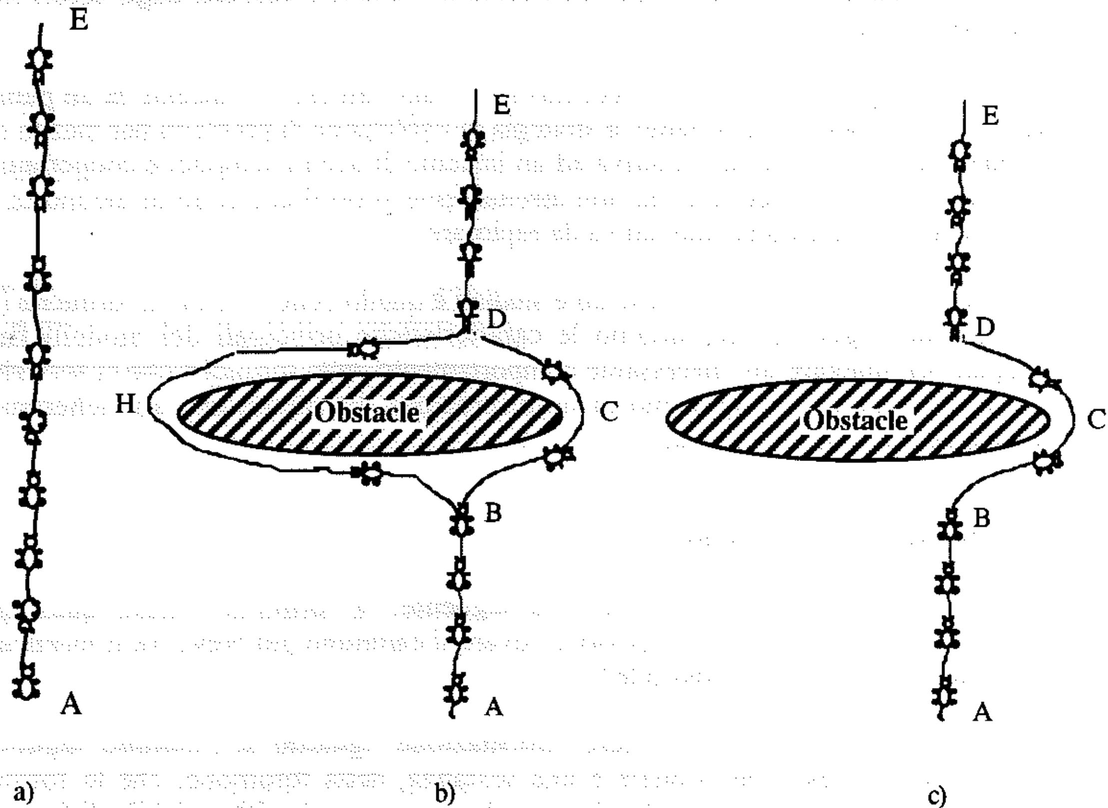


Fig.2.1 - a) Alcune formiche percorrono un sentiero che unisce i punti A ed E.
b) Un ostacolo interrompe il flusso di formiche (che devono perciò aggirarlo).
c) All'equilibrio le formiche hanno scelto il cammino più breve.

La prima formica che arriva nel punto D (o B) dopo la comparsa dell'ostacolo ha la stessa probabilità di girare a destra o a sinistra, data la completa assenza di feromone su entrambi i possibili cammini. Consideriamo due formiche provenienti da A e supponiamo che una scelga di raggiungere D passando per C e l'altra passando per H. Dato che il cammino BCD è più breve del cammino BHD, la formica che lo segue raggiunge D prima della prima formica che ha scelto di seguire BHD. Ciò fa sì che le formiche che vengono nel senso opposto (cioè da E verso A) trovino una traccia più forte sul cammino DCB, traccia lasciata dalla metà (in media) delle formiche provenienti da E che hanno scelto per caso il cammino DCB e dalle formiche provenienti da A già arrivate (in altre parole, dato che le formiche che scelgono il tratto più breve ci mettono meno tempo a percorrerlo, possono influenzare la scelta delle formiche che vengono nel senso opposto prima di quanto possano farlo le formiche che hanno scelto il tratto più lungo). Pertanto il cammino DCB verrà scelto da un numero più elevato di formiche che non il cammino DHB. Questa analisi vale evidentemente (pur di cambiare A con E e B con D) anche per

le formiche provenienti da E. Inoltre man mano che il processo procede il rapporto tra le intensità di traccia sui cammini DCB e DHB diverrà sempre più elevato, con il risultato finale che tutte le formiche sceglieranno di seguire il cammino più breve (Fig.2.1c). Un aspetto molto importante, come vedremo nel capitolo 3, è che la decisione riguardante il cammino da scegliere rimane sempre probabilistica, rendendo comunque possibile una esplorazione di nuovi cammini.

2.2.2 Caratteristiche generali

Introduciamo ora i concetti e le definizioni che costituiscono la base dei vari algoritmi (a tutt'oggi tre) appartenenti al sistema formiche (SF).

Il sistema è composto di semplici agenti, detti *formiche*, che interagiscono tra di loro. In questa sezione descriveremo le caratteristiche generali di un insieme di algoritmi basati su questo sistema, mentre i dettagli e le singole particolarità verranno introdotte nel capitolo 3.

La prima applicazione del sistema formiche è stata pensata per il problema del commesso viaggiatore: pertanto utilizzeremo tale problema come esempio per facilitare l'esposizione. Inoltre i risultati sperimentali sono stati ottenuti applicando il sistema formiche a questo problema. Nondimeno, come vedremo nella sezione 3.6, non dovrebbe essere difficile estendere l'algoritmo ad altri problemi di ottimizzazione combinatoria.

Il problema del commesso viaggiatore (TSP - Travelling Salesman Problem) può essere definito come segue.

ISTANZA: Una sequenza di città $c_1, c_2, \ldots, c_n$, per ogni arco (c_i, c_j) $(i \neq j)$ una distanza $d(c_i, c_j)$.

QUESTIONE: Trovare una permutazione π delle città che minimizzi la quantità

$$\sum_{i=1}^{n-1} \mathbf{d}(c_{\pi(i)}, c_{\pi(i+1)}) + \mathbf{d}(c_{\pi(n)}, c_{\pi(1)})$$

Sia $b_i(t)$ $(i=1, \ldots, n)$ il numero di formiche nella città i al tempo t e sia

$$m = \sum_{i=1}^{n} b_i(t)$$

il numero totale di formiche.

La distanza $d_{ij}=d(c_i, c_j)$ è , nel caso del TSP euclideo nel piano, la distanza Euclidea tra i e j (cioè: $d_{ij}=[(x_i-x_j)^2 + (y_i-y_j)^2]^{1/2}$).

In modo informale introduciamo le caratteristiche delle formiche come segue.

Ogni formica è un agente che:

- quando si sposta dalla città i alla città j deposita una sostanza, detta traccia, sull'arco (i,j),
- sceglie la città j dove andare con una probabilità che è funzione della distanza da percorrere e della quantità di traccia che è presente sui vari archi della rete che escono dalla città i,
- memorizza le città già visitate (ciò è fatto per forzare le formiche a compiere dei cicli: le città già visitate sono automaticamente escluse dall'insieme delle città sceglibili fintanto che un ciclo completo non sia stato effettuato; vedi il concetto di *tabu list* più avanti).

Queste caratteristiche fanno si che il nostro modello si discosti dal comportamento delle formiche reali, in quanto ogni formica artificiale è dotata di capacità di memorizzazione e capacità di valutazione delle distanze. Inoltre il nostro è un sistema a tempo discreto.

Definita $\tau_{ij}(t+1)$ la *intensità di traccia* sull'arco (i,j) all'istante t+1, essa soddisfa la seguente equazione

$$\tau_{ij}(t+1)=\rho\cdot\tau_{ij}(t)+\Delta\tau_{ij}(t,t+1) \qquad (2.1)$$

dove ρ è un coefficiente (intuitivamente $(1-\rho)$ rappresenta l'*evaporazione* della traccia) e

$$\Delta\tau_{ij}(t,t+1) = \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t,t+1)$$

essendo $\Delta\tau_{ij}^{k}(t,t+1)$ la quantità per unità di lunghezza (intensità) di traccia (feromone nel caso delle formiche vere) depositato sull'arco (i,j) dalla k-esima formica nell'intervallo di tempo (t,t+1).

L'intensità della traccia all'istante iniziale, $\tau_{ij}(0)$, può essere fissata ad un valore iniziale arbitrariamente scelto (nei nostri esperimenti è stato scelto lo stesso valore iniziale per ogni arco (i,j)).

Affinché il vincolo che ogni formica visiti tutte le n città (percorra cioè un ciclo completo) sia soddisfatto, associamo ad ogni formica una struttura dati, chiamata *tabu list*[1], che memorizza le città già visitate fino all'istante t e impedisce che queste vengano visitate di nuovo prima che un ciclo sia completato.

Quando è completato la *tabu list* viene vuotata e la formica è di nuovo libera di scegliere il proprio cammino.

Definiamo $\mathbf{tabu}_k$ un vettore che contiene la *tabu list* della k-esima formica, e $\mathbf{tabu}_k(s)$ l's-esimo elemento della *tabu list* della k-esima formica, cioè la s-esima città visitata dalla k-esima formica.

Chiamiamo visibilità η_{ij} la quantità $1/d_{ij}$, e definiamo la probabilità di transizione dalla città i alla città j come segue

---

[1] Sebbene il nome scelto richiami il concetto di ***tabu search*** proposto in [51] and [52], il nostro approccio presenta rispetto a questo delle differenze sostanziali: (i) l'assenza di una funzione di aspirazione, (ii) il fatto che gli elementi che noi memorizziamo nella tabu list sono nodi del grafo e non permutazioni come nel caso del tabu search.

$$p_{ij}(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum\limits_{j \in J} [\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}} & \text{se } j \in J \\ 0 & \text{altrimenti} \end{cases} \tag{2.2}$$

dove $J = \{j: j \notin \mathbf{tabu}_k(t)\}$ e dove α e β sono parametri che permettono il controllo dell'importanza relativa della traccia rispetto alla visibilità. Pertanto la probabilità di transizione risulta essere un *trade-off* tra la visibilità (che dice che le città vicine dovrebbero essere scelte con alta probabilità) e l'intensità della traccia (che dice che se sull'arco (i,j) c'è stato nel passato molto traffico, allora quell'arco è molto desiderabile).

Scelte differenti relative a come calcolare $\Delta\tau_{ij}^{k}(t,t+1)$ e quando aggiornare il valore di $\tau_{ij}(t)$ determinano la realizzazione di algoritmi fra loro differenti. Nel capitolo 3 ne verranno presentati tre, chiamati **Ant-density**, **Ant-quantity** e **Ant-cycle**.

2.2.3 Caratteristiche del modello e risultati attesi

Il modello qui presentato, e che verrà esposto con maggiori dettagli nel capitolo 3, presenta alcune caratteristiche molto interessanti che esponiamo di seguito.

La prima è che si tratta di un sistema distribuito, nel quale ogni singola componente coopera alla risoluzione del problema in un modo originale: modificando la struttura del problema stesso. Infatti se consideriamo la matrice delle probabilità di transizione $\mathbf{p}(t)$, ogni suo elemento $p_{ij}(t)$ rappresenta la probabilità di transizione dalla città i alla città j. Allo stato iniziale, ponendo come detto un uguale valore di traccia su tutti gli archi, questa matrice riflette la struttura geometrica del problema. Infatti i valori $p_{ij}(0)$ sono proporzionali a η_{ij}, e perciò le città più vicine vengono scelte con probabilità più alta. Man mano che il processo evolve, ogni $p_{ij}(t)$ si modifica secondo le equazioni (1) e (2). Il processo perciò può essere visto come una deformazione dello spazio del problema, deformazione che procede rendendo più vicine le città fra le quali c'è molto traffico e, corrispondentemente, più lontane quelle fra le quali ce ne è poco. Dalle simulazioni, i cui risultati verranno presentati nel capitolo 3, abbiamo osservato che la matrice $\mathbf{p}(t)$ ha piccole oscillazioni intorno ad uno stato[2] che, nei limiti dell'approssimazione del presente lavoro, assumeremo come stato stazionario (con ciò intendiamo dire che le variazioni nella matrice di transizione sono molto piccole).

In questo stato si possono osservare due tipi di comportamento delle formiche. Nel caso meno frequente solo una probabilità di transizione è significativamente maggiore di zero in ogni riga/colonna e perciò tutte le formiche scelgono lo stesso arco ad ogni passo: quindi non vengono più esplorati nuovi cammini (chiamiamo questo tipo di situazione *uni-cammino*). Nella situazione più frequente invece la maggior parte delle righe hanno due (a volte anche più) probabilità di transizione con un valore significativamente maggiore di zero. In questi casi il processo di ricerca non si ferma, anche se la dimensione dello spazio di ricerca è fortemente ridotta (se raffrontata allo spazio iniziale). Si consideri per esempio la rappresentazione grafica di una matrice $\mathbf{p}(t)$ riportata in Fig.2.2. Questa matrice rappresenta le probabilità di transizione all'equilibrio per un problema con 10 città. Una formica nella città 1 ha una probabilità molto alta di andare

[2] Il processo stocastico che regola l'evoluzione della matrice $\mathbf{p}(t)$ è un processo Markoviano con memoria infinita

nella città 5 (circa il 50%) o di andare nella città 2 (circa il 35%), ed ha una probabilità molto bassa di scegliere una qualunque delle altre città. Una analisi simile può essere fatta per ognuna delle formiche nelle altre città (ad esempio dalle città 9 e 0 ogni destinazione è ugualmente probabile).

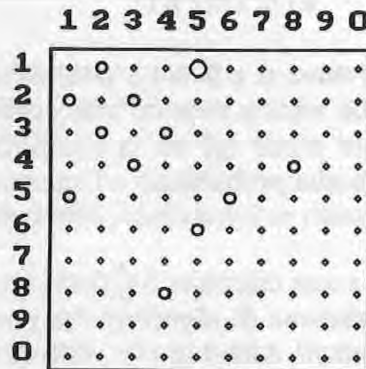


Fig.2.2 - Matrice delle probabilità di transizione

Un altro modo di interpretare il funzionamento dell'algoritmo è quello di immaginare che avvenga una specie di sovrapposizione degli effetti probabilistica: ogni formica, se isolata (cioè se $\alpha=0$), si muoverebbe con una regola di tipo *greedy* (tenderebbe cioè a scegliere la città più vicina fra quelle non ancora visitate) basandosi solo su informazione di tipo locale. Questa regola conduce a risultati di qualità molto scadente. Infatti i primi archi scelti dalla formica (quando la sua *tabu list* è ancora quasi vuota e quindi la formica è libera di muoversi come meglio crede) sono molto buoni, ma, man mano che la tabu list si riempie, le scelte diventano sempre più obbligate e gli ultimi passi inseriscono (in generale) degli archi molto cattivi (lunghi) che rendono il ciclo complessivamente poco buono. Abbiamo visto però che il ciclo seguito da una formica *greedy* è composto da alcune parti (quelle iniziali) che sono molto buone e da altre che non lo sono (quelle finali): se ora consideriamo l'effetto della presenza contemporanea di molte formiche, vediamo che sottoinsiemi di archi che appartengono a sottocammini buoni vengono seguiti da molte formiche (formiche che allo stato iniziale si trovano in città fra di loro vicine seguiranno con alta probabilità gli stessi insiemi di archi buoni), mentre gli archi cattivi (a causa dei vincoli posti dalla *tabu list*) che ogni formica sceglie per poter terminare un ciclo saranno in generale diversi da formica a formica. Perciò i sottoinsiemi buoni di archi riceveranno una più elevata quantità di traccia e verranno scelti nel futuro con probabilità più elevata.

Come ultima considerazione vorremmo far notare come il nostro sistema abbia alcune delle proprietà dei sistemi che apprendono in modo adattativo. Infatti la conoscenza acquisita nel processo di risoluzione del problema è memorizzata nella struttura del problema stesso ed è utilizzata per indirizzare il processo nei passi futuri. Il sistema impara delle buone strategie di ricerca e modifica la struttura del problema affinché questa conoscenza vi si rifletta. Le formiche, che all'istante iniziale sono "cieche", acquisiscono la capacità di prendere decisioni "furbe", riducendo la probabilità di ripetere errori fatti nel passato.

2.2.4 Caratteristiche naturali del sistema formiche

Come già detto, ogni algoritmo presentato in questa tesi appartiene alla classe degli algoritmi naturali. Le caratteristiche peculiari di questa classe sono state introdotte, insieme ad una tabella riassuntiva, nella sezione 2.1. In questa sottosezione (e così faremo nell'ultima sottosezione di 2.3, 2.4, 2.5, 2.6), illustriamo le caratteristiche che giustificano per ogni algoritmo l'appartenenza alla suddetta classe.

Il sistema formiche è un modello[3], sebbene opportunamente modificato per permettere la sua applicazione a problemi di ottimizzazione su grafo, di un sistema naturale: una colonia di formiche. Il sistema è intrinsecamente parallelo: non è difficile immaginare come distribuire il carico computazionale su di un calcolatore parallelo nel quale ogni processore corrisponde ad una o più formiche[4]. La stocasticità dell'algoritmo è data dalla forma probabilistica della regola di transizione (vedi formula 2.2), mentre l'adattatività risulta dalla capacità dell'algoritmo di fare uso di retroazione: tramite l'uso della retroazione, memorizzata nel sistema sotto forma di traccia, il SF può migliorare le proprie prestazioni. Una caratteristica molto importante del SF è che utilizza la retroazione in modo autocatalitico: la storia passata, per mezzo del meccanismo della traccia, influenza le decisioni future aumentando la probabilità che decisioni utili al sistema siano ripetute. Il SF, nella sua presente versione, non fa uso del concetto di evoluzione e selezione: algoritmi di ottimizzazione del tipo degli algoritmi genetici o delle strategie evolutive potranno essere utilizzati per arrivare ad una taratura automatica dei parametri che controllano il comportamento delle formiche (α e β).

[3] Quando in questa tesi diciamo che un certo sistema artificiale è un "modello" di un sistema naturale intendiamo dire che tale sistema naturale ha fornito l'ispirazione per la definizione del sistema artificiale (e non che il sistema artificiale sia una modellizzazione fedele del sistema naturale).

[4] Ovviamente, sebbene anche ad una osservazione superficiale sia subito evidente la parallelizzabilità del sistema formiche, la realizzazione pratica di tale parallelizzazione potrebbe scontrarsi con problemi dovuti alle limitazioni nelle capacità di comunicazione tra i vari processori e la memoria.

2.3 Algoritmi genetici

Gli algoritmi genetici (AG) sono un modello di calcolo che ha preso ispirazione dalla metafora della genetica delle popolazioni. In questo approccio infatti si ipotizza che il processo di ricerca della soluzione ottima in un problema di ottimizzazione possa essere guidato da una "forza" che tende a favorire le soluzioni migliori e da una "forza" esplorativa che propone soluzioni nuove. Nell'analogia con la genetica delle popolazioni, ogni soluzione è vista come un individuo e l'insieme di soluzioni considerate all'istante t viene detto popolazione. Le idee che dirigono il processo di ricerca sono l'equivalente della selezione naturale, del crossover e della mutazione.

La struttura dell'algoritmo è la seguente:

Algoritmo genetico

$\mathcal{P}(0) := (P_1, ..., P_n)$ {popolazione iniziale: è un multiinsieme};

Finché (condizione di terminazione) esegui

Genera $\mathcal{P}(t+1)$ da $\mathcal{P}(t)$ applicando l'operatore di *riproduzione* a $\mathcal{P}(t)$;

Genera $\mathcal{P}'(t+1)$ da $\mathcal{P}(t+1)$ applicando l'operatore di *crossover* a $\mathcal{P}(t+1)$;

Genera $\mathcal{P}''(t+1)$ da $\mathcal{P}'(t+1)$ applicando l'operatore di *mutazione* a $\mathcal{P}'(t+1)$;

$\mathcal{P}(t+1) := \mathcal{P}''(t+1)$;

$t := t+1$.

La condizione di terminazione può essere ad esempio un numero NC_{MAX} massimo di cicli o quando l'algoritmo non genera nuove soluzioni migliori per un numero fissato di cicli.

Si consideri ad esempio il seguente problema.

Supponiamo di voler risolvere il problema di ottimizzazione

$$\max f(x,y) = x^2-2xy+y^2$$

$$\text{con } 0 \leq x,y \leq q-1, \quad x,y \in \mathbb{N},$$

dove q è una potenza di 2 (per esempio, q=32): possiamo allora codificare x ed y con stringhe binarie di lunghezza $k=\log_2 q$.

Chiamiamo cromosoma la stringa composta dalle due sottostringhe codificanti rispettivamente x ed y. Ogni posizione nel cromosoma è detta *gene*. Il valore assunto da ogni gene è detto valore allelico o *allele*; per motivi legati alla performance dell'algoritmo (vedi per esempio [30]), i valori allelici sono di solito fatti variare sull'insieme {0,1}, detto *alfabeto allelico*. Supponiamo che ad ogni cromosoma sia possibile associare una *fitness function* (f.f.) che misura la bontà della soluzione considerata: nel nostro esempio la f.f. è rappresentato dalla funzione f(x,y). L'esempio è mostrato in figura 2.3.

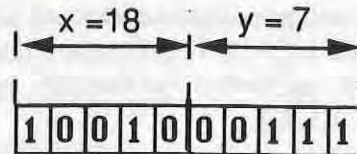


Fig.2.3 - Un cromosoma che rappresenta una soluzione con fitness function f(x,y)=324-252+49=121

Informalmente l'algoritmo funziona allora come segue.

Sia $\mathcal{P}(t)$ la popolazione di n cromosomi (*individui* di P) all'istante t. Sia $\mathcal{P}(0)$ la popolazione iniziale, generata casualmente. Il ciclo principale dell'AG genera una nuova popolazione $\mathcal{P}(t+1)$ a partire dalla vecchia popolazione $\mathcal{P}(t)$, applicando alcuni *operatori genetici*. Questi operatori (descritti più approfonditamente nel seguito) modificano alcuni individui scelti a caso nella popolazione $\mathcal{P}(t)$ e generano la popolazione $\mathcal{P}(t+1)$ garantendo una maggiore probabilità di riproduzione agli individui con valori di f.f. più alti. L'effetto complessivo dell'AG è quindi quello di spostare la popolazione $\mathcal{P}(t)$ verso le composizioni migliori: ciò corrisponde a dirigere la ricerca verso zone dello spazio delle soluzioni caratterizzate da più alti valori della f.f.

Il vantaggio computazionale ottenuto usando gli AG rispetto ad una ricerca casuale è dovuto al fatto che la ricerca è indirizzata dalla f.f. In particolare, si può operare in modo che l'indirizzamento non sia dovuto agli interi cromosomi, ma a loro parti fortemente correlate con alti valori della f.f.: queste parti sono dette *building blocks* [61], [54], [30]. È stato dimostrato (vedi ad esempio la sezione 2.3.3 oppure [61] o [10]) che gli AG elaborano ad ogni ciclo un numero di building blocks dell'ordine del cubo del numero degli individui della popolazione. Gli AG sono perciò vantaggiosi in tutti i casi in cui una soluzione ottima possa essere pensata come una collezione di building blocks.

2.3.1 Gli operatori genetici

In questa sottosezione introduciamo i tre operatori genetici principali: riproduzione, crossover e mutazione.

Riproduzione.

Nella nuova popolazione $\mathcal{P}(t+1)$, assegna un numero proporzionalmente crescente di copie agli individui che avevano una f.f. migliore della media nella vecchia popolazione $\mathcal{P}(t)$. Per applicare questo operatore, si calcola il valore della f.f. per ogni individuo e il valore totale per tutta la popolazione e si assegna ad ogni individuo h della popolazione $\mathcal{P}(t)$ una probabilità di riproduzione $p_r(h)$ pari alla sua f.f. divisa per la fitness totale della popolazione. Applicando poi n volte il metodo Montecarlo, si crea la nuova popolazione $\mathcal{P}(t+1)$ riproducendo ogni individuo h della popolazione $\mathcal{P}(t)$ con una probabilità uguale alla sua probabilità $p_r(h)$:

$$p_r(h) = \frac{f(h)}{\sum_{h=1}^{n} f(h)}$$

dove f(h) è il valore della f.f. dell'individuo h.

Per esempio, considerando una popolazione formata da 4 individui (chiamati 1,2,3,4) aventi f.f. rispettivamente pari a 10, 5, 4, 1, la fitness totale della popolazione è pari a 20; essendo $p_r(1)=0.5$, $p_r(2)=0.25$, $p_r(3)=0.2$, $p_r(4)=0.05$, la nuova popolazione potrebbe essere formata dagli individui 1,1,2,3.

Crossover.

Viene attivato con una probabilità p_c, indipendente dagli individui su cui agisce. Come ingresso sceglie a caso due individui diversi (genitori) e li combina, generando così due figli. La combinazione è effettuata scegliendo (con probabilità uniformemente distribuita) un "punto di taglio" nei due cromosomi rappresentanti i genitori: il taglio determina la scissione di ognuno dei due cromosomi in due parti; queste vengono poi scambiate e ricombinate per formare i due figli, come mostrato in figura 2.4. Questo operatore, che è l'elemento centrale negli algoritmi genetici, permette di giustapporre eventuali *building blocks* che prima erano presenti separatamente nei due genitori, ottenendo così figli almeno uno dei quali ha normalmente un valore di f.f. (molto) più alto.

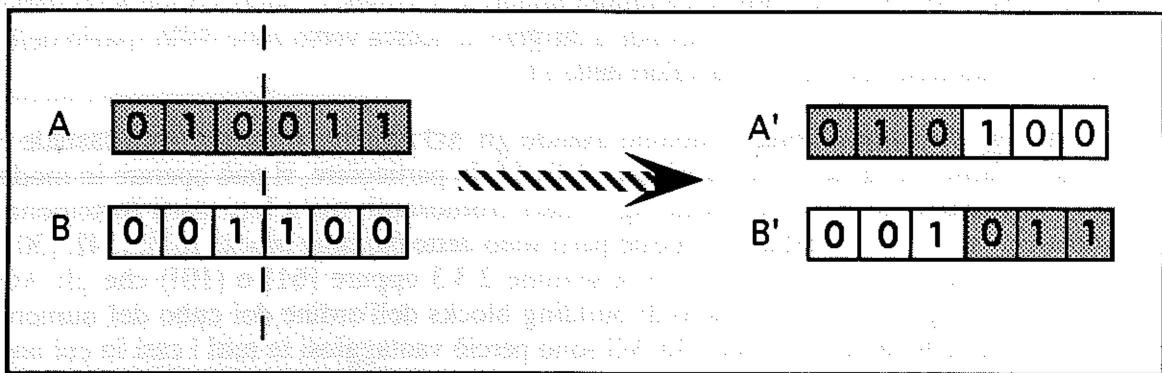


Fig.2.4 - I due cromosomi (genitori) A e B vengono accoppiati tramite crossover per ottenere i due nuovi cromosomi (figli) A' e B'.

Mutazione.

Causa, con probabilità p_m, il cambiamento del valore allelico di un gene scelto a caso. Ad esempio, se l'alfabeto è {0,1}, un allele che vale 0 si modifica in 1 e viceversa. Questo operatore introduce variazioni "elementari" nella popolazione e garantisce la possibilità di esplorare l'intero spazio di ricerca indipendentemente dalla specifica popolazione iniziale.

2.3.2 Ammissibilità dei cromosomi.

Il tentativo di estendere gli AG alla soluzione di problemi *vincolati* di ottimizzazione combinatoria presenta alcune difficoltà, la più rilevante delle quali consiste nel fatto che gli operatori di mutazione e di crossover prima definiti non sono più accettabili, in quanto possono generare soluzioni (cromosomi) non ammissibili.

Consideriamo ad esempio il TSP[5]: per questo problema la codifica più semplice di una soluzione consiste in un cromosoma nel quale ogni gene assume il valore di una città e queste sono elencate nello stesso ordine in cui devono essere visitate dal commesso

[5] Definito nella sezione 2.2.2

viaggiatore. Per esempio, in figura 2.5 il cromosoma A corrisponde a un viaggio che comincia nella città a e che continua attraverso le città b, c, ... finché viene raggiunta la città f (sottintendendo il successivo ritorno nella città a). Applicando l'operatore di crossover a questo caso si generano quasi certamente individui non ammissibili.

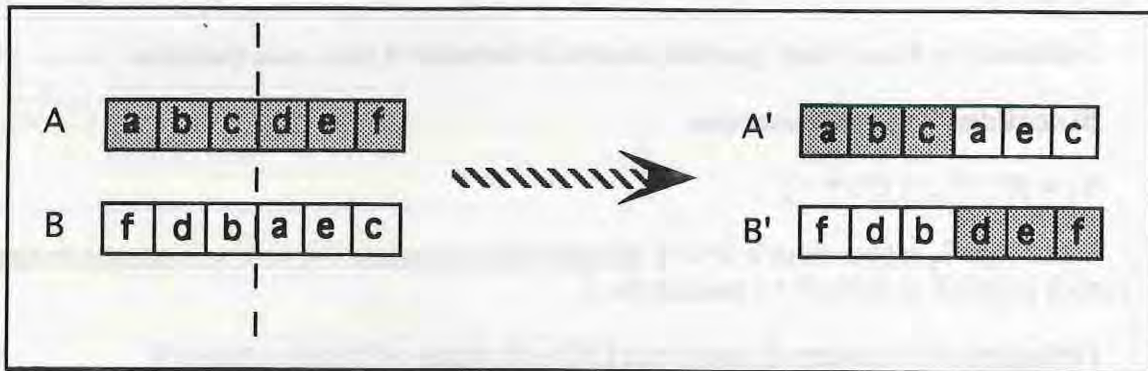


Fig.2.5 - Il problema del commesso viaggiatore:
(a sinistra) due individui ammissibili A e B vengono accoppiati;
(a destra) il crossover genera due soluzioni A' e B' non ammissibili.

Al problema delle soluzioni non ammissibili sono state date le seguenti risposte:

i) cambiare le modalità di rappresentazione della soluzione, in modo da poter applicare gli operatori standard;
ii) definire nuovi operatori di mutazione e di crossover che generino solo soluzioni ammissibili;
iii) applicare gli operatori standard e quindi effettuare un *genetic repair*, cioè modificare le soluzioni non ammissibili ottenute tramite un opportuno *algoritmo di filtro*.

Nel caso del TSP, gli approcci più efficaci sono stati l'introduzione di nuovi operatori [53] e l'applicazione del genetic repair [69].

Segnaliamo infine che (come si può vedere dall'esempio precedente) i valori allelici di un gene non devono necessariamente appartenere all'insieme (alfabeto) $\{0,1\}$: in generale, qualsiasi alfabeto può essere usato, ma l'efficienza dell'algoritmo è fortemente correlata alla cardinalità dell'alfabeto stesso [61], [3].

2.3.3 Risultati teorici fondamentali

In questa sezione riportiamo i due risultati teorici fondamentali sugli algoritmi genetici: il teorema sul parallelismo implicito e il teorema fondamentale degli algoritmi genetici (Schema Theorem).

Diamo per prima cosa alcune definizioni.

Sia $\mathcal{M}=<\omega_1, ..., \omega_n>$ una popolazione di n stringhe $\omega_i \in \{0,1\}^k$ generate mediante estrazioni casuali e indipendenti.

Sia XOVER($\mathcal{M}$) la popolazione ottenuta dalla $\mathcal{M}$ mediante l'applicazione dell'operatore di crossover.
Sia $S \in \{0,1,*\}^k$ uno schema. Dato S, con $\mathbf{S}$ indichiamo l'insieme delle stringhe che si possono ottenere da S mediante sostituzione di tutte le occorrenze del simbolo $*$ con i simboli 0,1 in tutti i modi possibili: $\mathbf{S}=\{\omega \mid \omega$ ottenuto da S con la sostituzione $* \rightarrow 0,1\}$.

Data una popolazione $\mathcal{M}'$, diremo che $\mathcal{M}'$ contiene una istanza di S se $\mathcal{M}' \cap S \neq \emptyset$.

Si dice che, in seguito all'applicazione dell'operatore di crossover, lo schema S si propaga se $(\mathcal{M} \cap S \neq \emptyset)$ AND $(XOVER(\mathcal{M}) \cap S \neq \emptyset)$.

I simboli 0 e 1 sono detti *specifici*, mentre il simbolo * è detto *non specifico.*

Si consideri il seguente esempio:

$S_1 = (0\ *\ *\ 1\ 1\ 0\ 0\ *\ *)$

La stringa S_1 rappresenta le $2^4=16$ stringhe che si possono ottenere sostituendo in tutti i modi possibili ai simboli * i simboli 0 e 1.

Definiamo ora i concetti di *lunghezza* L(S) e di *ordine* o(S) dello schema S.

L(S) è la distanza tra il primo e l'ultimo simbolo specifico della stringa S. Nell'esempio $L(S_1)$ vale 6.

o(S) è il numero di simboli specifici nella stringa S. Nell'esempio $o(S_1)$ vale 5.

Il teorema sul parallelismo implicito è stato proposto da Holland e la sua dimostrazione può essere trovata in [61] o in [10]). Nel seguito proponiamo un enunciato e una dimostrazione originali che, a giudizio dell'autore, rimuovono alcuni errori presenti nelle citate versioni di Holland.

Teorema del parallelismo implicito

Fissato un valore $\varepsilon = \frac{2l}{k}$, in una popolazione $\mathcal{M}$ di $n=c_1 2^l$ individui, ottenuta tramite estrazioni casuali e indipendenti dall'insieme delle stringhe $\{0,1\}^k$, si ha che il valore atteso del numero di schemi che si propaga con probabilità di essere distrutto dall'applicazione del crossover $\leq \varepsilon$ è almeno $\binom{2l}{l} \cdot 2^l \cdot (1-e^{-c_1}) \sim \left(\frac{n}{c_1}\right)^3 \cdot (1-e^{-c_1}) \cdot \frac{1}{\sqrt{\pi l}}$.
Inoltre con probabilità $P \geq (1-2e^{-c_1})$ il numero di schemi che si propaga è almeno $\frac{1}{2} \cdot \left(\frac{n}{c_1}\right)^3 \cdot (1-e^{-c_1}) \cdot \frac{1}{\sqrt{\pi l}}$.

Dimostrazione

Si consideri una finestra di $2l$ posizioni contigue sulla stringa $\omega \in \mathcal{M}$, tale che $\frac{2l}{k} = \varepsilon$. Chiaramente ogni schema che abbia le sue posizioni definitorie all'interno di questa finestra sarà soggetto ad un errore di trascrizione minore o uguale ad ε.

In questa finestra è possibile scegliere $\binom{2l}{l}$ differenti combinazioni di l posizioni definitorie e per ognuna di queste sono possibili 2^l differenti schemi. Il numero totale di schemi definiti su l posizioni nella finestra è quindi

$$L=\binom{2l}{l}\cdot 2^l \sim \left(\frac{n}{c_1}\right)^3 \cdot \frac{1}{\sqrt{\pi l}} \qquad (l \to \infty)$$

Sia $\{S_1, ..., S_L\}$ l'insieme degli schemi con l posizioni definitorie nella finestra.

Sia X_{S_i} una variabile aleatoria definita da

$$\mathsf{X}_{S_i} = \mathsf{X}_{S_i}(<\omega_1, ..., \omega_n>) = \begin{cases} 1 & \exists \omega_j \in S_i \\ 0 & \forall \omega_j \ (\omega_j \notin S_i) \end{cases}$$

Definita la variabile aleatoria $\mathsf{X}^l = \sum_{i=1}^{L} \mathsf{X}_{S_i}$, ne segue che

$$\mathsf{X}^l(<\omega_1, ..., \omega_n>) = \sum_{i=1}^{L} \mathsf{X}_{S_i}(<\omega_1, ..., \omega_n>)$$

rappresenta il numero di schemi distinti presenti in $\mathcal{M}=<\omega_1, ..., \omega_n>$.

Il valore atteso del numero di schemi con l posizioni definitorie nella finestra rappresentati in $<\omega_1, ..., \omega_n>$ è dato da

$$E(\mathsf{X}^l) = \sum_{i=1}^{L} E(\mathsf{X}_{S_i})$$

e poiché il valore atteso $E(\mathsf{X}_{S_i})$ è uguale, per ovvie ragioni di simmetria, per tutti gli S_i, possiamo scrivere

$$E(\mathsf{X}^l) = \sum_{i=1}^{L} E(\mathsf{X}_{S_i}) = \binom{2l}{l}\cdot 2^l \cdot E(\mathsf{X}_{S_1})$$

Per calcolare il valore di $E(\mathsf{X}_{S_1})$ si consideri, senza perdita di generalità, lo schema S_1 definito sulle prime l posizioni definitorie. Sia $S_1 = [b_1, ..., b_i, ..., b_l, *, ..., *, ..., *]$, dove b_i è il valore della i-esima posizione definitoria. Si ha allora che

$$E(\mathsf{X}_{S_1}) = \mathrm{Prob}(<\omega_1, ..., \omega_n> \mid \exists k \ \ \omega_k = b) = 1 - \mathrm{Prob}(<\omega_1, ..., \omega_n> \mid \omega_1 \neq b, ..., \omega_n \neq b)$$

che, per l'indipendenza delle estrazioni, si può scrivere come segue

$$E(\mathsf{X}_{S_1}) = 1 - \prod_{i=1}^{n} \mathrm{Prob}(\omega_i \neq b) = 1 - \left(\frac{2^l - 1}{2^l}\right)^n = 1 - \left(1 - \frac{1}{2^l}\right)^n$$

Poiché, per x reale, $1-x \leq e^{-x}$, concludiamo che $\left(1-\frac{1}{2^l}\right)^n \leq e^{\frac{-n}{2^l}} = e^{-c_1}$, poiché nel nostro caso $n=c_1 \cdot 2^l$.

Si ha allora

$$E(\mathsf{X}^l) \geq \binom{2l}{l}\cdot 2^l \cdot (1-e^{-c_1}) \sim \left(\frac{n}{c_1}\right)^3 \cdot \frac{1}{\sqrt{\pi l}} (1-e^{-c_1}) \qquad (l \to \infty)$$

Calcoliamo ora la probabilità con la quale si verifica l'evento

$$Q = \{\mathsf{X} \geq \frac{1}{2} \cdot \left(\frac{n}{c_1}\right)^3 \cdot (1-e^{-c_1}) \cdot \frac{1}{\sqrt{\pi l}}\}.$$

Ponendo per comodità di notazione

$a = (1 - e^{-c_1})$, $A = \binom{2l}{l} \cdot 2^l$

segue allora:

$E(\chi^l) \geq a \cdot A$

Il nostro problema è allora: qual'è la probabilità $P(Q)$ dell'evento $Q = \{\chi \geq \frac{a.A}{2}\}$?

Dato che $\chi^l \leq A$

$$a \cdot A = E(\chi^l) = \int \chi^l d\mu = \int_Q \chi^l d\mu + \int_{Q^c} \chi^l d\mu \leq A \cdot \int_Q d\mu + \frac{a \cdot A}{2} \cdot \int_{Q^c} d\mu =$$

$$A \cdot P(Q) + \frac{a.A}{2} \cdot (1 - P(Q))$$

da cui

$$a \leq P(Q) + \frac{a}{2} \cdot (1 - P(Q))$$

$$P(Q) \geq \frac{1 - e^{-c_1}}{1 + e^{-c_1}} \geq 1 - 2e^{-c_1} \qquad \blacksquare$$

La probabilità dell'evento Q per diversi valori di c_1 è data in tabella 2.2

Tabella 2.2 - Probabilità dell'evento Q in funzione di c_1

c_1	$\frac{1 - e^{-c_1}}{1 + e^{-c_1}}$	$1 - 2e^{-c_1}$
1	0.46	0.26
2	0.76	0.73
3	0.91	0.90
4	0.96	0.96

Si deve inoltre tenere conto del fatto che vi sono $k-2l+1$ modi di scegliere una finestra di $2l$ posizioni contigue su una stringa di lunghezza k.

Rispetto al teorema di Holland (vedi [10]), le differenze fondamentali sono:

- il limite inferiore è un valore atteso,

- viene calcolata la probabilità che il numero di schemi effettivamente propagati sia superiore alla metà del valore atteso.

Si può migliorare il lower bound ottenuto da Holland ragionando come segue.

Il motivo per cui Holland sceglie di contare gli schemi con *l* posizioni definitorie nella finestra di dimensioni *2l* è dovuto al fatto che il numero di differenti combinazioni di x posizioni definitorie nella finestra di dimensioni *2l* è dato dalla quantità $\binom{2l}{x}$, che ha un massimo per x=*l*. In realtà la quantità di cui ci interessa calcolare il massimo è $\binom{2l}{x} \cdot 2^x$, quantità che rappresenta il numero di schemi differenti definibili su x posizioni definitorie. Questa quantità raggiunge il valore massimo per $x = \frac{4l}{3} - \frac{1}{3}$. Si ha pertanto che, trascurando la costante, il numero massimo di schemi differenti definibile nella finestra *2l* è dato da:

$$L_1 = \binom{2l}{4l/3} \cdot 2^{4l/3}$$

che, utilizzando la formula di Stirling ($n! = \sqrt{\pi n} \cdot n^n \cdot e^{-n}$), può essere trasformato in

$$\frac{3}{2 \cdot \sqrt{2\pi l}} \cdot 2^{2l \cdot \log_2 3} \approx \frac{3}{2 \cdot \sqrt{2\pi l}} \cdot 2^{3,17 \cdot l}$$

Pertanto, con un analisi del tutto equivalente a quella fatta precedentemente, pur di sostituire L con L_1, si ottiene

$$E(\chi^l) \geq \binom{2l}{4l/3} \cdot 2^{4l/3} \cdot (1 - e^{-c_1}) \sim \left(\frac{n}{c_1}\right)^{3,17} \cdot \frac{1}{\sqrt{\pi l}} \cdot (1 - e^{-c_1}) \qquad (l \to \infty).$$

Teorema fondamentale degli algoritmi genetici (Schema Theorem)

Esaminiamo ora l'effetto che gli operatori genetici hanno sulla propagazione di un generico schema H di lunghezza L(H) e ordine o(H). Al risultato ottenuto si da il nome di Schema Theorem.

Sia $\mathcal{P}(t)$ la popolazione di n individui all'istante t e w(H,t) il numero di occorrenze dello schema H nella popolazione $\mathcal{P}(t)$.

<u>*Effetto dell'operatore di riproduzione sullo schema H*</u>

L'effetto dell'operatore di riproduzione sul numero di occorrenze dello schema H è il seguente:

$$w(H,t+1) = w(H,t) \cdot n \cdot \frac{f(H)}{\sum_{i=1}^{n} f_i} \qquad (2.3)$$

dove f_i è la fitness dell'i-esimo individuo della popolazione $\mathcal{P}(t)$; la (2.3) si può anche scrivere come:

$$w(H,t+1) = w(H,t)\cdot\frac{f(H)}{\bar{f}} \qquad (2.4)$$

dove $\bar{f}$ è la fitness media della popolazione: $\bar{f} = \frac{\sum_{i=1}^{n} f_i}{n}$ e f(H) è una stima[6] della fitness dello schema H (questa stima può essere ottenuta ad esempio come valor medio della fitness di tutti gli individui della popolazione che contengono lo schema H).

Ciò significa che, in seguito all'applicazione dell'operatore di riproduzione, il valore atteso del numero degli schemi con fitness al di sopra della media aumenterà passando dalla generazione t alla generazione t+1.

Dato che l'equazione (2.4) può facilmente essere trasformata in una equazione esponenziale in t (nel discreto), possiamo dire che l'operatore di riproduzione fa aumentare il numero delle occorrenze degli schemi con fitness al di sopra della media con una velocità esponenziale nel numero di generazioni.

Effetto dell'operatore di crossover sullo schema H

L'operatore di crossover, se il punto di crossover è scelto con probabilità uniformemente distribuita p_c, distrugge uno schema H di lunghezza L(H) con probabilità $p_d = \frac{L(H)}{(k-1)}$ (dove k è la lunghezza della stringa).

La probabilità che lo schema H sopravviva al crossover è $p_s = 1 - p_c p_d$.

Pertanto l'effetto combinato di riproduzione e crossover è dato dalla formula seguente:

$$w(H,t+1) = w(H,t)\,\frac{f(H)}{\bar{f}}\left[1 - p_c\frac{L(H)}{k-1}\right] \qquad (2.5)$$

Effetto dell'operatore di mutazione sullo schema H

Detta p_m la probabilità di mutazione di un elemento specifico (cioè diverso da *) appartenente ad uno schema H di ordine o(H), la probabilità che lo schema H non venga distrutto dall'operatore mutazione è $(1-p_m)^{o(H)}$ che, nel caso il valore di p_m sia molto piccolo, può essere approssimata dall'espressione

$$(1-p_m)^{o(H)} \approx 1 - o(H)p_m \qquad (2.6)$$

Pertanto l'effetto combinato di riproduzione, crossover e mutazione risulta essere:

[6] Questa stima diviene sempre meno accurata con la convergenza dell'algoritmo verso popolazioni che rappresentano zone "buone" dello spazio di ricerca. Infatti mentre all'inizio la popolazione di soluzioni è uniformemente distribuita, il procedere del processo di ottimizzazione fa sì che nella popolazione la distribuzione degli schemi non sia più uniforme.

$$w(H,t+1) = w(H,t)\,\frac{f(H)}{\bar{f}}\left[1 - p_c\,\frac{L(H)}{k-1} - o(H)p_m\right] \qquad (2.7)$$

Questa formula è detta Teorema fondamentale degli algoritmi genetici. La forma di questa espressione è quella di una esponenziale e ci dice quindi che gli schemi con valore di fitness stimata superiore alla media crescono, in numero, con velocità esponenziale. In realtà ciò può essere vero solo nelle fasi iniziali dell'algoritmo, quando la stima è fatta su una popolazione di stringhe generate in modo uniformemente casuale. Man mano che l'algoritmo procede infatti la popolazione converge verso zone dello spazio di ricerca che contengono soluzioni "più buone" e quindi tale stima diviene meno significativa. Questo potrebbe anche spiegare il fatto che l'algoritmo migliora la fitness delle soluzioni molto rapidamente nelle fasi iniziali della ricerca per poi rallentare nelle fasi finali di raffinamento.

2.3.4 Considerazioni conclusive

La teoria presentata nelle sezioni precedenti mostra da un lato la potenza e dall'altro i punti deboli degli algoritmi genetici. Gli aspetti positivi sono così riassumibili:

(i) parallelismo implicito: la quantità di informazione trattata è dell'ordine della terza potenza del numero di individui effettivamente processati;
(ii) parallelismo esplicito: la popolazione di individui può essere facilmente distribuita su molti processori che lavorano in parallelo scambiandosi messaggi sull'andamento della computazione in modo sincrono o asincrono;
(iii) facilità di adattamento dell'algoritmo alla risoluzione di una vasta classe di problemi (vedi ad esempio la sezione 2.4 nella quale gli AG vengono utilizzati come meccanismo di scoperta di nuove regole in un sistema di apprendimento automatico).

Molti però sono i problemi ancora aperti; fra di essi hanno particolare importanza i seguenti:

(a) il criterio da seguire per assegnare valori adeguati ai numerosi parametri che regolano il funzionamento degli AG (ad esempio la probabilità di mutazione, di crossover, ecc.; vedi anche [58]);
(b) il possibile utilizzo degli AG in problemi di ottimizzazione combinatoria (in questa tesi tentiamo di dare una risposta parziale a questo quesito, applicando gli AG al problema dell'orario scolastico; per altri approcci vedi [28]);
(c) il modo di mappare le variabili continue su uno spazio di ricerca discreto (cioè, come devono essere codificate le stringhe nel caso si voglia ottimizzare una funzione di variabili reali?). Questo problema è stata affrontato in modo autonomo da ricercatori tedeschi che hanno sviluppato un algoritmo simile agli AG che va sotto il nome di *Evolution Strategie* (Strategie evolutive) e che verrà presentato nella sezione 2.5.

2.3.5 Caratteristiche naturali degli algoritmi genetici

Gli algoritmi genetici sono una modellizzazione del processo che sta alla base dell'evoluzione delle specie. Le forme di parallelismo presentate dall'algoritmo sono due: quella classica nella quale si può immaginare che molte popolazioni di individui evolvano contemporaneamente con sovrapposizioni parziali (migrazioni di individui da una popolazione all'altra), e quella data dal parallelismo implicito presentata in 2.3.3. Il fatto che ogni operatore venga applicato con una certa probabilità e che la riproduzione favorisca, probabilisticamente, gli individui migliori, rende l'AG un sistema stocastico. La retroazione positiva è ottenuta per mezzo dei meccanismi di riproduzione e crossover combinati (vedi formula 2.7). Ovviamente entrambi i concetti di evoluzione e selezione sono utilizzati dagli AG.

2.4 Sistemi a classificatori

I sistemi a classificatori (SC) sono un modello di calcolo che può essere utilizzato per applicazioni di apprendimento automatico [88]. Come detto nella sezione 1.1 ci sono problemi per i quali non si conosce una procedura di soluzione che trovi la soluzione ottima in tempi ragionevoli, ma che sono risolti molto velocemente, e in modo *soddisfacente*, dagli esseri viventi. Con l'espressione "in modo soddisfacente" intendiamo dire che la soluzione trovata, pur non essendo ottima, permette la sopravvivenza del sistema che l'ha generata[7]. I sistemi a classificatori si pongono come obiettivo quello di creare una base di conoscenza che permetta ad una "creatura artificiale" di sopravvivere in un ambiente dato: tale obiettivo viene perseguito per mezzo di un algoritmo adattativo che modifica il suo comportamento a seconda dei risultati ottenuti nell'interazione con l'ambiente con il quale la creatura artificiale interagisce. La valutazione delle azioni eseguite dalla creatura artificiale viene fatta per mezzo di una funzione di valutazione che restituisce premi o punizioni a seconda che le mosse proposte dal sistema di apprendimento siano coerenti o meno con gli obiettivi. I sistemi a classificatori appartengono pertanto ai sistemi di apprendimento automatico con supervisore (supervised reinforcement learning [16]).

Da questa breve introduzione è chiaro che la complessità dei problemi che si vogliono affrontare utilizzando i SC è molto maggiore di quella dei tipici problemi di ottimizzazione combinatoria. Nel seguito di questa sezione introdurremo le caratteristiche principali dei SC, mentre una applicazione di questo modello di calcolo ad un problema di robotica verrà presentato nel capitolo 5.

2.4.1 Un problema

Si consideri il problema di un robot autonomo che deve imparare a seguire una sorgente luminosa (questo problema verrà ampiamente ripreso nel capitolo 5). Il robot interagisce con l'ambiente per mezzo di sensori di luce in input e per mezzo di azioni motorie in output.

Una soluzione a questo problema può essere descritta come segue: allo stato iniziale viene creata, in modo del tutto casuale, una base di conoscenza che rappresenta le capacità possedute dal robot. Con meccanismi che verranno illustrati in seguito, il robot sceglie una azione da compiere fra quelle suggerite dalla sua conoscenza. Il risultato di tale azione viene valutato dalla funzione di valutazione che premierà o punirà la base di conoscenza in base alla azione scelta.

Operando una scomposizione funzionale, i moduli necessari per realizzare questo tipo di sistema di apprendimento sono:

- il sistema che gestisce l'interazione con l'ambiente (sistema di performance),
- il sistema di creazione della base di conoscenza,
- il sistema di valutazione della base di conoscenza,
- la funzione di valutazione.

[7] Si veda anche il concetto di viabilità di un sistema introdotto da F.Varela.

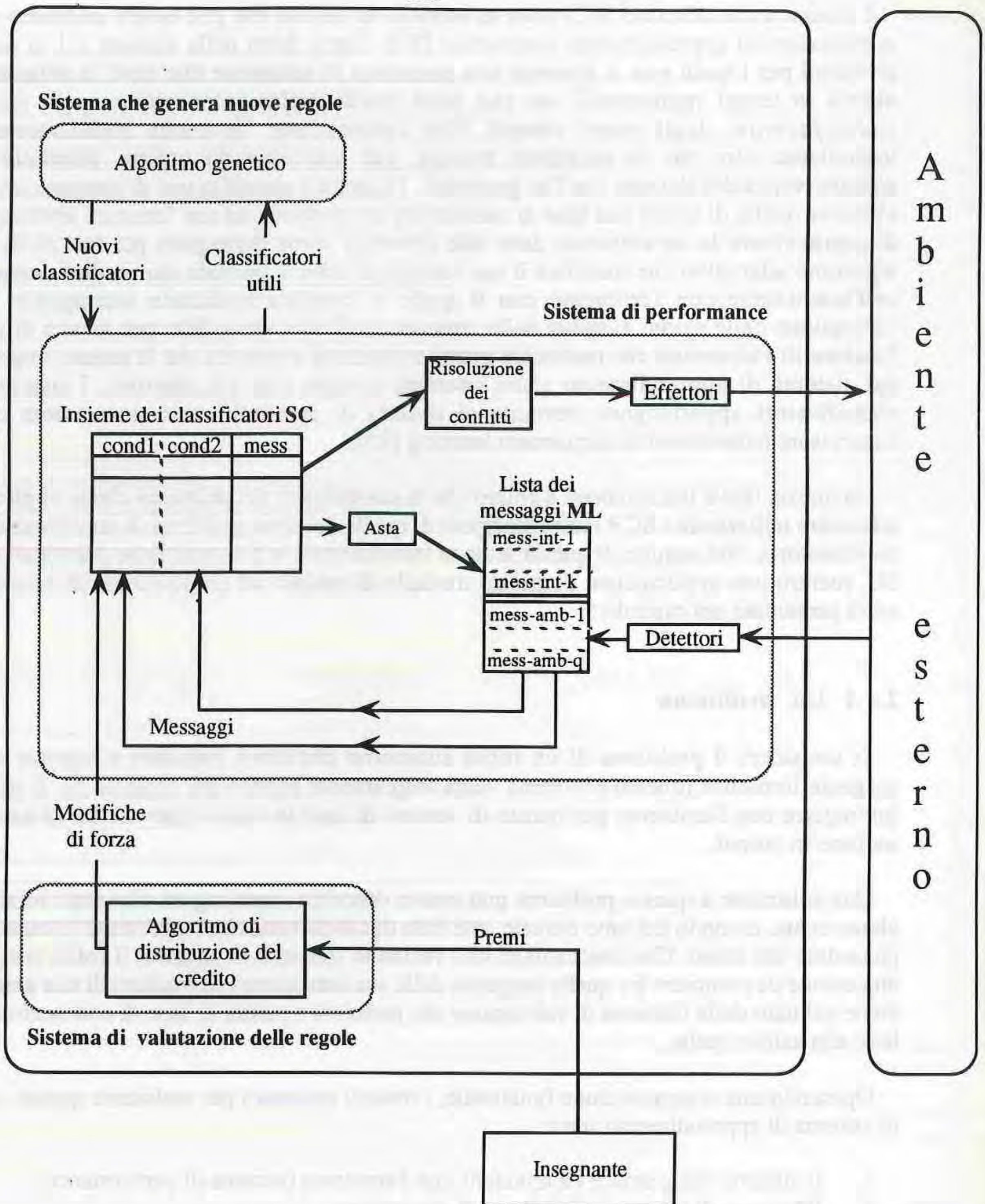


Figura 2.6 - Sistema a classificatori e sue componenti

È inoltre necessario disporre di un ambiente nel quale far vivere la creatura artificiale. Questo ambiente è in generale un ambiente simulato, ma può essere l'ambiente vero nel caso si utilizzi un robot reale.

Nei sistemi a classificatori la base di conoscenza è realizzata per mezzo di un insieme di regole di produzione [71], [70]. L'algoritmo che gestisce le regole, detto algoritmo di performance, permette l'attivazione simultanea di più regole. Affinché il sistema presenti capacità di apprendimento, sia cioè in grado di modificare la base di regole in modo tale da adattarle alle caratteristiche dell'ambiente con il quale il sistema interagisce e da perseguire gli obiettivi posti dalla funzione di valutazione, vengono introdotti algoritmi di creazione di nuove regole e di valutazione delle regole esistenti; il primo tipo di algoritmo utilizza le regole esistenti per crearne di nuove che abbiano alta probabilità di risultare utili, mentre il secondo tipo, che funziona in modo strettamente collegato con quello di performance, ha il compito di valutare l'effettivo grado di utilità delle regole. Questo grado di utilità viene misurato con un valore numerico reale detto *forza,* che risulta indispensabile per decidere quali regole debbano essere effettivamente utilizzate, sia nel caso che diverse regole propongano azioni fra di loro incompatibili (ad es. "vai avanti" e "vai indietro"), che nel caso in cui ve ne siano troppe attivate simultaneamente (questo secondo aspetto è legato alla necessità di limitare la quantità di memoria utilizzata per gestire le azioni proposte dalle regole e favorisce l'insorgere di un meccanismo implicito di mantenimento della coerenza, nel quale regole che hanno un basso valore di forza vengono attivate con minore probabilità). Nelle sezioni seguenti presenteremo il funzionamento in dettaglio dei sistemi che costituiscono un SC.

2.4.2 Il sistema di performance

Il sistema di performance è un sistema a regole di produzione parallelo.

Una regola (o classificatore) è una tupla $(K_1, ..., K_n, M)$ dove $n \geq 2$, K_i è una stringa di lunghezza N su $\{0,1,\#\}^*$ ed M è una stringa di lunghezza N su $\{0,1\}^*$.

Informalmente $K_1, ..., K_n$ è la congiunzione di n condizioni. Ogni K_i identifica uno schema (linguaggio), cioè l'insieme di tutte le stringhe ottenibili da K_i operando la sostituzione $(\# \rightarrow 0,1)$ in tutti i modi possibili[8]. M è la parte azione della regola (anche detta messaggio). Ad ogni regola i-esima all'istante t è associato il valore $\mathbf{For}_i^{(t)} \in \mathbb{R}$, che misura l'utilità della regola al tempo t.

L'operazione principale è $MATCH(K_i,M') \equiv M' \in K_i$.

Le strutture dati utilizzate dall'algoritmo di performance sono:

- il multiinsieme $\mathbf{SC} = \{C_1, ... , C_i, ... , C_p\}$, dove C_i è una regola (SC è la base di conoscenza del sistema),
- la lista dei messaggi $\mathbf{ML} = \mathbf{MLI} \cup \mathbf{MLE}$, dove $\mathbf{MLI} = (M_1, ... , M_k)$ è la lista dei messaggi interni[9], $\mathbf{MLE} = (M_1, ... , M_q)$ è la lista dei messaggi esterni[10] e M_j è un messaggio,
- l'insieme **SCA** i cui elementi sono le regole in cui ogni condizione è verificata da almeno un messaggio.

[8] Il simbolo # viene chiamato don't care. E convenzione in letteratura utilizzare il simbolo * nel caso l'oggetto siano gli AG e il simbolo # nel caso si tratti di SC, sebbene il significato dei due simboli sia del tutto equivalente.

[9] I messaggi generati da regole attive al passo precedente sono detti messaggi interni, quelli che arrivano dall'ambiente sono detti messaggi esterni.

[10] Il sistema è progettato in modo tale da avere un numero di messaggi esterni sempre $\leq$ q.

La struttura dell'algoritmo è la seguente.

Algoritmo di performance

Fase di inizializzazione

Attribuisci un valore iniziale di forza F(C) ad ogni classificatore C.
Leggi i messaggi ambientali e appendili a **MLE**.
Esegui l'operazione di MATCH tra i messaggi in **MLE** e la parte condizione delle regole in **SC**.
SCA := regole $C_1, ..., C_r$ in cui ogni condizione è verificata da almeno un messaggio.
Estrai un k-campione $\mathcal{A}=[C_1, ..., C_k]$ da **SCA** in modo indipendente con $Prob(C) = F(C)/\Sigma F(C_i)$ {F(C) è la forza del classificatore C}.
Metti in **MLI** i messaggi $[M_1, ..., M_k]$ del k-campione.
Se vi sono messaggi nelle regole attive diretti agli attuatori (cioè verso l'ambiente esterno), effettua le azioni proposte ed eventualmente chiama il modulo "risolutore dei conflitti".
Cancella tutti i messaggi presenti in **MLE**.

Run forever

Leggi i messaggi ambientali e appendili a **MLE**.
Esegui l'operazione di MATCH tra i messaggi in **ML** e la parte condizione delle regole in **SC**.
SCA := $\emptyset$.
SCA := regole $C_1, ..., C_r$ in cui ogni condizione è verificata da almeno un messaggio.
Se vi sono messaggi nelle regole attive diretti agli attuatori (cioè verso l'ambiente esterno), effettua le azioni proposte ed eventualmente chiama il modulo "risolutore dei conflitti".
Cancella tutti i messaggi presenti in **ML**.
Estrai un k-campione $\mathcal{A}=[C_1, ..., C_k]$ da **SCA** in modo indipendente con $Prob(C) = F(C)/\Sigma F(C_i)$ {F(C) è la forza del classificatore C}.
Metti in **MLI** i messaggi $[M_1, ..., M_k]$ del k-campione.

Nella fase di inizializzazione delle strutture dati viene creato l'insieme delle regole **SC** secondo una qualche modalità di generazione che può tenere in considerazione la struttura del problema.

Il modulo risolutore dei conflitti effettua una competizione fra le diverse azioni proposte dando probabilità di vittoria più alta a quelle azioni che sono state proposte da regole più forti.

2.4.3 Il sistema di valutazione delle regole

L'algoritmo di distribuzione del credito ha il compito di attribuire ad ogni classificatore in SC un valore di forza proporzionale alla sua utilità. Il problema, di per se non semplice, è ulteriormente complicato dalle seguenti considerazioni.

L'utilità di una regola dipende dal contesto, per cui la sua forza dovrebbe essere dipendente oltre che dalla situazione ambientale in cui il sistema si trova, anche dallo stato interno del sistema.

I classificatori dovrebbero organizzarsi in strutture che permettano sia una partizione dello spazio degli stati ambientali in classi di equivalenza[11] coerenti con la funzione obiettivo, che il formarsi di catene di regole, permettendo così lo svilupparsi di memoria e la creazione di "modelli interni del mondo", dove con modello del mondo intendiamo una strutturazione delle regole che fornisca una performance del sistema soddisfacente in presenza di regolarità ambientali [9], [88].

Presentiamo l'algoritmo più utilizzato per l'apprendimento dei valori delle forze; questo algoritmo è stato introdotto da Holland [61]. Dato che la modifica delle forze avviene durante l'esecuzione dell'algoritmo di performance e che la procedura di modifica delle forze, detta *bucket brigade*, non può essere descritta come modulo separato, riportiamo nel seguito l'algoritmo di performance arricchito della componente di modifica del valore delle forze. Per aiutare il lettore a distinguere fra le due componenti, i passi dell'algoritmo che identificano la componente di modifica delle forze sono contrassegnati da un asterisco.

La struttura dell'algoritmo è la seguente:

[11] Queste classi di equivalenza dovrebbero rappresentare quegli insiemi di stati ambientali che richiedono lo stesso tipo di risposta dal sistema. Vedi anche più avanti il concetto di gerarchia di default.

Algoritmo di valutazione delle regole

Fase di inizializzazione

Attribuisci un valore iniziale di forza F(C) ad ogni classificatore C.

Leggi i messaggi ambientali e appendili a **MLE**.

Esegui l'operazione di MATCH tra i messaggi in **MLE** e la parte condizione delle regole in **SC**.

SCA := regole $C_1, ..., C_r$ in cui ogni condizione è verificata da almeno un messaggio.

Estrai un k-campione $\mathcal{A}=[C_1, ..., C_k]$ da **SCA** in modo indipendente con $Prob(C_i) = F(C_i)/\Sigma F(C_i)$ {$F(C_i)$ è la forza del classificatore C_i}.

Metti in **MLI** i messaggi $[M_1, ..., M_k]$ del k-campione.

(*) Per ogni $C_s \in \mathcal{A}$, $F(C_s) := F(C_s) - f(F(C_s))$ {f è una funzione, solitamente lineare della forza del classificatore}

(*) $SCA^- := \mathcal{A}$.

Se vi sono messaggi nelle regole attive diretti agli attuatori (cioè verso l'ambiente esterno), effettua le azioni proposte ed eventualmente chiama il modulo "risolutore dei conflitti".

(*) Se vi sono premi o punizioni distribuiscili alle regole che hanno impostato i messaggi relativi.

Cancella tutti i messaggi presenti in **MLE**.

Run forever

Leggi i messaggi ambientali e appendili a **MLE**.

Esegui l'operazione di MATCH tra i messaggi in **ML** e la parte condizione delle regole in **SC**.

(*) **SCA** := $\varnothing$.

(*) **SCA** := regole $C_1, ..., C_r$ in cui ogni condizione è verificata da almeno un messaggio.

Se vi sono messaggi nelle regole attive diretti agli attuatori (cioè verso l'ambiente esterno), effettua le azioni proposte ed eventualmente chiama il modulo "risolutore dei conflitti".

(*) Se vi sono premi o punizioni distribuiscili alle regole che hanno impostato i messaggi relativi.

Cancella tutti i messaggi presenti in **ML**.

Estrai un k-campione $\mathcal{A}=[C_1, ..., C_k]$ da **SCA** in modo indipendente con $Prob(C_i) = F(C_i)/\Sigma F(C_i)$ {$F(C_i)$ è la forza del classificatore C_i}.

Metti in **MLI** i messaggi $[M_1, ..., M_k]$ del k-campione.

(*) $\mathbf{C}_s := \varnothing$ {$\mathbf{C}_s$ è il multiinsieme dei classificatori i cui messaggi hanno contribuito all'attivazione di C_s}.

(*) Per ogni $C_s \in \mathcal{A}$ metti in $\mathbf{C}_s$ le regole in SCA^- i cui messaggi hanno contribuito all'attivazione di C_s.

(*) Per ogni $C_j \in \mathbf{C}_s$, $F(C_j) := F(C_j) + f(F(C_s))/|\mathbf{C}_s|$

{f è una funzione, solitamente una costante moltiplicativa, della forza di un classificatore}.

(*) Per ogni $C_s \in \mathcal{A}$, $F(C_s) := F(C_s) - f(F(C_s))$

(*) $SCA^- := \mathcal{A}$.

L'algoritmo può essere descritto informalmente nel modo seguente.

Quando viene effettuata un'azione esterna, tutti i classificatori che avevano impostato il messaggio responsabile dell'azione ricevono un premio proporzionale alla utilità dell'azione effettuata. Questo premio viene sommato alla forza del classificatore che lo riceve. Quello che serve è un meccanismo per distribuire all'indietro il premio a tutta la catena di regole che, a partire da un messaggio ambientale, ha condotto all'azione in oggetto. Ciò avviene nel modo seguente. Ognuno dei classificatori appartenenti all'insieme **SCA** dei classificatori attivati, cioè l'insieme dei classificatori le cui condizioni sono state soddisfatte dai messaggi della **ML** fa un offerta proporzionale alla sua forza per ottenere la possibilità di appendere un messaggio alla **MLI**. La scelta delle regole vincenti viene fatta in modo probabilistico dando ad ogni regola una probabilità di vittoria proporzionale all'offerta fatta. Quando una regola vince paga la quantità offerta ai classificatori che all'istante precedente avevano impostato i messaggi che sono stati causa della sua attivazione. A sua volta questa regola potrà ricevere pagamenti da quelle regole che saranno attivate dal messaggio da lei impostato. In questo modo si viene a creare un flusso di pagamenti che ha l'effetto di distribuire i premi ricevuti dall'ambiente lungo la catena che porta alle regole che sono causa prima dell'azione effettuata.

2.4.4 Gerarchie di default

Una gerarchia di default è un insieme di regole dove poche regole (quelle dette di default o regole generali) coprono la maggior parte di situazioni; le situazioni al di fuori della norma sono invece gestite per mezzo di regole specifiche [62].

Si considerino ad esempio i due insiemi di regole in figura 2.7: in questo semplice esempio si vuole che gli insiemi di regole implementino la funzione booleana OR sui due bit delle due condizioni (che in questo esempio sono uguali tra di loro). Il primo insieme di regole ottiene questo risultato per mezzo di una partizione logica dello spazio degli stati: ad ogni possibile stato corrisponde una regola. Nel secondo insieme di regole viene utilizzata invece una gerarchia di default: una regola molto generale (la seconda) viene attivata da qualunque stato ambientale, mentre una più specifica (la prima) è attivata solo dallo stato ambientale corrispondente al messaggio 00. Affinché questo secondo modo di implementare la funzione OR sia equivalente alla partizione logica è necessario implementare un meccanismo che favorisca l'attivazione delle regole più specifiche[12]. Infatti quando è presente il messaggio 00 sulla lista dei messaggi entrambe le regole, quella specifica e quella di default, divengono attive. Per favorire la vittoria di quella più specifica l'offerta fatta dalle due regole viene moltiplicata per la rispettiva specificità $\mathbf{Spe}_i$, definita come segue:

$$\mathbf{Spe}_i = \left(\frac{\text{numero di posizioni diverse da \# in } C_i}{\text{lunghezza del classificatore } C_i} \right) = \text{\% di caratteri diversi da \# in } C_i .$$

In questo modo si favoriscono le regole più specifiche, dato che queste hanno una specificità più alta (una regola massimamente specifica ha specificità uno, una massimamente generale ha specificità zero). Nel capitolo 5 questo argomento verrà ripreso e, nel tentativo di proporre un metodo più efficiente nel creare gerarchie di default, verrà introdotto il nuovo algoritmo chiamato *message-based bucket brigade*.

---

[12] L'equivalenza fra i due meccanismi c'è solo se le regole più specifiche vengono preferite *sempre* a quelle più generali. Normalmente ci si accontenta di un meccanismo che si limita a polarizzare la scelta verso le regole più specifiche e quindi gli approcci gerarchico e non gerarchico non sono equivalenti in senso stretto.

insieme di regole non gerarchico	insieme di regole gerarchico
00;00→00	00;00→00
01;01→11	**;**→11
10;10→11	
11;11→11	

Figura 2.7 - Esempio di insiemi di regole che implementano una suddivisione logica (a sinistra) e gerarchica (a destra) dello spazio degli stati nel caso della funzione booleana OR: si vede come l'insieme gerarchico possa partizionare lo stesso spazio con un numero minore di regole

2.4.5 La generazione di nuove regole: l'algoritmo genetico applicato ai sistemi a classificatori

Vi sono essenzialmente due modi di utilizzare un algoritmo genetico nell'ambito dei sistemi a classificatori. In un caso si può immaginare di codificare l'insieme delle regole che costituiscono un sistema di apprendimento in un unica stringa che sarà un individuo della popolazione. In questo caso la ricerca avviene nello spazio dei possibili sistemi di apprendimento e la fitness di un individuo è valutata in funzione della performance, stimata per mezzo di una simulazione, del sistema di apprendimento che esso codifica. Nel secondo approccio si fa corrispondere un individuo ad ogni classificatore; in questo caso la ricerca avviene nello spazio delle possibili regole e la fitness di un individuo è valutata come funzione dell'utilità per il sistema dell'individuo stesso (misurata dalla forza dell'individuo). Il nostro approccio si basa sul secondo modello, nel quale l'algoritmo genetico viene utilizzato come strumento per la generazione di nuove regole che abbiano una elevata probabilità di risultare utili.

Per poter applicare l'algoritmo genetico è necessario definire una politica di sostituzione dei classificatori che permetta al sistema di mantenere un comportamento istante per istante significativo. Infatti, a differenza di quanto accade quando lo si applica a problemi di ottimizzazione in senso classico (vedi sezione 2.3), il sostituire l'intera popolazione vecchia con una nuova potrebbe portare alla distruzione di strutture utili, per esempio gerarchie o catene di regole. Normalmente la politica seguita è quella di sostituire una percentuale fissata di classificatori scegliendoli tra quelli con forza (e quindi utilità) più bassa: ciò rende meno probabile l'eliminazione di regole utili al sistema.

Vi sono poi situazioni particolari nelle quali sarebbe necessaria l'introduzione di nuove regole, ma l'algoritmo genetico standard non è in grado di intervenire in modo sufficientemente tempestivo. Questo succede ad esempio quando non esistono regole in grado di essere attivate dai messaggi ambientali o quando tutti i messaggi generabili verso l'ambiente sono privi di significato. La creazione di nuove regole che risolvono questo problema da parte dell'algoritmo genetico può richiedere un tempo di attesa eccessivo: si introducono pertanto degli operatori detti di *genetica di contesto* che creano classificatori adatti alla soluzione del particolare problema (questa estensione del modello, che viene introdotta per ragioni di efficienza, non è contemplata nel modello di Fig.2.6). Ad esempio l'operatore detto *cover detector* crea un nuovo classificatore la cui parte condizione si accoppia correttamente con almeno uno dei messaggi ambientali presenti; la regola *cover effector* invece genera un nuovo classificatore in cui una condizione è

soddisfatta da uno dei messaggi presenti su **ML** e in cui il messaggio impostato va agli effettori.

2.4.6 La funzione di valutazione

La funzione di valutazione può essere parte integrante del sistema di apprendimento o esterna ad esso. Nel primo caso rappresenta una caratteristica strutturale del sistema che apprende che è capace di decidere della utilità di una azione: un esempio possono essere il senso di appagamento che deriva dal mangiare sotto lo stimolo della fame o il dolore provocato da una caduta causata da un movimento errato. Nel secondo caso rappresenta un insegnante il cui compito sia limitato a segnalare la correttezza (o la non correttezza) delle azioni effettuate (è questo il caso presentato in Fig.2.6).

2.4.7 Caratteristiche naturali dei sistemi a classificatori

I sistemi a classificatori possono essere considerati un modello naturale, sebbene in modo diverso dagli altri modelli presentati in questa tesi. Gli aspetti naturali di questo modello sono i seguenti:

(i) i SC utilizzano gli AG come meccanismo di ricerca delle regole,
(ii) è possibile organizzare insiemi di SC in strutture che rispecchiano l'organizzazione funzionale del sistema di controllo del comportamento in animali superiori (vedi il modello di Tinbergen e il sistema ALECSYS nella sezione 5.2).

Con riferimento alle caratteristiche elencate nella sezione 2.1, i SC non sono direttamente una modellizzazione di un sistema naturale (solo la componente AG lo è). I SC si possono parallelizzare abbastanza facilmente (vedi ad esempio la sezione 5.4), sono stocastici e capaci di modificare il loro comportamento al variare delle condizioni esterne. L'uso di retroazione positiva riguarda solo la componente AG.

Questo è certamente, dei modelli presentati, quello che presenta capacità di apprendimento più evidenti: obiettivo del sistema a classificatori è, dato un problema, l'apprendere insiemi di regole con relative forze che ne permettano la soluzione.

I concetti di evoluzione e selezione sono presenti nella componente AG.

2.5 Strategie evolutive

L'insieme di algoritmi che va sotto il nome di *Strategie Evolutive* (SE - dal tedesco *Evolution Strategie*) è stato sviluppato da Rechenberg [72] e Schwefel [78] a partire dal 1965 presso la Technische Universität di Berlino. Questi algoritmi presentano, nonostante siano stati sviluppati in modo completamente indipendente, molti punti di somiglianza con gli algoritmi genetici.

La filosofia seguita da questo approccio è di nuovo quella di utilizzare la metafora della genetica delle popolazioni come modello di riferimento per sviluppare algoritmi di ottimizzazione. Come nel caso degli AG, le SE lavorano applicando degli operatori genetici ad una popolazione di stringhe che rappresentano soluzioni ammissibili al problema di ottimizzazione da risolvere. A differenza degli AG, le stringhe su cui operano le strategie evolutive sono vettori di $\mathbb{R}^n$ e sono in grado di modificare la probabilità di mutazione durante il processo di ottimizzazione. Le loro prime applicazioni sono state nel campo dell'ottimizzazione di parametri per applicazioni ingegneristiche [72]. Dato che gli algoritmi genetici sono già stati introdotti nella sezione 2.3, illustreremo qui il modello delle strategie evolutive per analogia con il modello degli algoritmi genetici.

Entrambi i modelli si ispirano alla metafora della evoluzione delle popolazioni e fanno uso di una popolazione come strumento di ricerca nello spazio delle soluzioni. Entrambi i modelli utilizzano una rappresentazione di derivazione genetica: una soluzione è codificata in un cromosoma (detto *genotipo*) cioè in una stringa di geni che contiene tutta l'informazione necessaria per costruire la soluzione esplicita (il cosiddetto *fenotipo*). Inoltre entrambi i modelli fanno uso di operatori, di nuovo ispirati alla genetica, per produrre nuove soluzioni (ad es. mutazione e crossover).

Le differenze fondamentali sono le seguenti:
(i) il tipo di codifica utilizzata,
(ii) il tipo di operatori genetici,
(iii) la strategia di ricerca.

Nelle SE, a differenza degli AG dove l'informazione è codificata secondo un alfabeto binario, ogni gene è un valore intero o reale che rappresenta direttamente uno dei parametri da ottimizzare.

Questo aspetto determina la fondamentale differenza che i due approcci danno agli operatori di mutazione e ricombinazione come strumenti per muoversi nello spazio delle soluzioni. Le SE usano infatti come strumento principale di ricerca una probabilità di mutazione adattativa: l'ampiezza del campo di valori in cui avviene la mutazione varia durante l'esecuzione dell'algoritmo e si adatta alle caratteristiche dello spazio di ricerca man mano che l'algoritmo trova soluzioni vieppiù vicine alla soluzione ottima. Al contrario, gli AG utilizzano come strumento principale di ricerca l'operatore di crossover (sebbene recenti studi abbiano rivalutato il ruolo della mutazione [76]).

Diamo ora di seguito una descrizione dell'algoritmo base del modello SE applicato ad un problema di ottimizzazione.

Un individuo I è un vettore di $\mathbb{R}^n$. Un insieme di individui è detto popolazione.

L'algoritmo lavora modificando popolazioni di individui come segue.
Poni t := 0;

Genera una popolazione iniziale di μ^t individui;
Finché (condizione di terminazione) esegui

t := t + 1;

Genera λ^t discendenti applicando gli operatori di mutazione e ricombinazione alla popolazione di μ^{t-1} individui ($\lambda^t \geq \mu^{t-1}$);

Seleziona gli individui che formeranno la popolazione μ^t corrente dalla popolazione precedente e dai discendenti (questa è la strategia chiamata ($\lambda+\mu$); è possibile utilizzare una strategia differente, detta strategia (λ,μ), nella quale la popolazione corrente è selezionata a partire dai soli discendenti λ^t);

La condizione di terminazione solitamente usata è sul numero massimo di iterazioni (N_{MAX} definito dall'utente) o sul numero massimo di iterazioni senza che vi sia un miglioramento della soluzione migliore trovata.

Qui di seguito diamo una descrizione degli operatori.

2.5.1 Gli operatori

In questa sezione introduciamo i tre operatori principali utilizzati dalle SE: riproduzione/selezione, ricombinazione e mutazione.

L'operatore di riproduzione/selezione

A differenza che negli AG, qui abbiamo due politiche di riproduzione/selezione, la prima detta strategia ($\lambda+\mu$), la seconda strategia (λ,μ).

In entrambe le strategie vengono generati $|\lambda^t|$ discendenti a partire da $|\mu^t|$ genitori, con $|\lambda^t| \geq |\mu^t|$, applicando un operatore di riproduzione uguale a quello utilizzato dagli AG. La differenza tra le due strategie consiste nell'insieme selezionato, che riporterà la popolazione alla dimensione $|\mu^t|$: nella strategia ($\lambda+\mu$) la selezione viene operata sulla unione della popolazione precedente con quella dei discendenti; nella strategia (λ,μ) la selezione viene operata sui soli discendenti.

L'operatore di ricombinazione

Quello che negli AG viene chiamato crossover, nelle strategie evolutive diviene una classe di operatori che va sotto il nome più generico di operatori di ricombinazione. La ricombinazione può avvenire in diversi modi (vedi [4]).

Noi riportiamo qui i due modelli di operatore di ricombinazione più usati.

Quello detto di *ricombinazione discreta* è l'operatore di crossover degli algoritmi genetici applicato a stringhe di reali e opera mescolando il materiale genetico preso da due genitori per produrre discendenti. Nell'esempio della figura 2.8 una ricombinazione discreta è effettuata prendendo il valore dei primi tre geni dal primo genitore e il valore degli altri tre dal secondo.

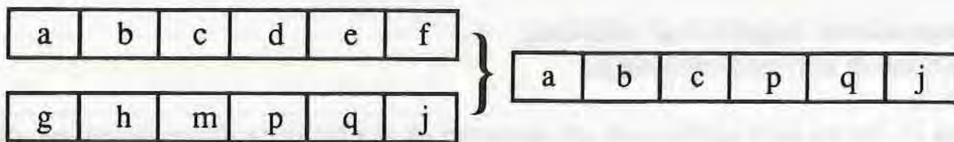


Fig. 2.8 - Esempio di applicazione dell'operatore di ricombinazione discreta

Il secondo tipo, detto operatore di *ricombinazione intermediata*, utilizza k genitori per generare un discendente nel quale il valore assunto dai geni è una funzione del valore assunto dai corrispondenti geni nei genitori. L'esempio di figura 2.9 illustra questo operatore nel caso di due genitori, con funzione di ricombinazione data dalla media.

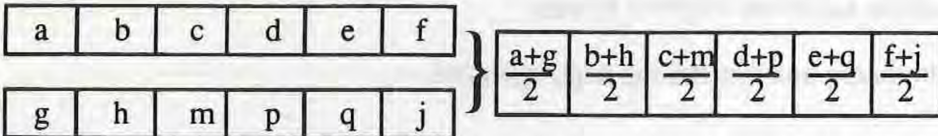


Fig. 2.9 - Esempio di applicazione dell'operatore di ricombinazione intermediata

<u>*L'operatore di mutazione*</u>

Se $I=(i_1, \ldots, i_k, \ldots, i_r)$ è un individuo della popolazione, l'individuo mutato è dato da

$$I_M = (i_1 * (1+z_1), \ldots, i_k * (1+z_k), \ldots, i_r * (1+z_r))$$

essendo z_k una variabile casuale z distribuita normalmente con valore atteso E=0 e deviazione standard σ.

L'ampiezza della mutazione dipende perciò dal valore della deviazione standard σ. Aspetto interessante e peculiare delle strategie evolutive è che la stessa deviazione standard σ è considerata una variabile da ottimizzare, e come tale è soggetta a sua volta all'azione degli operatori genetici.

Nella tabella seguente riassumiamo le maggiori differenze tra il modello AG e il modello SE.

Tabella 2.3 - Differenze tra algoritmi genetici e strategie evolutive

	Algoritmi genetici	**Strategie Evolutive**
Schema di codifica	binaria	valori reali
Probabilità di mutazione	costante	soggetta ad evoluzione
Operatore di ricombinazione	crossover	ricombinazione discreta o ricombinazione intermediata
Popolazione generante	solo genitori	genitori e discendenti

Vogliamo qui solo citare il fatto che il problema di cercare un algoritmo evolutivo che riunisca le caratteristiche positive delle due classi di algoritmi è stato affrontato da molti ricercatori. In particolare il problema dello schema di codifica (se cioè sia meglio utilizzare parametri reali o binari) è stato affrontato ad esempio da [17], [33], [34].

2.5.2 Caratteristiche naturali delle strategie evolutive

Le SE hanno caratteristiche molto simili a quelle degli algoritmi genetici: come gli AG sono una modellizzazione del processo che sta alla base dell'evoluzione delle specie e possono essere facilmente parallelizzate. Il fatto che ogni operatore venga applicato con una certa probabilità e che la riproduzione favorisca, probabilisticamente, gli individui migliori, rende le SE un sistema stocastico; rispetto agli AG hanno la capacità di adattare la probabilità di mutazione all'andamento della computazione. La retroazione positiva è dovuta all'operatore di riproduzione/selezione. Come negli AG, i concetti di evoluzione e selezione sono utilizzati dall'algoritmo.

2.6 Reti immunitarie

Le reti immunitarie (RI) sono un formalismo che ha preso spunto da recenti studi sul funzionamento del sistema immunitario negli animali superiori (vedi [82], [83], [84]). I collegamenti con i settori dell'ottimizzazione e dell'apprendimento automatico sono più deboli che nel caso dei sistemi esposti nelle sezioni precedenti, dato che solo molto recentemente le RI sono state proposte come metodologia di indagine in questi settori. Il funzionamento delle RI può essere spiegato facendo uso di due livelli di astrazione: la dinamica e la metadinamica, qui di seguito presentate. Nel seguito illustriamo come alcune delle caratteristiche del modello che spiega il funzionamento di tale sistema possano essere utilizzate per progettare un algoritmo per l'ottimizzazione di funzioni.

2.6.1 Componenti del sistema immunitario

Il sistema immunitario (SI) è una rete di cellule interagenti (*β-linfociti*) che protegge l'organismo da sostanze esterne, dette *antigeni*. Tale protezione è ottenuta dal sistema per mezzo della produzione di molecole (*anticorpi*) prodotti dai β-linfociti che, attaccandosi agli antigeni, ne determinano la distruzione. L'insieme dei possibili tipi di antigeni è a priori sconosciuto al SI: pertanto un requisito fondamentale che il SI deve soddisfare è la capacità di adattarsi all'evoluzione di un ambiente imprevedibile detto *ambiente degli antigeni*. Questa capacità adattativa è spiegata per mezzo di una teoria *selezionista* proposta da Varela e Coutinho ([82], [83]), che verrà illustrata in seguito.

Come detto, il sistema immunitario è una rete di cellule interagenti. Sostanze esterne all'organismo dette antigeni vengono identificate dal SI e poi distrutte: esempi di antigeni sono i batteri, i pollini, i virus ecc. Le molecole del SI che operano il riconoscimento sono gli anticorpi. Un anticorpo riconosce un antigene per mezzo del riconoscimento di una sua piccola regione, detta *epitope*. Il riconoscimento avviene per mezzo di un accoppiamento (binding) tra epitope e una parte dell'anticorpo detta *paratope*. Il paratope può essere considerato come una chiave che è adatta solo per alcuni epitope. La forza e la specificità dell'interazione tra un anticorpo e un antigene (cioè fra il paratope e l'epitope corrispondenti) è misurata da una grandezza detta *affinità* dell'interazione. L'affinità dipende dal grado di complementarietà dell'epitope con il paratope. Epitope e paratope sono caratterizzati dal valore assunto da alcune variabili, quali la lunghezza, l'area, la carica elettrica, la forma molecolare ecc. L'affinità può essere espressa come una funzione di queste grandezze. Un anticorpo può riconoscere un insieme di diversi epitopi, con i quali avrà diversi gradi di affinità. Un epitope può a sua volta essere riconosciuto da diversi tipi di anticorpi. Ogni β-linfocita produce un solo tipo di anticorpo. Sulla superficie di un β-linfocita sono attaccati circa 10^5 anticorpi, che sono utilizzati come sensori per segnalare l'eventuale presenza di epitopi corrispondenti. Il sistema immunitario di un organismo contiene approssimativamente 10^7 tipi diversi di β-linfociti; questo numero sembra essere sufficiente a riconoscere ogni possibile tipo di antigene se ipotizziamo che i β-linfociti siano distribuiti uniformemente sull'insieme di tutti i tipi di β-linfociti possibili e se ricordiamo che ogni anticorpo può riconoscere diversi tipi di epitopi.

Aspetto fondamentale per capire la dinamica del SI è che gli anticorpi riconoscono sia gli antigeni che gli altri anticorpi, a patto che questi siano provvisti del corretto epitope, che in questo caso viene chiamato *idiotope*. Se un anticorpo i riconosce un anticorpo j (cioè il paratope di i è complementare all'idiotope di j) allora j viene detto l'anticorpo anti-

idiotipico di *i* (l'anticorpo anti-idiotipico *j* di *i* può essere considerato "l'immagine interna" nel SI dell'antigene esterno riconosciuto da *i*).

Dinamica del Sistema Immunitario

Quando un anticorpo che si trova sulla superficie di un β-linfocita si lega ad un'altra molecola (un antigene o un altro anticorpo), il β-linfocita è stimolato a riprodursi (clonazione) e ad emettere degli anticorpi liberi: il linfocita è detto essere attivato. La quantità di anticorpi emessi è proporzionale alla affinità m_{ij} del legame fra le molecole di tipo i e j (l'indice i si riferisce al paratope, mentre l'indice j si riferisce all'epitope). La molecola alla quale un anticorpo si lega verrà successivamente distrutta per opera di altre cellule che fanno sempre parte del sistema immunitario. Se un β-linfocita non viene stimolato dalla presenza di molecole affini ai suoi anticorpi muore in pochi giorni: si ha pertanto un processo di selezione che opera eliminando i β-linfociti che non vengono attivati e incrementando il numero di quelli che vengono attivati. Questo processo è detto di selezione per clonazione. Un β-linfocita viene attivato non solo dalla presenza dei corrispondenti antigeni, ma anche da anticorpi anti-idiotipici. Questi anticorpi anti-idiotipici determinano l'attivazione di ulteriori anticorpi. In questo modo si ottiene un intera rete di anticorpi interagenti chiamata rete idiotipica.

Si definisce sensitività della rete all'anticorpo di tipo j il valore $\sigma_j(t)$ definito come segue

$$\sigma_j(t) = \sum_{i=1}^{N} m_{ij} f_i(t) + \sum_{k=1}^{M} m_{kj} a_k(t) \tag{2.8}$$

dove $f_i(t)$, i = 1 ... N, è la concentrazione degli anticorpi liberi di tipo *i* all'istante t e $a_k(t)$, k = 1 ... M, rappresenta la concentrazione degli M antigeni presenti nell'ambiente antigenico all'istante t.

Metadinamica del Sistema Immunitario

La dinamica del sistema immunitario spiega il comportamento del sistema per un dato insieme di anticorpi in un dato ambiente antigenico. La metadinamica spiega invece il meccanismo con il quale nuovi tipi di anticorpi vengono *arruolati* e anticorpi non più utili vengono lasciati morire. Il sistema immunitario mantiene circa costante il numero totale di tipi di anticorpi differenti. Il carattere adattativo e le capacità di apprendimento del SI risiedono essenzialmente nel meccanismo di arruolamento che seleziona nuovi tipi di β-linfociti da inserire nella dinamica del sistema scegliendoli tra un numero enorme di tipi generati in modo pressoché casuale. Questo processo di arruolamento seleziona le nuove specie generate sulla base dello stato corrente del sistema, cioè in accordo alla sensibilità del SI alla nuova specie generata.

I β-linfociti, che vengono generati dal midollo spinale, differiscono per il tipo di anticorpo che producono; questa differenza è determinata geneticamente e quindi nuovi tipi di β-linfociti possono essere generati per mezzo di operazioni di mutazione e ricombinazione genetica di β-linfociti preesistenti. Questo processo (essenzialmente casuale) determina la sostituzione di circa il 10% del totale dei tipi di β-linfociti ogni 24 ore. Inoltre quando un β-linfocita viene attivato, i geni che codificano l'informazione necessaria alla produzione degli anticorpi corrispondenti cominciano a mutare più

velocemente, determinando pertanto un ulteriore incremento del numero di specie, questa volta però orientato verso anticorpi simili a quelli già attivi.

Sebbene nuovi tipi di anticorpi siano creati continuamente, la probabilità che un nuovo tipo sia incorporato nella dinamica del SI è differente per tipi differenti. Sia S lo spazio di tutti i tipi di anticorpi possibili. Allora la probabilità che una nuova specie $j \in S$ sia incorporata nella dinamica del SI dipende dalla sensitività $\sigma_j(t)$ della rete per il tipo j all'istante t. Si noti che, in accordo con l'equazione (2.8), la sensitività $\sigma_j(t)$ della rete per il tipo j è alta se j ha una elevata affinità m_{jk} con un numero sufficiente di anticorpi attivati di tipo k. Questa strategia nella quale β-linfociti creati in modo casuale che risultano essere affini ad anticorpi attivati vengono reclutati viene detta *strategia di arruolamento.* Pertanto, nel SI, vengono incorporati nella dinamica del sistema nuovi anticorpi che sono affini ad anticorpi che sembrano essere molto utili al sistema.

Se interpretiamo il SI come una rete di tipi (anticorpi ed antigeni) interagenti con un livello di interazione dato dal valore dell'affinità, allora la metadinamica opera una continua modifica della topologia della rete.

2.6.2 Il meccanismo di arruolamento come strumento di ottimizzazione

In questa sottosezione illustreremo come utilizzare le reti immunitarie per la soluzione di problemi di ottimizzazione (vedi [6], [7]). Dato un problema di ottimizzazione, dove il vettore $x^{(i)}$ rappresenta una soluzione ammissibile e $f_i=f(x^{(i)})$ il valore della funzione obiettivo corrispondente al vettore $x^{(i)}$, se uguagliamo la concentrazione degli anticorpi liberi f_i al valore della funzione obiettivo $f_i=f(x^{(i)})$ e definiamo una grandezza equivalente all'affinità m_{ij} che rappresenti una misura della distanza fra due soluzioni $x^{(i)}$ e $x^{(j)}$, allora possiamo definire un algoritmo che, operando in modo simile alla metadinamica del sistema immunitario, può essere utilizzato per l'ottimizzazione di funzioni.

Due sono gli algoritmi di ottimizzazione finora sviluppati utilizzando il modello del SI come metafora di riferimento. La differenza fondamentale fra i due consiste nel modo scelto per rappresentare ogni singolo tipo di β-linfocita.

Ogni clone di β-linfocita è caratterizzato dal valore assunto da un certo numero n di variabili chimico-fisico-strutturali e quindi può essere rappresentato come un punto nello spazio $S \in \mathbb{R}^n$. In questo caso l'affinità m_{ij}, che rappresenta in qualche modo l'intensità dell'interazione tra gli anticorpi liberi[13] di tipo i e j, è una funzione della distanza[14] tra i e j nello spazio $\mathbb{R}^n$. Questo tipo di rappresentazione è utilizzata per la definizione dell'algoritmo che va sotto il nome di IRM (Immune Recruitment Mechanism - Metodo di arruolamento immunitario, vedi [7]). Il metodo alternativo di rappresentazione, che da origine ad un algoritmo chiamato GIRM (Genetic Immune Recruitment Mechanism - Metodo genetico di arruolamento immunitario, vedi [7]), si basa su una rappresentazione a livello di genotipo dell'informazione necessaria alla generazione di un anticorpo. In questo caso l'affinità tra due anticorpi è misurata da una funzione della differenza tra i cromosomi relativi. Nell'algoritmo GIRM i cromosomi sono stringhe di valori binari e

---

[13] A differenza di quanto accade con le reti neurali, il valore di m_{ij} non viene modificato dalla dinamica del sistema (esso dipende solo dalla posizione dei due punti rappresentanti i due anticorpi nello spazio $\mathbb{R}^n$).

[14] Sono finora state proposte, e parzialmente testate, le seguenti funzioni: gaussiana, quadratica, inversa, lineare ed esponenziale negativa.

pertanto l'affinità risulta essere una funzione della distanza di Hamming fra i due cromosomi[15].

Consideriamo ora, a puro scopo illustrativo per introdurre l'algoritmo IRM, un esempio di ottimizzazione di una funzione di n variabili:

Max f(x) con $x \in \mathbb{R}^n$, dove $x = (x_1, ..., x_n)$ è un vettore di n variabili.

Data una popolazione $\mathcal{A}=(x^{(1)} ... x^{(N)})$, ordinata per valori di $f_i=f(x^{(i)})$ decrescenti, un intero N_b ($N_b \leq N$) e un vettore $d=(d_1, ..., d_n)$, definiamo

$$T(\mathcal{A},N_b) = \frac{\sum_{i=1}^{N_b} f_i}{N_b} \tag{2.9}$$

e

$$Rett(\mathcal{A},d)=\{(x_1 ... x_n) \mid \underset{x^{(j)}\in\mathcal{A}}{\mathrm{Min}}\ x_i^{(j)}- d_i \leq x_i \leq \underset{x^{(j)}\in\mathcal{A}}{\mathrm{Max}}\ x_i^{(j)}+ d_i\} \tag{2.10}$$

L'algoritmo è il seguente:

Sia $\mathcal{P}$ l'insieme di N punti generati casualmente con distribuzione uniforme nell'n-cubo unitario.
Finché (condizione di terminazione) esegui
    $\mathcal{A} := \emptyset$;
    Finché $|\mathcal{A}| < N_n$ ($N_n \in \mathbb{N}$, $N_n \leq N$) esegui
        Genera a caso un punto y nel rettangolo Rett($\mathcal{P}$,d);
        Scegli a caso N_p ($N_p \in \mathbb{N}$, $N_p \leq N$) punti $x^{(1)}, ..., x^{(N_p)}$ in $\mathcal{P}$;
        Se $\dfrac{\sum_{j=1}^{N_p} m(x^{(j)},y)f_j}{\sum_{j=1}^{N_p} m(x^{(j)},y)} \geq T(\mathcal{P},N_b)$ allora $\mathcal{A} := \mathcal{A} \cup \{y\}$
    $\mathcal{C}$:= gli N_n punti peggiori in $\mathcal{P}$
    $\mathcal{P} := (\mathcal{P} - \mathcal{C}) \cup \mathcal{A}$

Una condizione di terminazione è quando, dopo un numero prefissato di tentativi, non si è riusciti ad arruolare nessun nuovo punto.

Dati due vettori, la funzione di affinità può essere ad esempio

$$m_{ij} = m(x^{(i)},x^{(j)})=1- e^{-dist(x^{(i)},x^{(j)})}$$

dove *dist* è la distanza euclidea tra i due punti.

Il funzionamento dell'algoritmo IRM può essere descritto in modo informale nel modo seguente:

[15] IRM è quindi un algoritmo che opera nel continuo, mentre GIRM opera nel discreto.

All'istante iniziale viene generata una popolazione $\mathcal{P}$ di N punti scelti in modo casuale con distribuzione uniforme nell'n-cubo unitario. Si generano in modo casuale nuovi punti $x^{(k)}$ con coordinate appartenenti al rettangolo definito dalla (2.10). I nuovi punti $x^{(k)}$ vengono arruolati se la loro affinità con N_p punti scelti a caso nella popolazione è maggiore di un valore soglia dato dalla disequazione (2.9). Quando N_n nuovi punti sono stati arruolati gli N_n punti peggiori della popolazione $\mathcal{P}$ vengono scartati e sostituiti dai nuovi punti arruolati. L'algoritmo si ferma quando, dopo un numero prefissato di tentativi, non si è riusciti ad arruolare nessun nuovo punto.

L'algoritmo utilizza alcuni parametri il cui significato informale è riassunto qui di seguito:

$N \in \mathbb{N}$ è la dimensione della popolazione di punti (cardinalità dell'insieme $\mathcal{P}$).

$N_b \in \mathbb{N}$ è il numero di punti utilizzato per calcolare il valore soglia $T(\mathcal{A},N_b)$. Questo valore soglia viene usato per decidere se reclutare o no un nuovo punto.

$N_n \in \mathbb{N}$ è il numero di nuovi punti che viene arruolato ad ogni generazione (e corrispondentemente il numero di punti che viene scartato dalla vecchia popolazione, in modo tale da mantenere costante la dimensione di $\mathcal{P}$).

$N_p \in \mathbb{N}$ è il numero di punti con i quali un potenziale candidato all'arruolamento $x^{(k)}$ determina la sua affinità.

Applicazioni di entrambi gli algoritmi - IRM e GIRM - a problemi di ottimizzazione combinatoria hanno finora dato risultati incoraggianti [7], in alcuni casi paragonabili ai risultati ottenuti con le strategie evolutive (IRM) o con gli algoritmi genetici (GIRM).

2.6.3 Caratteristiche naturali delle reti immunitarie

Le RI sono ispirate al sistema immunitario. La generazione di nuovi punti e il test di reclutamento possono avvenire in parallelo. La generazione casuale dei nuovi punti rende l'algoritmo stocastico. L'algoritmo è adattativo perché utilizza i punti arruolati nel passato per dirigere il processo di ricerca. Il concetto di selezione è presente in forma diversa da quella degli AG e delle SE: nelle RI i nuovi punti generati sono selezionati in base alla loro affinità con il sistema.

3

Il sistema "formiche" e il problema del commesso viaggiatore

3.1 Il sistema formiche
3.2 Gli algoritmi Ant-density e Ant-quantity
3.3 L'algoritmo Ant-cycle
3.4 La taratura dei parametri
3.5 Ulteriori indagini su Ant-cycle
3.6 Applicazioni ad altri problemi
3.7 Conclusioni e lavoro futuro

3.1 Il sistema formiche

In questo capitolo riprendiamo il sistema formiche (SF) introdotto in 2.2. Nel seguito verranno riformulate brevemente le proprietà caratteristiche del sistema e verranno poi introdotti tre algoritmi che si differenziano a causa del modo e della frequenza con cui le formiche depositano la traccia sugli archi da loro percorsi.

Come già detto il SF è composta da m agenti (le formiche) che si muovono su un grafo secondo la regola probabilistica che qui riportiamo

$$p_{ij}(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{j \in J} [\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}} & \text{se } j \in \mathbf{J} \\ 0 & \text{altrimenti} \end{cases} \tag{3.1}$$

essendo J un insieme di nodi opportunamente definito (vedi seguito). In questa regola di transizione sono presenti due variabili e due parametri. La variabile η_{ij} è caratteristica del problema e, nella nostra applicazione al problema del commesso viaggiatore (TSP, introdotto nel capitolo 2.2), è definita come il reciproco della distanza tra le città i e j ($\eta_{ij}=1/d_{ij}$); in generale questa variabile rappresenta la componente greedy della regola decisionale seguita dalle formiche; in alcune applicazioni potrebbe essere funzione del tempo ($\eta_{ij}(t)$). La variabile $\tau_{ij}(t)$ rappresenta l'intensità di traccia sull'arco che unisce le città i e j: nel passaggio dal tempo t al tempo t+1 varia secondo la regola

$$\tau_{ij}(t+1)=\rho \cdot \tau_{ij}(t)+\Delta\tau_{ij}(t,t+1) \tag{3.2}$$

con

$$\Delta\tau_{ij}(t,t+1) = \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t,t+1) \tag{3.3}$$

quantità di traccia per unità di lunghezza depositata sull'arco (i,j) dalla k-esima formica nell'intervallo temporale (t,t+1); ρ è un coefficiente reale $0 \leq \rho \leq 1$; $(1-\rho)$ può essere intuitivamente interpretato come *evaporazione* che permette al sistema di perdere memoria delle azioni passate.

A seconda del valore assunto dalla (3.3) si hanno diverse versioni del SF, che verranno presentate in dettaglio nelle prossime due sezioni. Nella sezione 3.4 studieremo la performance del SF in funzione dei valori di α e β.

Quella che viene illustrata in questo capitolo è la prima applicazione del SF (si veda anche [22], [23], [24]: pertanto SF è visto come una particolare euristica per la risoluzione del TSP. Alla fine del capitolo vedremo come sia possibile generalizzare questo approccio mettendone in luce gli aspetti fondamentali e daremo qualche suggerimento su come lo stesso metodo possa essere utilizzato per la risoluzione di altri problemi di ottimizzazione: SF risulta allora essere uno schema di algoritmo per problemi di ottimizzazione combinatoria.

Ricapitolando, il SF funziona come segue: data una istanza di dimensione n del problema del TSP, si posizionano sulla città i-esima un numero $b_i(0)$ di formiche (con $b_i(0) \geq 0$, $b_i(0) \in \mathbb{N}$ e $\Sigma_i b_i(0)=m$), si pone su ogni arco un valore iniziale $\tau_{ij}(0)$ di traccia

(nella nostra applicazione i valori iniziali sono uguali su tutti gli archi), per ogni formica si applica la regola di transizione data dalla (3.1). Dopo che tutte le formiche si sono mosse le tracce vengono aggiornate secondo la (3.2) e la procedura è iterata. Ad ogni iterazione ciascuna formica appende in una lista, chiamata *tabu list*, il nome dell'ultima città visitata, al fine di evitare di ritornarvi prima di avere compiuto un giro completo: l'insieme **J** della formula 3.1 è definito come l'insieme delle città che non appartengono alla tabu list della formica k-esima, cioè $\mathbf{J}=\{j: j \notin \mathbf{tabu_k}\}$ dove $\mathbf{tabu_k}$ è un vettore contenente la tabu list della k-esima formica. Dopo n passi tutte le formiche hanno compiuto un giro completo e si ritrovano nella città di partenza. A questo punto le *tabu list* vengono vuotate, la lunghezza del ciclo percorso da ogni formica viene calcolato, il ciclo migliore viene memorizzato, se non è verificato un test di uscita il sistema riprende a funzionare come se ripartisse dallo stato iniziale (ma il valore delle tracce non è più quello iniziale).

Il test di uscita è il seguente: l'algoritmo si arresta se è stato superato il numero massimo di cicli P_{MAX} definito dall'utente o se tutte le formiche stanno percorrendo lo stesso ciclo (è la situazione di *uni-cammino* descritta in seguito cui corrisponde il fatto che il sistema non sta più esplorando nuove soluzioni).

3.2 Gli algoritmi Ant-density e Ant-quantity

Gli algoritmi Ant-density e Ant-quantity differiscono per il fatto che nel primo viene depositata una quantità di traccia Q_1 per ogni unità di lunghezza dell'arco (i,j) ogni volta che una formica si sposta dalla città i alla città j, nel secondo viene depositata una quantità di traccia Q_2/d_{ij} per ogni unità di lunghezza dell'arco (i,j):

nel modello Ant-density

$$\Delta\tau_{ij}^{k}(t,t+1) = \begin{cases} Q_1 & \text{se la k-esima formica percorre l'arco (i,j) nell'intervallo (t,t+1)} \\ 0 & \text{altrimenti} \end{cases} \tag{3.4}$$

nel modello Ant-quantity

$$\Delta\tau_{ij}^{k}(t,t+1) = \begin{cases} \dfrac{Q_2}{d_{ij}} & \text{se la k-esima formica percorre l'arco (i,j) nell'intervallo (t,t+1)} \\ 0 & \text{altrimenti} \end{cases} \tag{3.5}$$

Intuitivamente, nel secondo modello le formiche rendono gli archi brevi più desiderabili, rinforzando in questo modo il fattore chiamato visibilità η_{ij} nell'equazione (3.1)). Per quanto detto i due algoritmi sono:

Algoritmo Ant-density (Ant-quantity)

1 Inizializzazione
  Poni t:=0 {t è il contatore del numero di cicli}
  Per ogni arco (i,j) poni una quantità di traccia iniziale $\tau_{ij}(t)$ e poni $\Delta\tau_{ij}(t,t+1):=0$
  Posiziona $b_i(t)$ formiche su ogni nodo i {$b_i(t)$ è il numero di formiche sul nodo i all'istante t}
  Poni s:=1 {s è l'indice della tabu list}
  Per i:=1 fino ad n
    Per k:=1 fino a $b_i(t)$
      tabu$_k$(s):=i {la città di partenza è il primo elemento della tabu list}

2 Ripeti fino a quando la tabu list è piena {questo passo viene ripetuto n-1 volte}
  2.0 Poni s:=s+1
  2.1 Per i:=1 fino ad n {per ogni città}
    Per k:=1 fino a $b_i(t)$ {per ogni k-esima formica non ancora mossa sulla città i}
      Scegli la città j dove andare, con probabilità $p_{ij}(t)$ data dall'equazione (3.1)
      Muovi la k-esima formica in j {questa istruzione crea i nuovi valori $b_j(t+1)$}
      Inserisci il nodo j in **tabu**$_k$(s)
      Poni $\Delta\tau_{ij}(t,t+1):=\Delta\tau_{ij}(t,t+1)+Q_1$ se il modello è Ant-density
      ($\Delta\tau_{ij}(t,t+1):=\Delta\tau_{ij}(t,t+1)+\dfrac{Q_2}{d_{ij}}$ se il modello è Ant-quantity)
  2.2 Per ogni arco (i,j) calcola $\tau_{ij}(t+1)$ secondo l'equazione (3.2)

```
3 Memorizza il cammino più breve trovato finora
    Se (P < P_MAX) o (comportamento ≠ uni-cammino)      {P è il numero di cicli}
    allora
        Vuota tutte le tabu list
        Poni s:=1
        Per i:=1 fino ad n do
            Per k:=1 fino a b_i(t) do
                tabu_k(s):=i                {dopo un ciclo ogni formica è nella posizione
                                             iniziale}
        Poni t:=t+1
        Per ogni arco (i,j) poni Δτ_ij(t,t+1):=0
        Vai al passo 2
    altrimenti
        Stampa il ciclo più breve trovato e Fermati
```

È importante sottolineare come la probabilità di transizione data dalla formula (3.1) dipenda da due "fattori di desiderabilità": la visibilità η_{ij} e la traccia τ_{ij}. Il primo fattore dice che più una città è vicina più è desiderabile (è una regola euristica di tipo greedy, che porta a scegliere la mossa localmente ottima). Il secondo fattore dice che sono più desiderabili le città collegate mediante archi che sono stati scelti nel passato da molte formiche (in questo caso la regola euristica seguita è che se molte formiche lo hanno scelto, quell'arco deve essere buono).

L'importanza relativa di questi due fattori è funzione del valore assunto dai parametri α e β. Se per esempio poniamo $\alpha = 0$ otteniamo un algoritmo greedy stocastico con punti di partenza multipli; se poniamo $\alpha = 0$ e $\beta \to \infty$ otteniamo il classico algoritmo greedy deterministico (sempre con punti di partenza multipli).

Esaminando l'algoritmo si può rilevare che la complessità computazionale espressa in funzione del numero di formiche m, del numero di città n e del numero di cicli P dopo i quali fermiamo l'algoritmo (dove un ciclo è l'insieme di n passi che porta una formica a visitare tutte le n città ed a ritornare alla città di partenza) è $O(P \cdot (m \cdot n + n^3))$. Infatti, per entrambi gli algoritmi Ant-quantity e Ant-density si ha che:

Il passo 1 è $O(n^2+m)$,
Il passo 2 è $O(n \cdot m+n^3)$ (il passo 2.1 è $O(m)$, il passo 2.2 è $O(n^2)$ e sono ripetuti n volte),
Il passo 3 è $O(n \cdot m+n^2)$

Nei nostri esperimenti (vedi sezione 3.4) abbiamo trovato che è conveniente porre il numero di formiche $m=c_1 \cdot n$, dove c_1 è una costante piccola (il suo valore migliore, ottenuto sperimentalmente, è $c_1=1$). Perciò la complessità dell'algoritmo è $O(P \cdot n^3)$.

3.3 L'algoritmo Ant-cycle

Rispetto ai due precedenti, in questo modello abbiamo introdotto una variazione rilevante. Infatti in Ant-cycle la quantità di traccia depositata $\Delta\tau_{ij}^{k}$ non è calcolata ad ogni passo, bensì solo dopo un ciclo completo (cioè dopo n passi). Il valore di $\Delta\tau_{ij}^{k}(t,t+n)$ è dato da:

$$\Delta\tau_{ij}^{k}(t,t+n) = \begin{cases} \dfrac{Q_3}{L^k} & \text{se la k-esima formica usa l'arco (i,j) nel suo ciclo} \\ 0 & \text{altrimenti} \end{cases} \tag{3.6}$$

dove Q_3 è una costante e L^k è la lunghezza del ciclo della k-esima formica. Ciò corrisponde ad un adattamento dell'algoritmo Ant-quantity, nel quale le tracce sono aggiornate alla fine di un ciclo invece che ad ogni passo. Ci aspettiamo che questo algoritmo abbia prestazioni migliori di quelle dei modelli Ant-density e Ant-quantity, perché in questo caso viene utilizzata informazione globale sull'andamento della ricerca (il valore del risultato ottenuto dalla formica, cioè la lunghezza del suo ciclo).

Il valore della traccia è aggiornato ogni n passi, secondo la seguente equazione (molto simile alla (3.2)):

$$\tau_{ij}(t+n) = \rho\cdot\tau_{ij}(t)+\Delta\tau_{ij}(t,t+n) \tag{3.7}$$

$$\text{dove } \Delta\tau_{ij}(t,t+n) = \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t,t+n) \tag{3.8}$$

L'algoritmo **Ant-cycle** risulta pertanto essere:

Algoritmo Ant-cycle

1 Inizializzazione
    Poni t:=0 {t è il contatore dei passi}
    Per ogni arco (i,j) poni una quantità di traccia iniziale $\tau_{ij}(t)$ e poni $\Delta\tau_{ij}(t,t+n)$:= 0
    Posiziona $b_i(t)$ formiche su ogni nodo i {$b_i(t)$ è il numero di formiche sul nodo i all'istante t}
    Poni s:=1 {s è l'indice della tabu list}
    Per i:=1 fino ad n
        Per k:=1 fino a $b_i(t)$
            tabu$_k$(s):=i {la città di partenza è il primo elemento della tabu list}

2 Ripeti fino a quando la tabu list è piena {questo passo viene ripetuto n-1 volte}
    2.0 Poni s:=s+1
    2.1 Per i:=1 fino ad n {per ogni città i}
        Per k:=1 fino a $b_i(t)$ {per ogni k-esima formica sulla città i}
            Scegli la città j dove andare, con probabilità $p_{ij}(t)$ data dall'equazione (3.1)
            Muovi la k-esima formica in j {questa istruzione crea i nuovi valori $b_j(t+1)$}
            Inserisci il nodo j in **tabu**$_k$(s)

```
3  Per k:=1 fino ad m                          {per ogni formica}
     Calcola L^k utilizzando la rispettiva tabu list
     Per s:=1 fino ad n-1                      {scandisci la tabu list della k-esima formica}
        Poni (h,l):=(tabu_k(s),tabu_k(s+1))
                                               {(h,l) è l'arco che connette (s-1,s) nella tabu list
                                                della formica k}
        Δτhl(t+n):=Δτhl(t+n) + Q3/L^k

4  Per ogni arco (i,j) calcola τij(t+n) secondo l'equazione (3.7)
   Poni t:=t+n
   Per ogni arco (i,j) poni Δτij(t,t+n):=0

5  Memorizza il cammino più breve trovato finora
   Se (P < PMAX) o (comportamento ≠ uni-cammino)     {P è il numero di cicli}
     allora
        Vuota tutte le tabu list
        Poni s:=1
        Per i:=1 fino ad n
          Per k:=1 fino a bi(t)
             tabu_k(s):=i                      {dopo un ciclo ogni formica è nella posizione
                                                iniziale}
        Vai al passo 2
     altrimenti
        Stampa il ciclo più breve trovato e Fermati
```

La complessità dell'algoritmo Ant-cycle è $O(P \cdot m \cdot n^2)$ se fermiamo l'algoritmo dopo P cicli. Infatti si ha che:

Il passo 1 è $O(n^2+m)$
Il passo 2 è $O(m \cdot n^2)$
Il passo 3 è $O(m \cdot n)$
Il passo 4 è $O(n^2)$
Il passo 5 è $O(m \cdot n)$

Anche per l'algoritmo Ant-cycle abbiamo trovato, sperimentalmente, una relazione lineare tra il numero di città e il numero di formiche del tipo trovato per gli algoritmi Ant-density e Ant-quantity (vedi fine della sezione 3.2). Pertanto la complessità dell'algoritmo è $O(P \cdot n^3)$.

Per valutare appieno la complessità rimane ovviamente da determinare la relazione che intercorre tra P ed n, cioè la funzione $P=P(n)$. A tal fine sono stati effettuati degli esperimenti che verranno presentati nella sezione 3.5.6.

3.4 La taratura dei parametri

Non avendo a disposizione un modello formale che permetta di prevedere il comportamento del sistema al variare dei parametri, abbiamo condotto una serie di esperimenti per determinare l'insieme ottimo dei parametri. In questa sezione presentiamo i risultati ottenuti facendo variare, uno per volta, i parametri che influenzano direttamente o indirettamente la probabilità di transizione $p_{ij}(t)$, e cioè: α, β, ρ, Q_h (h=1, 2, 3). In tutti gli esperimenti il valore di default dei parametri è α=1, β=1, ρ=0.7, Q_h=100. Il numero delle formiche è sempre uguale al numero di città ed all'istante iniziale poniamo una formica in ogni città. Abbiamo eseguito gli esperimenti testando i seguenti valori per ogni parametro: $\alpha \in \{0, 0.5, 1, 5\}$, $\beta \in \{0, 0.5, 1, 2, 5, 10\}$, $\rho \in \{0.2, 0.5, 0.7, 0.9\}$ e $Q_h \in \{1, 100, 10000\}$. I risultati degli esperimenti sono riportati dettagliatamente nell'Appendice A. Tutti gli esperimenti riportati qui sono basati sul problema conosciuto in letteratura sotto il nome di Oliver30. Questo problema, i cui dati possono essere trovati in [87], è un problema con trenta città per il quale è nota una soluzione di lunghezza 424.635, trovata usando gli algoritmi genetici[1]. Questo stesso risultato viene trovato dal sistema formiche anche quando i parametri non sono quelli ottimi. Come vedremo in seguito, con l'insieme ottimo[2] di parametri il SF (con il modello Ant-cycle) trova una nuova soluzione migliore di quella pubblicata.

Al fine di permettere il confronto con altri approcci (vedi sezione 3.5.7), la lunghezza dei cicli percorsi dalle formiche è stata calcolata sia come somma di distanze intere che reali, utilizzando il codice in linguaggio C pubblicato nella raccolta di problemi standard [11]. Tutte le prove sono state fermate dopo P=5000 cicli dell'algoritmo (il che significa che nel caso del problema Oliver30, con m=30 formiche, il numero di cicli effettivamente esplorato è in ogni prova 30 x 5.000 = 150.000) e i valori riportati sono ottenuti come media su dieci prove.

Oltre alla lunghezza del ciclo più breve trovato, siamo interessati anche allo studio di una particolare proprietà: il comportamento *uni-cammino*. Questo comportamento si presenta quando tutte le formiche percorrono lo stesso ciclo, il che significa che l'algoritmo non fa più ricerca nello spazio delle soluzioni. Nella ricerca dei valori ottimi dei parametri vogliamo pertanto evitare quegli insiemi di parametri che portano il sistema verso questo comportamento, perché una volta che il sistema ha raggiunto l'uni-cammino non è (statisticamente) più in grado di uscirne e quindi rimane bloccato nella soluzione trovata (che si è spesso verificato essere lontana da quella ottima).

I tre algoritmi mostrano una diversa sensitività ai valori dei parametri.

Nel caso di Ant-density il valore migliore del parametro β risulta essere pari a 10.

β	0	1	2	5	[10]	20
Lungh.media	881.56	456.98	455.52	431.37	426.74	428.79

Le prove effettuate sugli altri parametri mostrano le migliori prestazioni per α=1

α	0	0.5	[1]	2	5
Lungh.media	578.52	464.99	456.98	508.44	695.25

[1] Gli algoritmi genetici sembrano essere una buona euristica per la soluzione di problemi di ottimizzazione combinatoria.

[2] Nel seguito indicheremo con "insieme dei parametri ottimi" i valori dei parametri, ottenuti sperimentalmente, per i quali le prestazioni del sistema sono risultate le migliori.

per ρ che assume il massimo valore possibile (ρ deve avere un valore minore di 1 perché si vuole evitare l'accumularsi infinito della traccia)

ρ	0.3	0.5	0.7	0.9	0.999
Lungh.media	649.86	530.58	456.98	431.31	429.09

per Q_1 qualsiasi. Il valore Q_h {h=1, 2, 3}non sembra infatti influenzare il funzionamento di nessuno dei tre algoritmi.

Inoltre, gli esperimenti con Ant-density mostrano che il sistema presenta il comportamento di uni-cammino solo per valori di $\beta \geq 2$ (di solito questo comportamento appare dopo 200-300 cicli).

Ant-quantity mostra una sensibilità al parametro β ancora molto importante, come si vede dalla tabella di seguito

β	0	1	2	5	10	20	30
Lungh.media	454.72	441.85	436.77	431.60	428.52	427.95	438.83

nella quale si può osservare una diminuzione della lunghezza media dei cicli fintanto che β raggiunge il valore di 20.

Anche in questo caso abbiamo che valori troppo bassi o troppo alti del parametro α possono far peggiorare la performance: il valore migliore si ha per α=0.5.

α	0	0.5	1	5
Lungh.media	649.45	430.70	441.85	478.48

Le prove su ρ mostrano che anche in questo caso (come per Ant-density) il mantenere una forte memoria del passato è la politica migliore

ρ	0.3	0.5	0.7	0.9	0.999
Lungh.media	555.24	451.51	441.85	426.48	426.25

Ant-quantity entra nel comportamento chiamato uni-cammino per $\alpha \geq 1$ e per $1 \leq \beta \leq 10$. In tutti gli altri casi l'attività di esplorazione non si ferma.

Ant-cycle è risultato essere l'algoritmo più efficiente. Il valore migliore di α è risultato essere intorno a 1, quello di β compreso tra 2 e 5 e quello di ρ intorno a 0.5.

I risultati degli esperimenti sono i seguenti

β	0	0.5	1	2	5	10	20
Lungh.media	848.31	452.62	427.44	424.63	424.25	428.35	438.88

α	0	0.5	1	2
Lungh.media	651.27	533.49	427.44	456.11

ρ	0.3	0.5	0.7	0.9
Lungh.media	427.85	426.86	427.44	428.28

Ant-cycle presenta il comportamento chiamato uni-cammino solo per valori di α maggiori di 2.

Osservando il comportamento degli algoritmi al variare dei parametri si può osservare come il parametro ρ sia il solo a presentare un andamento qualitativamente diverso nei tre algoritmi: in Ant-density e Ant-quantity il valore migliore di ρ è il massimo possibile, mentre in Ant-cycle è ρ=0.5. Questo diverso comportamento può essere spiegato come segue. Ant-density e Ant-quantity usano solo informazione strettamente locale, la visibilità η_{ij} (locale per definizione) e la traccia τ_{ij} che, in questi casi, è di nuovo un'informazione di tipo locale: infatti la quantità di traccia depositata su ogni arco è una diretta conseguenza delle decisioni prese utilizzando la regola greedy e non contiene nessun tipo di informazione legata al risultato effettivamente ottenuto (cioè la lunghezza del ciclo) dalla formica che l'ha depositata. Viceversa nell'algoritmo Ant-cycle la traccia depositata è legata alla lunghezza del ciclo (vedi formula (3.6)) e perciò in questo caso l'algoritmo utilizza un'informazione globale (anche se le decisioni sono ancora prese su base locale). In altre parole ciò che succede è che nei primi due modelli l'informazione fornita dalla traccia τ_{ij} altro non è che rafforzamento di quella fornita dalla visibilità, mentre nel modello Ant-cycle τ_{ij} rappresenta un'informazione di tipo diverso da η_{ij}.

Tutti e tre gli algoritmi utilizzano una euristica greedy per guidare le fasi iniziali della computazione, ma è il solo Ant-cycle che sposta, progressivamente col procedere della computazione, l'attenzione sull'informazione di tipo globale fornita dalla traccia. Tutto ciò spiega sia i migliori risultati ottenuti con Ant-cycle che la differenza su citata nel valore ottimo di ρ per questo algoritmo: l'algoritmo ha bisogno di dimenticare parte dell'esperienza accumulata, al fine di poter meglio utilizzare la nuova informazione di tipo globale che viene aggiunta ad ogni ciclo. Negli algoritmi Ant-density e Ant-quantity questa capacità di dimenticarsi del passato non è necessaria poiché il solo tipo di informazione che viene utilizzato per tutta la durata della ricerca è di tipo strettamente locale.

Per il modello Ant-cycle, che ha dato le migliori prestazioni, abbiamo investigato la relazione intercorrente tra performance dell'algoritmo e valori dei parametri nel caso di variazioni a coppie dei parametri α e β (gli altri parametri sono stati posti ai valori: ρ = 0.5 e Q_3 = 100; l'esperimento è stato bloccato dopo P=2500 cicli).

I risultati sono presentati in figura 3.1. Sono state identificate tre zone: per alti valori di α e bassi valori di β l'algoritmo entra molto velocemente nel comportamento uni-cammino senza trovare soluzioni particolarmente buone (questa situazione è rappresentata dal simbolo ■ in figura 3.1). Se l'importanza attribuita alla traccia non è sufficientemente alta (cioè se α ha un valore basso) o se diamo troppa importanza alla *regola greedy* , allora l'algoritmo non è in grado di trovare soluzioni buone nei 2500 cicli a disposizione; dato però che non entra nel comportamento uni-cammino si può ritenere che fornendolo della possibilità di continuare a cercare nello spazio delle soluzioni potrebbe migliorare il risultato trovato: il simbolo utilizzato per questo caso è ∞.

Soluzioni molto buone sono trovate quando α e β si posizionano nella zona centrale: il simbolo utilizzato è ✖. In questo caso abbiamo osservato che diverse coppie di valori per i parametri α e β ((α=1, β=1),(α=1, β=2),(α=1, β=5),(α=0.5, β=5)) determinano livelli di prestazione paragonabili: lo stesso risultato (il ciclo più corto noto sul problema Oliver30) utilizzando approssimativamente le stesse risorse di calcolo (lo stesso numero di cicli).

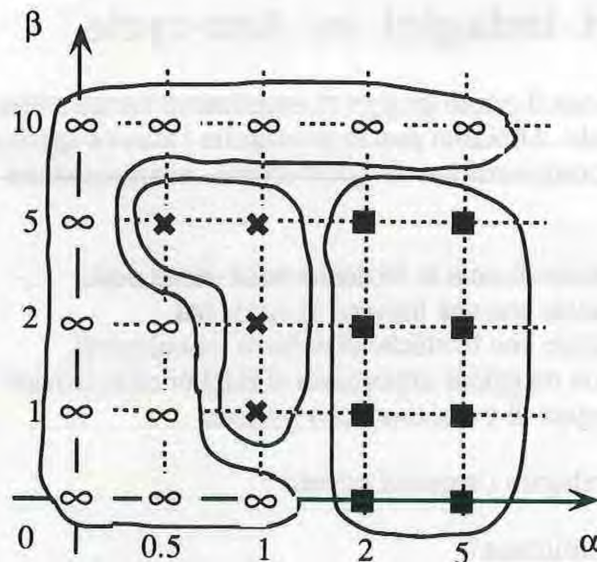


Fig.3.1 - Comportamento di Ant-cycle per diverse combinazioni dei parametri α e β
✖ - l'algoritmo trova soluzioni molto buone e non entra nel comportamento uni-cammino
∞ - l'algoritmo non trova soluzioni buone e non entra nel comportamento uni-cammino
■ - l'algoritmo non trova soluzioni buone ed entra nel comportamento uni-cammino

I risultati ottenuti in questo esperimento possono essere così spiegati: un valore alto di α significa che la traccia è molto importante e pertanto le formiche tendono a ripetere le scelte fatte nel passato. Questo è vero fintanto che β diviene molto elevato: in questo caso anche se il livello di traccia è molto elevato la desiderabilità delle città vicine continua ad essere importante e quindi le formiche conservano una probabilità elevata di scegliere una città vicina piuttosto che di andare dove c'è più traccia.

Alti valori di β o bassi valori di α rendono l'algoritmo molto simile ad un algoritmo greedy stocastico con molti punti di partenza.

3.5 Ulteriori indagini su Ant-cycle

I risultati ottenuti con il primo gruppo di esperimenti hanno messo in luce la superiorità del modello Ant-cycle. Abbiamo perciò proseguito l'attività sperimentale con l'obiettivo di approfondire la comprensione di quest'ultimo, analizzandone il comportamento in varie situazioni:

- distribuzione iniziale di tutte le formiche nella stessa città,
- distribuzione iniziale con una formica in ogni città,
- distribuzione iniziale con formiche distribuite casualmente,
- attribuzione di una maggiore importanza al miglior ciclo trovato fino a quel momento,
- utilizzo di una regola di transizione con rumore.

Abbiamo inoltre indagato i seguenti aspetti:

- quante formiche utilizzare?
- quanti cicli ci vogliono per trovare la soluzione ottima?
- come si comporta il nostro algoritmo in confronto ad euristiche specializzate per il TSP (2-opt, Lin-Kernighan)?

3.5.1 Alcune proprietà generali del modello Ant-cycle

Gli aspetti più interessanti dell'algoritmo Ant-cycle possono essere riassunti nei seguenti punti:

- con i parametri scelti all'interno del loro intervallo di ottimalità l'algoritmo trova sempre soluzioni molto buone, e la maggior parte delle volte trova una soluzione che è migliore della miglior soluzione conosciuta (ottenuta utilizzando gli algoritmi genetici). In Fig.3.2 presentiamo il nuovo ciclo migliore conosciuto trovato con Ant-cycle utilizzando il seguente insieme di parametri: α=1, β=2, ρ=0.5, Q_3=100. Questo ciclo risulta essere di lunghezza 423.74 e presenta due inversioni rispetto alla soluzione (pubblicata in [87]);

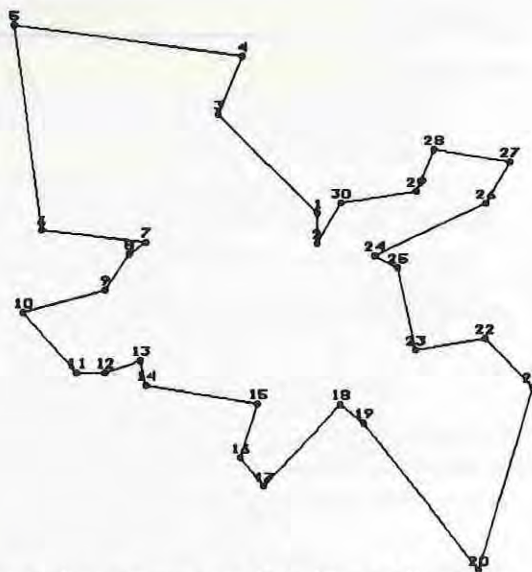


Fig.3.2 - Il nuovo cammino migliore ottenuto per il problema Oliver30 con 342 iterazioni dell'algoritmo Ant-cycle (α=1, β=2, ρ=0.5, Q_3=100, e=4).
Lunghezza reale = 423.741.
Lunghezza intera = 420.

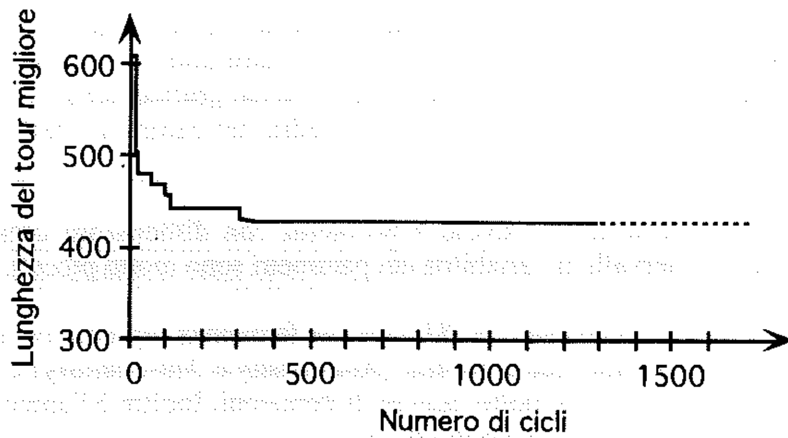


Fig.3.3 - L'algoritmo trova cicli molto buoni molto velocemente e il nuovo ciclo migliore (423.741) dopo 342 iterazioni. In questa figura, così come in Fig.3 e 4, ad ogni iterazione vengono esplorati m cicli completi.

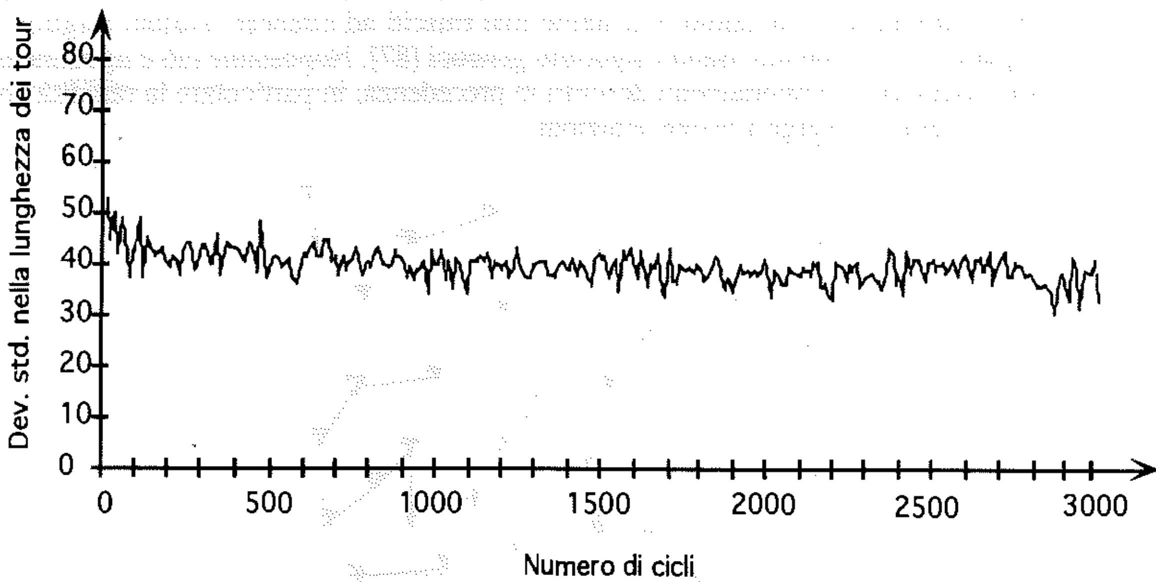


Fig.3.4 - La varianza nella lunghezza del ciclo fatto dalle varie formiche mostra come l'algoritmo continui a ricercare nuove possibili soluzioni migliori di quella trovata finora.

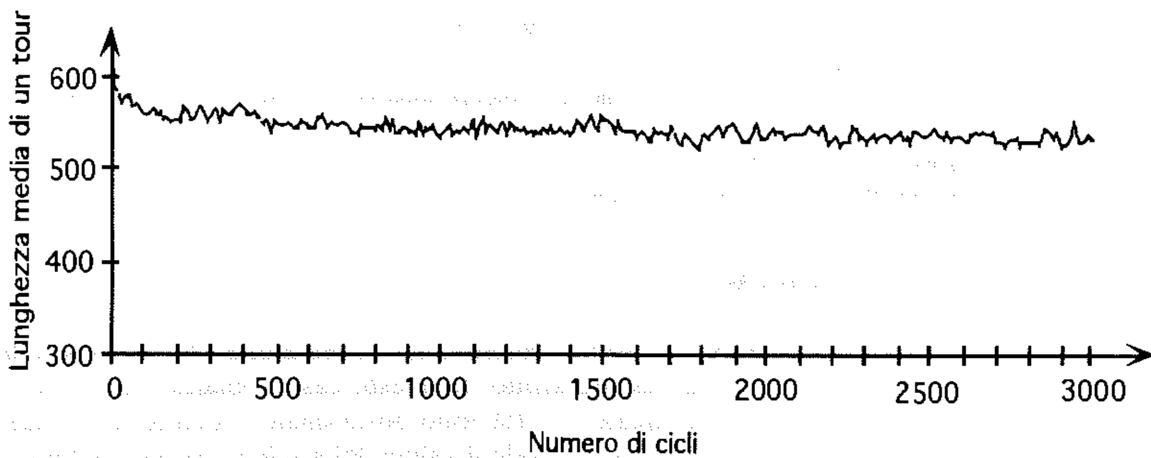


Fig.3.5 - Lunghezza media del ciclo (i.e. $\sum_{k=1}^{m} \frac{L^k}{m}$).

- l'algoritmo trova soluzioni molto buone molto velocemente (vedi Fig.3.3); ciò nonostante non presenta il comportamento uni-cammino, ma continua a cercare nello spazio delle soluzioni. Si può osservare tutto ciò dal grafico della deviazione standard in Fig.3.4 e dal grafico della lunghezza media del cammino della formica media (Fig.3.5);
- applicando l'algoritmo Ant-cycle a problemi con dimensioni crescenti abbiamo trovato che gli intervalli di variabilità dei parametri sono molto piccoli.

Ant-cycle trova il risultato citato in [87] con una frequenza significativamente più alta di quanto non facciano gli altri due algoritmi (Ant-density e Ant-quantity) e questo risultato viene ottenuto con un numero molto minore di iterazioni. Inoltre è l'unico degli algoritmi a riuscire a trovare il nuovo risultato migliore.

Finora l'algoritmo è stato solo parzialmente sperimentato sui problemi Eilon50 e Eilon75 (si tratta di due problemi con rispettivamente 50 e 75 città; per i dati vedi [44]) facendo solo pochi esperimenti e limitando il numero massimo di iterazioni a 3000 (Fig.3.6). Sotto queste condizioni non siamo mai riusciti ad ottenere risultati migliori o anche uguali a quelli ottenuti usando algoritmi genetici [87]. Nonostante ciò è ugualmente possibile osservare i comportamenti descritti in precedenza, in particolare la rapidità con la quale il sistema converge a buone soluzioni.

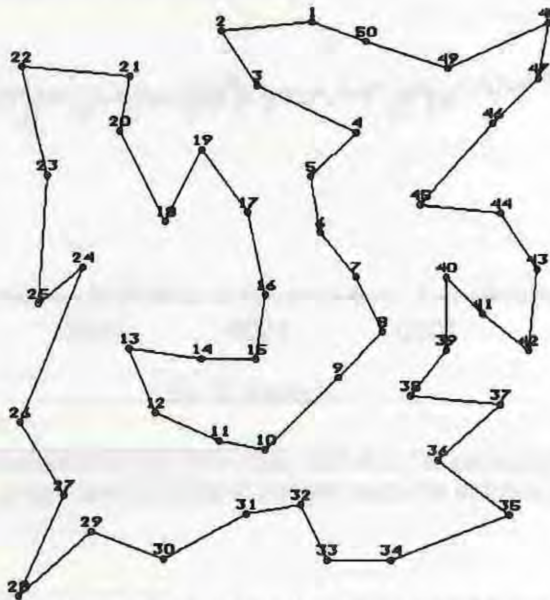


Fig.3.6 - Evoluzione tipica di Ant-cycle applicato al problema Eilon50 (α=1, β=2, ρ=0.7, Q_3=100).
Lunghezza reale= 441.572.
Lunghezza intera = 438.
Soluzione migliore conosciuta: lunghezza intera = 428.

3.5.2 Numero di formiche

Con questo esperimento abbiamo voluto determinare la relazione che intercorre tra numero di formiche ed efficienza dell'algoritmo. In questo caso abbiamo utilizzato un problema di dimensione n=16 nel quale le città sono posizionate secondo una griglia quadrata con quattro città per lato (in questo modo il valore della soluzione ottima è noto a priori e vale 160 se la distanza fra due città appartenenti allo stesso lato è posta uguale a 10 - vedi Fig.3.7).

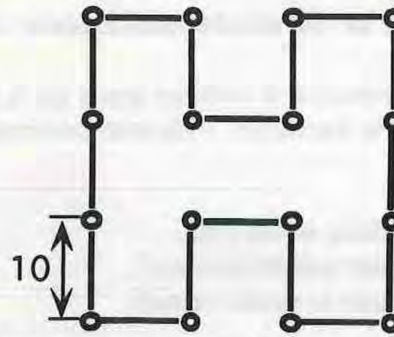


Fig.3.7 - Una soluzione ottima per il problema con griglia 4 x 4 (n=16).

I risultati dell'esperimento sono mostrati in Fig.3.8: sull'asse delle ascisse è riportato il numero di formiche utilizzato, mentre in ordinata è riportato il numero medio (su dieci esperimenti) di iterazioni richiesto per raggiungere l'ottimo (se questo viene raggiunto entro 2000 iterazioni) moltiplicato per il numero delle formiche che sono state impiegate (in questo modo il valore sulle ordinate rappresenta l'effettivo sforzo computazionale effettuato dall'algoritmo).

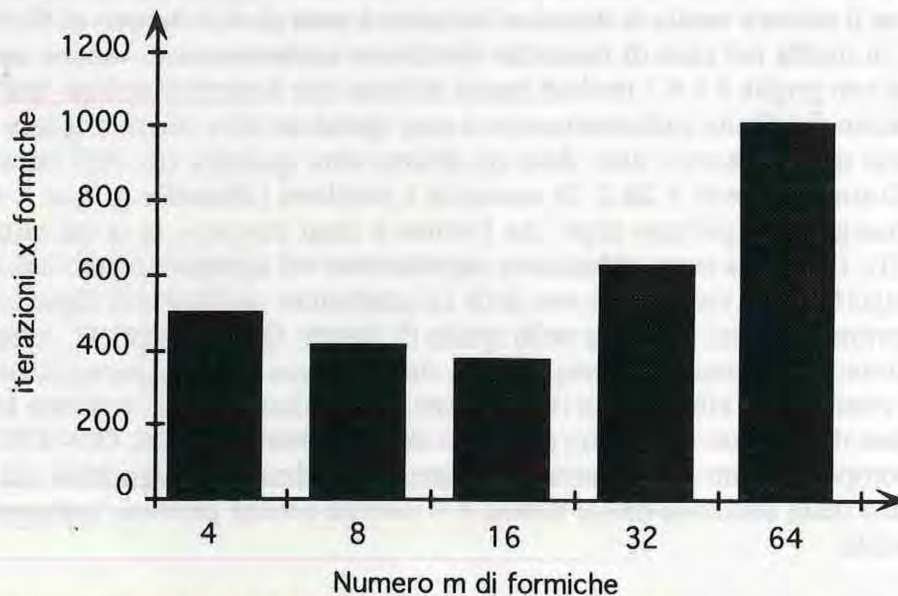


Fig.3.8 - Numero di iterazioni_x_formiche necessario per raggiungere l'ottimo in funzione del numero totale di formiche per il problema con griglia 4 x 4

È interessante notare che:

- l'algoritmo ha sempre trovato il valore ottimo per m≥4 (m = numero di formiche);
- esiste un valore ottimo per il numero di formiche da impiegare;
- test effettuati su un insieme di problemi con griglia di dimensioni r x r (r = 4, 5, 6, 7, 8) hanno mostrato che il valore ottimo del numero di formiche risulta essere molto vicino al numero di città (m≈n).

Un secondo insieme di esperimenti è stato fatto utilizzando un problema con 16 città distribuite in modo casuale. Anche in questo caso il risultato è stato che la migliore performance si ottiene utilizzando 8÷16 formiche, cioè di nuovo con un numero di formiche dell'ordine delle dimensioni del problema.

3.5.3 Come posizionare le formiche nell'istante iniziale?

Obiettivo di questo esperimento è di stabilire quale sia la politica migliore da seguire nel posizionamento iniziale delle formiche. Abbiamo confrontato i risultati ottenuti con tre diverse politiche:

(i) tutte le formiche partono dalla stessa città,
(ii) le formiche sono distribuite uniformemente[3],
(iii) le formiche sono distribuite in modo casuale.

Per questo esperimento sono stati utilizzati il problema della griglia 4 x 4, il problema di 16 città generate in modo casuale, il problema Oliver30. In tutti i casi la politica migliore è risultata essere la (iii), seguita da vicino dalla (ii).

Nel caso del problema con 16 città posizionate a caso (16rand) abbiamo condotto 16 esperimenti (ognuno ripetuto 5 volte, con differenti semi per il generatore di numeri pseudo casuali) nei quali tutte le formiche erano posizionate, inizialmente, sulla stessa città (nel primo esperimento sulla città 1, nel secondo sulla città 2 e così via). Il risultato ottenuto è stato che in tutti i casi le formiche sono state in grado di trovare la soluzione ottima, ma il numero medio di iterazioni richiesto è stato di 64.4 rispetto ai 57.1 iterazioni richiesti in media nel caso di formiche distribuite uniformemente. Anche nel caso del problema con griglia 4 x 4, i risultati hanno indicato che il comportamento dell'algoritmo con formiche distribuite uniformemente o a caso (politiche (ii) e (iii)) è migliore rispetto al caso in cui queste partono tutte dalla medesima città (politica (i): 26.9 iterazioni[4] per trovare l'ottimo rispetto a 28.2. In entrambi i problemi (16rand e griglia 4 x 4), se si lascia proseguire l'algoritmo dopo che l'ottimo è stato trovato e se si sta utilizzando la politica (i), il sistema entra abbastanza rapidamente nel comportamento uni-cammino. Questo significa che viene persa una delle caratteristiche positive dell'algoritmo: il fatto che l'algoritmo continui a cercare nello spazio di ricerca. Questa proprietà, come già detto nella sezione 3.4, è importante perché evita che il sistema rimanga intrappolato in ottimi locali. A conferma di ciò viene un risultato ottenuto applicando il SF con tutte le formiche che partono dalla stessa città ad un problema di dimensioni maggiori, Oliver30: in questo caso il comportamento di uni-cammino si presenta prima che l'algoritmo sia riuscito a trovare una delle soluzioni molto buone, e il sistema rimane pertanto intrappolato in un ottimo locale.

In tabella 3.1 riportiamo per esempio alcuni risultati ottenuti facendo partire le tutte le formiche dalla stessa città. Nella prima colonna è riportato il nome della città di partenza, nella seconda il ciclo migliore trovato (fermando l'algoritmo dopo 2000 iterazioni), nella terza l'iterazione alla quale tale ciclo è stato effettivamente trovato e nella quarta l'iterazione dopo la quale si può osservare l'uni-cammino.

Per quanto riguarda la differenza tra distribuzione uniforme e casuale, i risultati sono i seguenti: distribuendo le formiche in modo casuale i risultati ottenuti sono lievemente migliori (se si misura la performance in iterazioni_x_formiche, nel problema con griglia 4 x 4 e con 16 formiche la distribuzione casuale ha richiesto 370 iterazioni, mentre la distribuzione uniforme ne ha richieste 448; con 32 formiche la distribuzione casuale ha richiesto 640 iterazioni, mentre la distribuzione uniforme ne ha richieste 650).

---

[3] Diciamo che le formiche sono distribuite uniformemente se inizialmente su ogni città c'è lo stesso numero di formiche.

[4] Affinché sia possibile paragonare questi risultati con quelli presentati nella sezione precedente il numero di iterazioni deve essere moltiplicato per il numero di formiche (in questo caso m=16).

Tab.3.1 - Risultati ottenuti applicando Ant-cycle al problema Oliver30 nel caso di tutte le formiche che partono dalla stessa città.

Città di partenza	Lunghezza del ciclo migliore	Iterazione in cui è stato trovato il ciclo migliore	Iterazione in cui inizia l'uni-cammino
1	432.586	98	571
7	440.299	183	183
13	440.063	449	449
25	443.813	5	238
distr.unif.	424.635	1583	//

3.5.4 Strategia elitista

Chiamiamo strategia elitista[5] una versione modificata dell'algoritmo nella quale ad ogni iterazione il valore della traccia viene incrementato, oltre che nel solito modo, anche da una formica speciale (detta appunto formica elitista) che deposita una quantità di traccia $e \cdot Q_3/L^*$ sul ciclo che è risultato fino a quel momento essere il migliore: e risulta pertanto essere un nuovo parametro del sistema, e come tale è stato oggetto di esperimenti per valutarne il valore migliore. L'idea sottostante all'introduzione di questa formica elitista è che dando maggiore importanza al risultato migliore trovato finora, si possa far convergere il sistema più velocemente verso soluzioni buone.

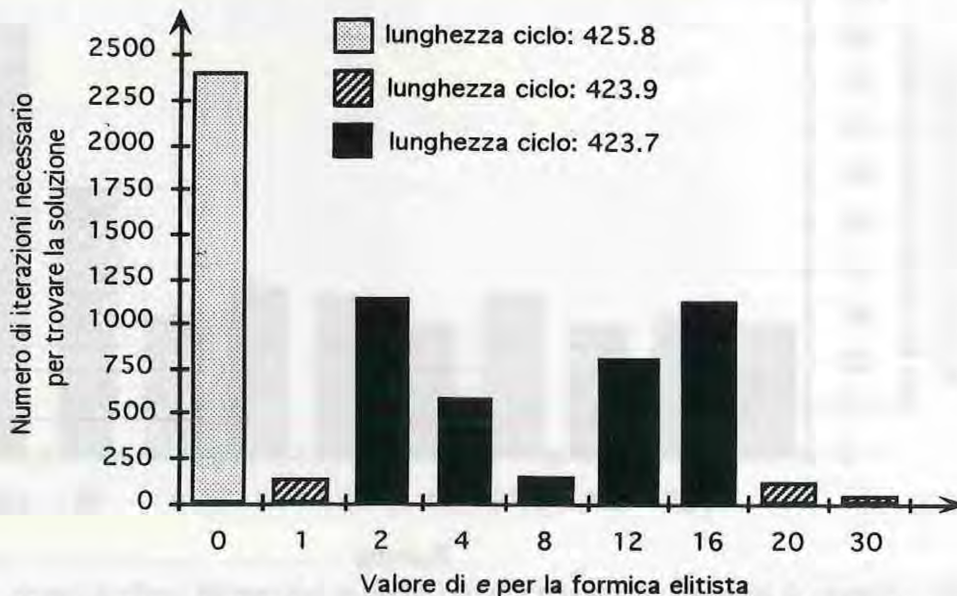


Fig.3.9 - Numero di iterazioni richiesto per trovare un ottimo locale in funzione del valore di e

I risultati sperimentali, sul problema Oliver30 (l'algoritmo è stato fermato dopo P_{MAX}=2500 iterazioni), sembrano confermare questa ipotesi. Risulta infatti che esiste un valore ottimo per il parametro e. Allontanandosi da questo valore si verificano due differenti comportamenti, a seconda di quanto grande è lo spostamento: per valori di e vicini a tale valore si ottiene lo stesso il ciclo migliore conosciuto, mettendoci però più

[5] Abbiamo chiamato questa strategia *elitista* perché richiama alla mente la strategia elitista utilizzata negli algoritmi genetici, dove consiste nel conservare con probabilità uno il migliore individuo di ogni generazione. Nel nostro caso l'effetto di un individuo, cioè di una formica con la sua tabu list che identifica il tour da lui seguito, è quello di incrementare il valore della traccia sugli archi che appartengono al suo tour. Pertanto l'equivalente del salvare il miglior individuo degli algoritmi genetici è nel nostro caso il sommare il contributo (eventualmente moltiplicato per una opportuna costante) della formica che ha fatto il tour migliore a quello di tutte le altre formiche.

tempo. Se invece lo scostamento è maggiore il sistema non è più in grado di ottenere il ciclo migliore e sembra rimanere intrappolato in un minimo locale. In Fig.3.9 è riportato un andamento tipico del numero di iterazioni necessario per trovare cicli di varia bontà in funzione di *e*.

3.5.5 Probabilità di transizione con rumore

Questo esperimento è stato effettuato per valutare l'effetto che può avere sul processo di ricerca l'introduzione di un rumore nel calcolo della probabilità di transizione. Il motivo per cui si è pensato di introdurre del rumore nella regola probabilistica di transizione è che si voleva provare un qualche metodo per contrastare l'eventuale comparsa prematura del comportamento uni-cammino. Come abbiamo visto precedentemente, quella di non entrare nel comportamento uni-cammino (o per lo meno di entrarvi non troppo presto), è una proprietà importante del nostro sistema.

I risultati sperimentali hanno confermato la nostra ipotesi che l'inserimento di rumore potesse avere solo effetti irrilevanti: valori bassi di rumore non influenzano significativamente la performance del sistema mentre valori molto elevati hanno un effetto negativo.

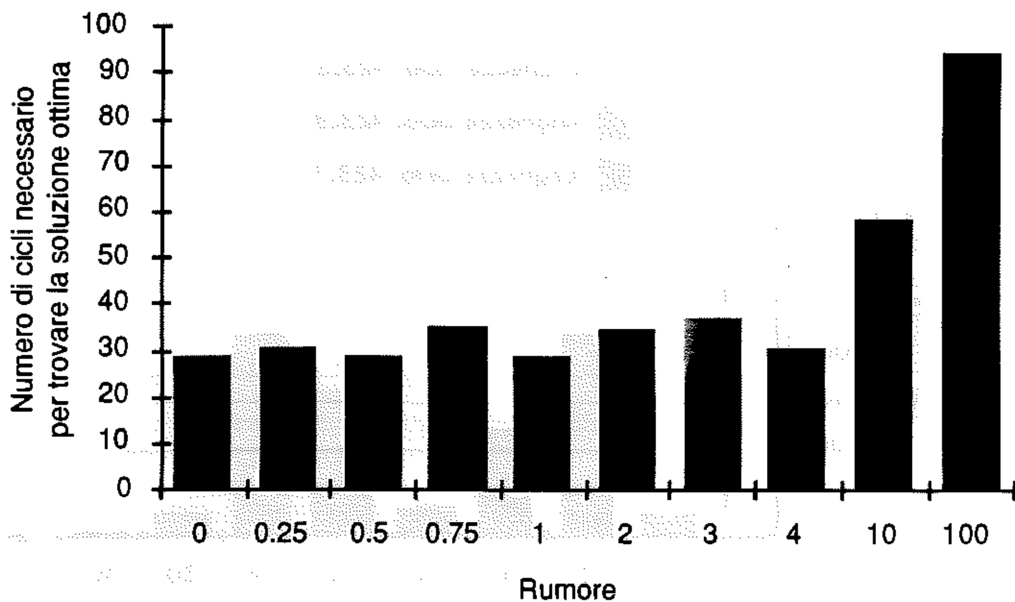


Fig.3.10 - Numero di iterazioni richiesto per trovare l'ottimo in funzione del livello di rumore

La probabilità di transizione con rumore da noi utilizzata è data dalla formula seguente

$$p_{ij}(t) = \begin{cases} \dfrac{[\tau_{ij}(t)\cdot(1 + \varepsilon_{ij}(\sigma))]^{\alpha}\cdot[\eta_{ij}]^{\beta}}{\sum\limits_{j\in J}[\tau_{ij}(t)\cdot(1 + \varepsilon_{ij}(\sigma))]^{\alpha}\cdot[\eta_{ij}]^{\beta}} & \text{se } j\in J \\ 0 & \text{altrimenti} \end{cases} \qquad (3.9)$$

dove $J=\{j: j\notin \mathbf{tabu_k}\}$.

Gli esperimenti sono stati effettuati utilizzando il problema con griglia 4 x 4: in Fig.3.10 sono riportati in ordinata il numero medio di iterazioni (su dieci esperimenti) necessario per trovare l'ottimo, mentre in ascissa è riportato il valore del rumore (espresso come deviazione standard della variabile casuale gaussiana ε_{ij} usata nella formula (3.9)).

3.5.6 Tempo necessario per trovare la soluzione ottima

Il calcolo della complessità dell'algoritmo presentato nella sezione 3.3 non ci dice nulla riguardo il tempo effettivamente necessario per raggiungere l'ottimo. Infatti la formula ricavata per esprimere la complessità, $O(P\cdot n^3)$, contiene il termine P la cui eventuale dipendenza da n non è nota. Gli esperimenti riportati in questa sezione vogliono essere una prima valutazione del rapporto che intercorre tra P ed n, cioè di P=P(n). La tabella 3.2 riporta i risultati ottenuti nel caso di problemi con griglia di dimensioni crescenti (problemi con griglia r x r, r = 4, 5, 6, 7, 8, e con lato di lunghezza 10 come in Fig.3.7; si noti che n = r x r).

Tab.3.2 - Tempo necessario per trovare la soluzione ottima in funzione della dimensione del problema

Problema r x r	Soluzione migliore	Dimensione relativa dello spazio di ricerca	Numero medio di iterazioni necessarie per trovare l'ottimo	Tempo richiesto per trovare l'ottimo[6] (secondi)
4 x 4	160	1	5.6	8
5 x 5	254.1	$\approx 10^{11}$	13.6	75
6 x 6	360	$\approx 10^{28}$	60	1020
7 x 7	494.1	$\approx 10^{49}$	320	13440
8 x 8	640	$\approx 10^{75}$	970	97000

È interessante notare che per problemi fino a 64 città Ant-cycle ha sempre trovato la soluzione ottima. Inoltre il numero di iterazioni necessario e il tempo di calcolo richiesto per trovare la soluzione ottima crescono molto più lentamente della dimensione dello spazio di ricerca: ciò suggerisce di nuovo che l'algoritmo utilizzi una strategia di ricerca molto efficiente. Infatti la dimensione dello spazio di ricerca cresce approssimativamente come $[(n-1)!]/2$, mentre il tempo necessario per trovare l'ottimo cresce esponenzialmente con $10^{(n-3)}$ (valore ottenuto interpolando i dati dell'ultima colonna, dove si può osservare come il tempo richiesto per trovare la soluzione ottima aumenti ad ogni riga di un ordine di grandezza).

[6] I test sono stati fatti su dei PC Ibm-compatibili.

3.5.7 Paragoni con altri algoritmi specializzati per il TSP

Abbiamo paragonato i risultati ottenuti con Ant-cycle con quelli ottenuti per mezzo di altri algoritmi specializzati per la soluzione del TSP. Il problema utilizzato per il paragone è stato di nuovo Oliver30. Per questo esperimento abbiamo utilizzato il pacchetto software Travel [11]. Questo pacchetto software, come del resto quasi tutti gli algoritmi che si cimentano con il TSP, utilizza distanze intere: pertanto, per poter fare dei paragoni significativi, abbiamo implementato nel nostro sistema lo stesso modo di calcolare le distanze. I risultati sono mostrati nella tabella 3.3, dove nella prima colonna c'è il nome dell'algoritmo utilizzato, nella seconda la lunghezza del ciclo migliore ottenuto, in terza colonna il valore del ciclo migliore ottenuto applicando l'euristica di miglioramento nota con il nome 2-opt (si tratta di una ricerca esaustiva su tutte le permutazioni ottenibili da quella di partenza scambiando tra loro coppie di città) e in quarta colonna il valore del ciclo migliore ottenuto applicando l'euristica di Lin e Kernighan [68].

Si noti come i risultati ottenuti da Ant-cycle siano migliori di quelli ottenuti da qualunque altro algoritmo considerato anche dopo l'applicazione dell'euristica di miglioramento 2-opt. Utilizzando Lin-Kernighan si trova a volte lo stesso risultato trovato con Ant-cycle, a volte un risultato meno buono ma molto vicino. È opportuno precisare però che il tempo di calcolo richiesto da Lin-Kernighan è significativamente minore.

Tab.3.3 - Risultati di Ant-cycle paragonati a quelli ottenuti con altri algoritmi.

	Risultato	2-opt	Lin-Kernighan[7]
Ant-cycle	420	-	-
Near Neighbour	587	437	420-421
Far Insert	428	421	420-421
Near Insert	510	492	420-421
Space Filling Curve	464	431	420-421
Sweep	486	426	420-421
Random	1212	663	420-421

Come commento generale a tutti gli esperimenti vorremmo sottolineare il fatto che l'algoritmo Ant-cycle, quando utilizzi parametri ottimi (ad esempio: $\alpha=1$, $\beta=2$, $\rho=0.5$, $Q_3=100$, $e=5$) trova quasi sempre la soluzione migliore nota, e trova molto rapidamente soluzioni molto buone (normalmente una soluzione di lunghezza <430 viene trovata in meno di 100 iterazioni, e la miglior soluzione nota, di lunghezza 423.74, in meno di 400 iterazioni).

In ogni caso l'esplorazione da parte dell'algoritmo continua, come è testimoniato dall'andamento del grafico della varianza e dal fatto che il valore medio della lunghezza dei cicli fatti dalle formiche rimane sempre maggiore del valore del ciclo migliore trovato.

Abbiamo inoltre testato l'algoritmo sul problema BAYG29 [73], problema con 29 città per il quale è nota la lunghezza del ciclo ottimo, ritrovando sempre la soluzione ottima.

[7] L'euristica di miglioramento Lin-Kernighan trova soluzioni di lunghezza 420 o 421 a seconda della soluzione di partenza a cui è applicata.

3.6 Applicazioni ad altri problemi

Dedichiamo questo paragrafo ad una breve discussione sulla applicabilità del modello "formiche" ad altri problemi. Affinché il SF possa essere applicato ad un problema di ottimizzazione combinatoria si deve identificare una adeguata rappresentazione per:

(i) il problema (che deve essere modellato in modo da sottintendere un grafo nel quale "si muovono" alcuni agenti);
(ii) il processo autocatalitico[8];
(iii) l'euristica che permette la realizzazione costruttiva di una soluzione (che potremmo chiamare "forza greedy");
(iv) il metodo di soddisfacimento dei vincoli (vedi la tabu list).

Nella nostra ricerca abbiamo finora definito dei SF specifici per il problema dell'assegnamento quadratico (QAP - vedi [13], [66]) e per il problema della soddisfacibilità di una formula booleana (SAT - vedi [49]): in entrambi i casi le maggiori difficoltà si incontrano nella definizione dei punti (i) e (iii), legate al fatto che non tutti i problemi hanno una rappresentazione standard che risulti subito adatta all'applicazione di un SF. Si consideri ad esempio il problema SAT, definito come segue: data una formula booleana in forma congiunta di n variabili, si trovi un assegnamento di verità, cioè si attribuisca ad ogni variabile il valore 0 o 1, tale che la funzione booleana valga 1. Allora il punto (i) può essere affrontato costruendo un grafo completo composto da 2n nodi, nel quale n nodi ($x_1, ..., x_j, ..., x_n$) rappresentano l'assegnamento alla rispettiva variabile j-esima del valore 1 e gli altri n nodi ($\overline{x}_1, ..., \overline{x}_j, ..., \overline{x}_n$) del valore 0. Per il punto (iii) varie euristiche sono attualmente sotto esame: si potrebbero ad esempio considerare più desiderabili quei nodi che determinano il soddisfacimento del più alto numero di clausole. In figura 3.11 è riportato il grafo di SAT per un problema con 4 variabili, mentre in figura 3.12 è mostrata una possibile soluzione finale, nella quale il ciclo tocca x_j o $\overline{x}_j$. La soluzione dell'esempio in figura 3.12 è pertanto: $x_1=1, x_2=0, x_3=1, x_4=0$.

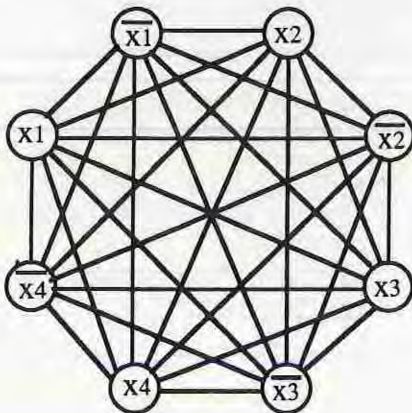


Fig.3.11 - Il grafo di SAT

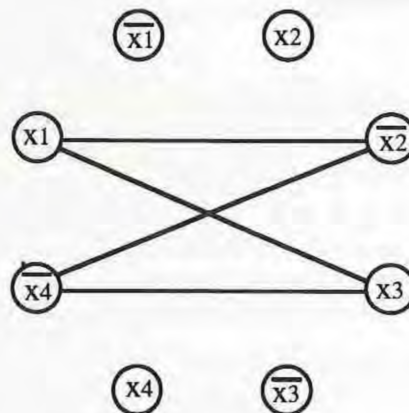


Fig.3.12 - Una possibile soluzione per un problema con 4 variabili

[8] Il processo autocatalitico, che abbiamo introdotto nella sezione 2.2.1, è reso possibile dall'utilizzo della traccia e della regola di transizione probabilistica.

3.7 Conclusioni e lavoro futuro

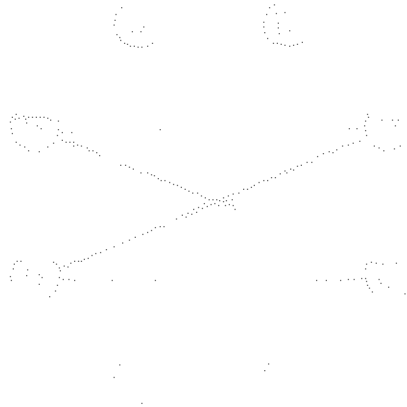
In questo capitolo abbiamo esposto un nuovo algoritmo basato su di una metafora naturale: quella delle colonie di formiche. Ovviamente il nostro modello si discosta notevolmente da un modello che volesse riprodurre fedelmente le capacità e le caratteristiche di un formicaio vero, e quindi abbiamo utilizzato questa metafora prevalentemente come meccanismo ispiratore e come strumento didattico per esporre le nostre idee.

Ne è derivata una metodologia di ottimizzazione su grafo basata sull'interazione di tre idee base:

(i) la distribuzione di uno stesso compito elementare a molti agenti interagenti, senza che nessuno degli agenti abbia una visione complessiva del sistema;
(ii) l'utilizzo di retroazione positiva (autocatalitica) per accelerare il processo di calcolo;
(iii) la separazione del processo di calcolo in due parti, una prima parte durante la quale il processo viene guidato prevalentemente da una euristica di tipo greedy ed una seconda parte durante la quale il sistema utilizza prevalentemente l'esperienza accumulata nel passato.

Il lavoro di ricerca è naturalmente suscettibile di approfondimenti nelle seguenti direzioni:

- valutazione della generalità del SF per mezzo della sua applicazione ad un certo numero di problemi di ottimizzazione combinatoria;
- studio dell'applicabilità di alcune delle idee proposte a problemi del settore di ricerca sull'apprendimento automatico;
- valutazione della scalabilità del SF su diverse architetture parallele (transputer, Connection Machine);
- valutazione analitica di proprietà dell'algoritmo (convergenza, stabilità delle soluzioni).



4

L'algoritmo genetico e il problema dell'orario scolastico

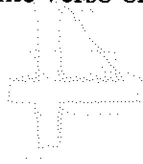
4.1	Introduzione
4.2	Il problema dell'orario scolastico
4.3	Gli algoritmi genetici e il problema dell'orario scolastico
4.4	La funzione obiettivo e la fitness function
4.5	Complessità dell'algoritmo
4.6	Risultati
4.7	Conclusioni e sviluppi futuri

4.1 Introduzione

Obiettivo della ricerca presentata in questo capitolo è quello di valutare i limiti e le potenzialità degli algoritmi genetici - AG - nell'affrontare problemi di ottimizzazione combinatoria "molto vincolati", problemi cioè in cui spostamenti anche minimi da una soluzione ammissibile possono generare soluzioni non ammissibili. Come applicazione abbiamo scelto il problema dell'orario (Time Table Problem - TTP), un problema noto per essere NP-hard [45] ma di notevole rilevanza pratica [1], [85], [18], [27], [50], [59], [60].

Come già detto, il nostro scopo è di mostrare che gli algoritmi genetici possono essere applicati con successo a problemi difficili e fortemente vincolati e che il loro uso può fornire una soluzione elegante e funzionale a problemi reali.

Nel capitolo viene dapprima presentato il problema dell'orario, poi vengono presentate le modifiche apportate all'algoritmo genetico classico affinché possa trattare in modo corretto il problema, infine uno schema dell'algoritmo, la struttura della *fitness function* ed alcuni risultati sperimentali che mostrano come l'algoritmo proposto converga effettivamente verso orari giudicati "buoni" dall'utente[1].



[1] Definire cosa si intende con orario "buono" è difficile. Dal punto di vista dell'utente un orario è buono se risulta essere di qualità almeno comparabile con quelli degli anni precedenti. Dal punto di vista dell'algoritmo genetico la bontà dell'orario è valutata per mezzo della *fitness function* da noi definita tenendo conto delle caratteristiche del problema.

4.2 Il problema dell'orario scolastico

Il problema della formulazione di un orario scolastico per una scuola secondaria superiore è normalmente ristretto alla formulazione di più orari interdipendenti, uno per ogni coppia di sezioni presenti nella scuola.

Sulla base di questa ipotesi, i dati su cui lavorare sono:

- una lista dei docenti della coppia di sezioni (20 nel nostro caso);
- una lista delle classi interessate (10 per la coppia di sezioni);
- una lista delle ore di lezione settimanali per ogni classe (30);
- un "curriculum" per ciascuna classe, cioè un elenco delle lezioni che ogni docente deve tenere in ogni classe;
- alcune "condizioni esterne", per esempio le ore che i docenti condivisi con altre coppie di sezioni devono svolgere all'esterno della coppia considerata, oppure le ore da svolgere in condizioni particolari (palestre, laboratori, ...).

La rappresentazione formale del TTP applicato ad un orario scolastico risulta essere quindi la seguente:

ISTANZA: Una quadrupla $< \mathbf{D}, \mathbf{O}, \mathbf{L}, f >$, dove

D è un insieme $\{D_1, \ldots, D_i, \ldots, D_m\}$ di risorse (docenti),
O è un insieme $\{O_1, \ldots, O_j, \ldots, O_n\}$ di intervalli (ore),
L è un insieme di compiti da assolvere (lezioni che ogni docente deve svolgere in ciascuna classe),
f è una funzione obiettivo da minimizzare, $f: \mathfrak{R} \Rightarrow \mathbb{R}$,
dove $\mathfrak{R}$ è l'insieme delle matrici $\mathbf{R}=[r_{ij}]$ di dimensioni m x n con $r_{ij} \in \mathbf{L}$ (nota che un "orario" è un elemento di $\mathfrak{R}$),

QUESTIONE: minimizzare f

Il problema è stato affrontato con tecniche di programmazione lineare a numeri interi, introducendo opportune euristiche [86]; se infatti lo si affrontasse con gli algoritmi standard, ad esempio definendo variabili binarie x_{ijk} (dove, secondo i parametri definiti in precedenza, i identifica un insegnante, j identifica un'ora settimanale, k identifica una classe), il problema da noi trattato verrebbe rappresentato da 6000 variabili (i=1,...,20; j=1,...,30; k=1,...,10), il che lo renderebbe computazionalmente intrattabile ([2], [26], [67]), implicando, nel caso di utilizzo di un algoritmo "forza bruta", uno spazio di ricerca con 2^{6000} nodi da esplorare.

Nella prossima sezione presenteremo un metodo di risoluzione basato sull'uso degli algoritmi genetici opportunamente modificati (vedi anche [25], [19], [20], [21]).

4.3 Gli algoritmi genetici e il problema dell'orario scolastico

Come visto nella sottosezione 2.3.2, i problemi che si incontrano quando si tenti di utilizzare gli algoritmi genetici per la soluzione di problemi di ottimizzazione combinatoria sono solitamente legati al fatto che gli operatori standard non funzionano più. Infatti una applicazione di tali operatori porta alla generazione di soluzioni non ammissibili. Nel caso del TTP diciamo, in modo informale, che la matrice $\mathbf{R}_1$ rappresenta un orario ammissibile se soddisfa i seguenti vincoli:

- ogni insegnante e ogni classe devono essere presenti nell'orario con un numero prestabilito di ore,
- in ogni classe non si possono avere più insegnanti nella stessa ora,
- ogni insegnante non può essere presente nella stessa ora in più classi,
- le ore "scoperte" (cioè le ore senza lezione precedute e seguite nello stesso giorno da ore di lezione) non sono ammesse per le classi.

Si dice invece non ammissibile se non ne soddisfa almeno uno. Il numero di inammissibilità $\#_{in}(\mathbf{R}_1)$ indica il numero di volte che una condizione di ammissibilità è violata.

Il problema delle inammissibilità è stato da noi affrontato sia ridefinendo gli operatori di crossover e di mutazione, che per mezzo dell'introduzione di un operatore detto *genetic repair*.

In questa sezione affrontiamo il problema di dare una rappresentazione di un orario scolastico, per una coppia di sezioni di una scuola secondaria superiore, utile per essere manipolata dalla variante da noi proposta dell'algoritmo genetico.

Sia dato un alfabeto $\mathbf{L} = \{1,2,3,4,5,6,7,8,9,0,D,A,S,\blacklozenge,-\}$. La scelta dell'alfabeto $\mathbf{L}$ permette di rappresentare il problema mediante una matrice $\mathbf{R}_1$ di dimensioni m x n, con $r_{ij} \in \mathbf{L}$ (vedi Fig.4.1), dove l'elemento r_{ij} della matrice indica l'attività del docente i-esimo nell'ora j-esima. Ogni elemento r_{ij} della matrice $\mathbf{R}_1$ è un gene il cui valore allelico può variare sul sottoinsieme di $\mathbf{L}$ caratteristico dell'insegnante i-esimo cui si riferisce la riga. Abbiamo indicato:

- $r_{ij} = s$, $s \in \{1,2, \ldots ,9,0\}$ significa che il docente i-esimo nell'ora j-esima è occupato con l'attività s (i simboli 1,2, ..., 9,0, denotano l'attività di insegnamento in una delle 10 classi, per esempio 1ªA,2ªA, ... , 4ªB,5ªB, della coppia di sezioni considerate);
- $r_{ij} = D$ significa che il docente i-esimo pone l'ora j-esima a disposizione per eventuali supplenze;
- $r_{ij} = A$ significa che il docente i-esimo dedica l'ora j-esima ad attività di aggiornamento;
- $r_{ij} = S$ significa che il docente i-esimo pone l'ora j-esima a disposizione per lezioni in classi non appartenenti alla coppia di sezioni considerate;
- $r_{ij} = \blacklozenge$ significa che il docente i-esimo non è occupato in attività scolastiche nell'ora j-esima;
- con i caratteri ----- si indica il giorno libero.

Insegnante - Materia	Lun	Mar	Mer	Gio	Ven	Sab
1 • Lettere-1	♦11♦1	112♦♦	♦♦♦11	2212♦	11111	———
2 • Lettere-2	♦6♦♦6	7777♦	♦♦♦77	66♦♦7	———	7777♦
3 • Lettere-3	♦♦♦2♦	666♦6	2♦♦22	———	6266♦	6622♦
4 • Lettere-4	♦8♦♦♦	44♦♦♦	———	♦♦4♦8	84888	88444
5 • Lettere-5	———	♦5555	♦♦355	♦♦353	3♦33♦	33♦♦♦
6 • Lettere-6	000♦0	———	0♦999	0099♦	9♦♦♦♦	♦9♦♦♦
7 • Lingue-1	152♦5	32411	53♦♦♦	♦♦♦♦5	43422	———
8 • Lingue-2 '	77997	98800	607♦6	———	♦6♦♦♦	♦♦08♦
9 • Storia e Filosofia-1	5♦33♦	♦3343	———	55♦44	♦♦♦4♦	4555♦
10 • Storia e Filosofia-2	9♦♦8♦	———	♦88♦0	990♦♦	0♦009	908♦8
11 • Matematica e Fisica-1	———	5♦♦♦♦	45434	4453♦	5♦55♦	♦4333
12 • Matematica e Fisica-2	♦9♦09	09998	♦♦♦08	88800	♦9♦♦0	———
13 • Matematica-1	SSSS2	2S1AA	112♦♦	♦♦♦1♦	———	22♦1♦
14 • Matematica-2	6S66S	SS♦AA	♦7S6♦	———	7777♦	♦♦♦6♦
15 • Scienze	33444	80022	———	7378♦	♦8995	5♦9♦♦
16 • Educazione Artistica	84518	———	96643	37279	2♦♦♦♦	01♦05
17 • Laboratorio Fisica	2277S	S♦♦67	———	1166♦	♦♦2DD	SS1♦♦
18 • Ginnastica-1	SSS♦♦	♦♦♦34	345SS	♦♦♦SS	S5♦♦♦	———
19 • Ginnastica-2	SSS♦♦	♦♦♦89	890SS	♦♦♦SS	S0♦♦♦	———
20 • Religione	4S853	♦♦♦♦♦	721S♦	SS♦♦♦	———	SS690

Fig.4.1 - Esempio di matrice che rappresenta un orario

4.3.1 Gli operatori genetici

Abbiamo già segnalato la necessità di definire per il TTP una serie di operatori genetici "non standard", al fine di tener conto delle caratteristiche del problema e della presenza dei vincoli. Sono stati quindi definiti gli operatori seguenti.

Riproduzione: è definita in modo standard, favorendo gli individui che hanno un valore della fitness function migliore della media; essa opera assegnando a ogni individuo h una probabilità di riproduzione $p_r(h)$ pari alla sua fitness function divisa per la fitness totale della popolazione.

Mutazione di ordine k: sia w = ... β ... δ ... una riga della matrice. Questo operatore prende due fattori di w tali che $|\beta| = |\delta| = k$ e li scambia tra di loro; la sua applicazione è soggetta ad alcune condizioni: non può operare se in uno dei due fattori scelti vi sono alcuni particolari caratteri, per esempio A o S (ciò perché ore di tipo A o S sono già state allocate, all'atto dell'inizializzazione, per attività che risultano bloccate).

Mutazione di giorno: sia $w = \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6$ una riga della matrice dove $|\alpha_i|=5$ (i=1, ... 6); questo operatore prende due fattori α_i e α_j di w e li scambia tra di loro; è un caso particolare di mutazione di ordine 5 ed è stato introdotto per motivi di efficienza computazionale legati all'assegnazione del giorno libero.

Crossover: come vedremo in seguito, la f.f. può essere ottenuta attraverso il calcolo di varie componenti; una di queste è la *fitness function locale* (f.f.l.), una funzione che ad ogni riga assegna un numero reale. Date due matrici $\mathbf{R}_1$ ed $\mathbf{R}_2$ di f.f. rispettivamente $f_1 < f_2$, sia $k = \lfloor (f_1 \cdot m)/(f_1+f_2) \rfloor$. L'operatore di crossover applicato a $\mathbf{R}_1$ ed $\mathbf{R}_2$ produce le matrici **A** e **B** ottenute come segue: **A** è ottenuta sostituendo le m-k righe "peggiori" (rispetto alla f.f.l.) di $\mathbf{R}_1$ con le analoghe righe di $\mathbf{R}_2$. **B** è ottenuta sostituendo le k righe "migliori" (rispetto alla f.f.l.) di $\mathbf{R}_1$ con le analoghe righe di $\mathbf{R}_2$.

Filtro: è un operatore che associa ad ogni "soluzione non ammissibile" una "soluzione ammissibile"; viene descritto dettagliatamente nella sezione 4.6.

Nella tabella 4.1 viene presentato un quadro completo degli operatori utilizzati, con le loro principali caratteristiche.

Tab.4.1 - Gli operatori genetici per il TTP

Operatore	Ammissibilità	Ammissibilità di riga [2]
Riproduzione	mantenuta	mantenuta
Mutazione di ordine k	non mantenuta	mantenuta
Mutazione di giorno	non mantenuta	mantenuta
Crossover	non mantenuta	mantenuta
Filtro	recuperata	mantenuta

Il mantenimento dei vincoli da parte dell'algoritmo opportunamente inizializzato avviene:

i attraverso gli operatori genetici: l'insieme di ore che un insegnante deve fare è mantenuto nell'applicazione di operatori specificamente ridefiniti in questo senso;

ii attraverso l'operatore di filtro: eventuali ore scoperte o sovrapposizioni causate dalla applicazione di altri operatori sono (parzialmente o totalmente) eliminate;

iii attraverso la funzione obiettivo: il miglioramento della popolazione può essere ottenuto anche mediante una riduzione del numero di ore scoperte o sovrapposizioni (considerate nella funzione obiettivo mediante penalità).

Le scelte relative ad ogni singolo docente sono vincolate l'una all'altra attraverso i vincoli "di colonna" (sovrapposizioni e ore scoperte), gestiti tramite una combinazione di penalità nella funzione obiettivo e di *genetic repair*.

La gestione delle sovrapposizioni e delle ore scoperte sia attraverso un operatore di filtro che attraverso penalità nella funzione obiettivo permette una maggiore libertà di movimento nello spazio di ricerca. Questa scelta è stata imposta dalla complessità del problema: ogni insegnante, infatti, propone un problema del tipo TSP, consistente nell'analizzare le permutazioni di un insieme predefinito di simboli.

[2] Cioè orario in cui ad ogni insegnante (riga) è stato assegnato un numero corretto di ore nelle classi di sua competenza.

4.4 La funzione obiettivo e la fitness function

La necessità di distinguere fra funzione obiettivo (f.o.) e fitness function (f.f.) deriva dal fatto che la f.o. viene formulata con riferimento ad un problema di costi da minimizzare mentre gli algoritmi genetici sono strutturati in modo da risolvere problemi di massimizzazione: non è sufficiente cambiare il segno della f.o. per ottenere la f.f., perché in tal modo i valori da massimizzare divengono negativi mentre l'algoritmo genetico richiede valori positivi. Abbiamo pertanto deciso di definire la fitness function componendo la f.o. con una opportuna funzione monotona decrescente a valori positivi; nel nostro caso abbiamo usato un'esponenziale negativa, del tipo $e^{-\rho x}$, con parametro ρ.

La f.o. per il nostro problema è costruita in modo da misurare un costo (generalizzato) che rappresenta lo scostamento dell'orario in esame da una condizione "ideale". Detta f la funzione obiettivo, f risulta:

$$f(\mathbf{R}_1) = \alpha \cdot \#_{in}(\mathbf{R}_1) + \beta_1 \cdot gr_d(\mathbf{R}_1) + \beta_2 \cdot gr_o(\mathbf{R}_1) + \beta_3 \cdot gr_p(\mathbf{R}_1)$$

dove

$\#_{in}(\mathbf{R}_1)$ conta il numero di inammissibilità nella matrice $\mathbf{R}_1$,
$gr_d(\mathbf{R}_1)$ misura il grado di non soddisfacimento delle esigenze didattiche nella matrice $\mathbf{R}_1$,
$gr_o(\mathbf{R}_1)$ misura il grado di non soddisfacimento delle esigenze organizzative nella matrice $\mathbf{R}_1$,
$gr_p(\mathbf{R}_1)$ misura il grado di non soddisfacimento delle esigenze personali nella matrice $\mathbf{R}_1$.

α, β_1,β_2,β_3, sono dei pesi che permettono di dare maggiore importanza ad alcuni aspetti rispetto ad altri.

Ad esempio la scelta $\alpha >> \beta_1, \beta_2, \beta_3$ induce una *struttura gerarchica* in cui le inammissibilità sono molto più importanti delle altre componenti. In generale, la struttura gerarchica tiene conto della diversa importanza dei vari gruppi di condizioni del problema, cui vengono attribuiti pesi che hanno ordini di grandezza diversi.

Nei casi da noi affrontati la struttura scelta è stata la seguente:

livello 1, condizioni di ammissibilità, con pesi dell'ordine delle migliaia;
livello 2, condizioni sulla gestione globale, con pesi dell'ordine delle centinaia;
livello 3, condizioni sui singoli docenti, con pesi dell'ordine delle decine.

Al *livello 1* vengono prese in considerazione eventuali sovrapposizioni di docenti (nella stessa ora sulla stessa classe) ed eventuali ore scoperte nelle classi (ore in cui una classe non è abbinata a nessun docente).

Al *livello 2* vengono prese in considerazione tre tipi di esigenze diverse, rispettivamente indicate come

Δ - esigenze didattiche:

- non più di 4 ore al giorno di lezioni effettive per docente ($\Delta 1$),
- non tutti i giorni l'ultima ora ad uno stesso docente ($\Delta 2$),
- distribuzione uniforme delle ore di lezione di un docente in una classe ($\Delta 3$),
- coppie di ore richieste per i compiti in classe ($\Delta 4$);

Il grado di non soddisfacimento $gr_d(\mathbf{R}_1)$ delle esigenze didattiche dell'orario $\mathbf{R}_1$ è quindi una funzione delle grandezze $\Delta 1, ..., \Delta 4$. La forma della funzione (di solito una somma pesata) e il modo di definire i valori $\Delta 1, ..., \Delta 4$ vengono scelti dall'utente.

Ω - esigenze organizzative:

- distribuzione il più possibile uniforme delle ore a disposizione dei docenti lungo la settimana ($\Omega 1$),
- non una sola ora al giorno di lezione per docente ($\Omega 2$),
- "ricevimento-parenti" per gli insegnanti di una stessa classe il più possibile negli stessi giorni ($\Omega 3$);

Il grado di non soddisfacimento $gr_o(\mathbf{R}_1)$ delle esigenze organizzative dell'orario $\mathbf{R}_1$ è calcolato in modo analogo a quanto visto per $gr_d(\mathbf{R}_1)$.

Π - esigenze personali:

- viene deciso che peso dare ai singoli docenti, tenendo conto di eventuali graduatorie tra docenti (anzianità, impegni, ecc.).

Il grado di non soddisfacimento $gr_p(\mathbf{R}_1)$ delle esigenze personali dei docenti dell'orario $\mathbf{R}_1$ viene calcolato dando ad ogni docente un peso. Questo peso viene moltiplicato per il grado di non soddisfacimento delle singole esigenze come definite (in modo interattivo, vedi Fig.4.2) da ogni utente nel successivo livello 3.

Preferenze personali dei docenti	**Peso G.L.**	**Peso P.O.**	**Peso U.O.**
1 • Lettere-1	0.20	0.00	0.80
2 • Lettere-2	0.80	0.10	0.10
3 • Lettere-3	1.00	0.00	0.00
4 • Lettere-4	1.00	0.00	0.00
5 • Lettere-5	0.80	0.00	0.20
6 • Lettere-6	0.80	0.00	0.20
7 • Lingue-1	0.75	0.25	0.00
8 • Lingue-2	1.00	0.00	0.00
9 • Storia e Filosofia-1	0.50	0.00	0.50
10 • Storia e Filosofia-2	0.25	0.25	0.50
11 • Matematica e Fisica-1	0.80	0.00	0.20
12 • Matematica e Fisica-2	0.80	0.20	0.00
13 • Matematica-1	0.80	0.00	0.20
14 • Matematica-2	0.60	0.00	0.40
15 • Scienze	0.40	0.30	0.30
16 • Educazione Artistica	0.70	0.00	0.30
17 • Laboratorio Fisica	0.80	0.00	0.20
18 • Ginnastica-1	1.00	0.00	0.00
19 • Ginnastica-2	1.00	0.00	0.00
20 • Religione	1.00	0.00	0.00

F10 ⇔

Lettere-1

GIORNO LIBERO	L M M G V S
SI	0 0 1 0 0 0
FORSE	0 0 0 1 0 0
NO	1 1 0 0 1 1

PRIME ORE	L M M G V S
SI	
FORSE	
NO	

ULTIME ORE	L M M G V S
SI	0 0 0 0 0 0
FORSE	0 0 0 0 0 0
NO	1 1 1 1 1 1

F5=Help ESC=Uscita

Fig.4.2 - Una maschera di input

Il *livello 3* è stato introdotto per il calcolo delle esigenze personali: ad ogni docente è data la possibilità di suddividere una certa quantità di "punti" tra diverse opzioni, dando una maggiore quantità di punti alle opzioni più desiderate. In questo modo il programma può tenere conto delle preferenze espresse da ogni singolo docente.

4.5 Complessità dell'algoritmo

L'algoritmo, in notazione Pascal-like, è il seguente:

```
Inizializzazione        {questa routine crea una popolazione iniziale di N individui, ognuno dei
                        quali soddisfa un insieme di vincoli:
                        - ad ogni insegnante (righe) è assegnato il numero corretto di ore che
                          deve insegnare
                        - alcune delle ore hanno un valore "fissato", il che significa che non
                          possono essere modificate}
while (not_verified_end_test) do
                        {l'End test al momento è semplicemente un test sul numero di iterazioni
                        effettuate}
begin
    applica l'operatore riproduzione;
    applica con probabilità pc l'operatore crossover;
    applica con probabilità pm l'operatore mutazione;
    applica l'operatore mutazione di giorno;
    for l:=1 to N do
        if (numero_di_inamissibilità> MAX_INAMISSIBILITÀ)
                        {MAX_INAMISSIBILITÀ è una costante di sistema}
        then applica l'operatore filtro;
end.
```

Uno schema a blocchi dell'algoritmo complessivo è riportato in figura 4.3.

Nel seguito di questa sezione diamo alcune valutazioni di complessità computazionale in funzione del numero N di individui che compongono la popolazione e delle probabilità di attivazione scelte per ogni operatore. Chiamiamo **FF** la complessità del calcolo della funzione di fitness, **LFF** la complessità del calcolo della funzione di fitness locale e **FI** la complessità del calcolo del filtro.

Abbiamo scelto di rappresentare esplicitamente le probabilità di attivazione nelle formule relative alla complessità degli operatori perché tali valori sono parametri definiti dall'utente che può quindi, per mezzo di tali formule valutare l'impatto che il valore di questi parametri può avere sull'efficienza di calcolo, cioè sul tempo richiesto per fare un ciclo.

Inizializzazione
Stop
si
End test
no
Riproduzione
Crossover
Mutazione
Non ammissibilita'
no
si
Filtro

Fig.4.3 - Schema a blocchi dell'algoritmo

<u>Riproduzione</u>

Il seguente algoritmo implementa l'operatore di riproduzione:

Passo 1. for l:=1 to N do
    calcola fitness(l)
Passo 2. calcola sumFit {calcola la somma delle fitness di tutti gli individui}
Passo 3. for l:=1 to N do
    calcola $p_r(l) = \frac{\text{fitness}(l)}{\text{sumFit}}$ {calcola le probabilità di riproduzione}
Passo 4. for l:=1 to N do
    attribuisci ad ogni individuo un valore reale proporzionale a $p_r(l)$ nell'intervallo reale [0,1]
    {prepara una lista di individui cui applicare il metodo Montecarlo}
    chiama reproduction_list questa lista
Passo 5. for l:=1 to N do
    scegli a caso un valore nell'intervallo [0,1] e seleziona il corrispondente individuo in reproduction_list affinché sia riprodotto nella nuova popolazione generata
    {applica il metodo Montecarlo}

Analisi della complessità:

Passo 1: O(FF·N) {FF è la complessità di calcolo della fitness function}
Passo 2: O(N)
Passo 3: O(N)
Passo 4: O(N)
Passo 5: O(N)

L'operatore di riproduzione sull'intera popolazione è calcolabile in O(FF·N) passi di calcolo.

Mutazione di ordine k

Definiamo per prima cosa la funzione *mutate(row_i, k)* che prende in input la riga cui applicare la mutazione e il numero k di elementi da mutare. Questa funzione genera le posizioni iniziali pos_1 e pos_2 delle due stringhe di lunghezza k che devono essere scambiate ed opera lo scambio. La funzione ritorna la nuova riga mutata.

```
function mutate(row_i, k): row;
begin
    Passo 1:  scegli in modo casuale un intero k ∈ [1,n/2] e due posizioni, pos_1 e
              pos_2, nella riga row_i tali che pos_1< pos_2≤ n-k;
    Passo 2:  scambia i valori nell'intervallo [pos_1 , (pos_1+k-1)] con i valori
              nell'intervallo [pos_2 , (pos_2+k-1)]
end;
```

Il seguente algoritmo implementa l'operatore *mutazione di ordine k* applicato all'intera popolazione:

```
for l:=1 to N do
   begin
     for i:=1 to m do              {con m = numero di righe della matrice}
        if (p_m≤ (random 1))       {(random 1) ritorna un valore ∈ [0,1], e p_m è la probabilità
                                    di mutazione}
        then applica mutate(row_i, k)
   end;
```

Analisi della complessità:

function mutate(row_i, k)
Passo 1: O(1)
Passo 2: O(k)=O(n/2) con n numero di colonne della matrice

L'aspettazione del numero di passi di calcolo causati da una applicazione dell'operatore di mutazione sull'intera popolazione è quindi $O[(N \cdot p_m) \cdot (n \cdot m)]$, dove p_m è la probabilità di applicazione dell'operatore di mutazione.

Crossover

L'operatore di crossover è implementato dal seguente algoritmo:

for l:=1 to ($N \cdot p_c$) do {è applicato solo ad una parte della popolazione data dal valore di p_c}
begin
Passo 1: scegli a caso due individui
Passo 2: calcola la loro l.f.f.
Passo 3: ordina per valori decrescenti della l.f.f. le righe dei due individui
Passo 4: crea due figli incrociando i due individui (il primo figlio è generato prendendo le k_1 migliori righe dal migliore dei due genitori e le righe ancora mancanti dall'altro; il secondo figlio è generato usando le righe rimaste inutilizzate da entrambi i genitori)
end

Analisi della complessità:

Passo 1: O(1)
Passo 2: O(2m·**LFF**) {**LFF** è la complessità dell'algoritmo di calcolo della l.f.f.}
Passo 3: O(2m) {algoritmo di ordinamento}
Passo 4: O(m)

L'aspettazione del numero di passi di calcolo causati da una applicazione dell'operatore di crossover è pertanto O[($N \cdot p_c$)·(m·**LFF**)].

Filtro

L'algoritmo che implementa l'operatore filtro è strutturato in quattro passi principali, ognuno dei quali è esso stesso un filtro, di potenza via via maggiore (ma anche più lento). Come vedremo dopo avere presentato l'algoritmo, la complessità dei primi due passi è $O(m^3 \cdot n^2)$, mentre quella degli ultimi due è $O(m^6 \cdot n^3)$. All'utente è lasciata la possibilità di decidere se utilizzare appieno il filtro o limitarne l'applicazione ai primi due passi. Nel primo caso la percentuale di orari senza inammissibilità ottenuta dopo una applicazione del filtro è più elevata, a fronte di un più elevato costo computazionale.

L'algoritmo filtro è il seguente:

Passo 0: per ogni colonna h (h=1, ..., n) della matrice:
- calcola l'insieme delle classi che hanno delle sovrapposizioni e metti queste classi in una lista chiamata **over(h)**, memorizzando con ogni classe i relativi indici della matrice;
- calcola l'insieme delle classi scoperte (cioè senza professore) e metti queste classi in una lista chiamata **miss(h)**, memorizzando con ogni classe i relativi indici della matrice;

Passo 1: fintantoché esistono coppie di classi c_i e c_j tali che:
- c_i e c_j appaiono nella stessa riga (cioè, le classi c_i e c_j hanno un insegnante in comune)
- in una delle righe nelle quali le due classi sono entrambe presenti, esse occupano le colonne h e k, con $c_i \in$ **over(h)**, $c_i \in$ **miss(k)**, $c_j \in$ **over(k)**, $c_j \in$ **miss(h)**

scambia c_i con c_j.

L'effetto di una applicazione del passo 1 è mostrato nella figura 4.4.

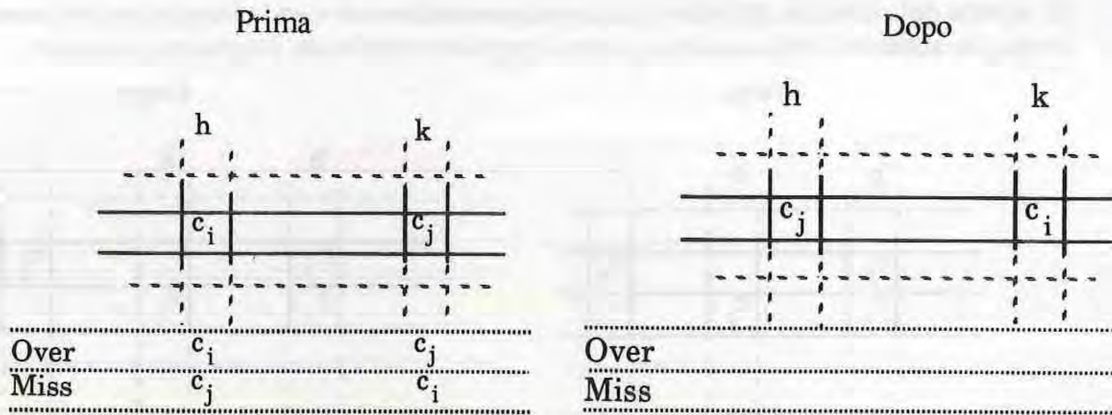


Fig.4.4 Scambio di due classi

Passo 2: fintantoché esistono coppie di elementi, costituiti da una classe c_i e un *gene mobile* e_j (cioè un gene che corrisponde o al carattere D o al carattere •), tale che:

- c_i e e_j appaiono nella stessa riga
- in una delle righe nelle quali i due elementi sono entrambi presenti, essi occupano le colonne h e k, con $c_i \in$ **over(h)**, $c_i \in$ **miss(k)**

scambia c_i con e_j.

L'effetto di una applicazione del passo 2 è mostrato nella figura 4.5.

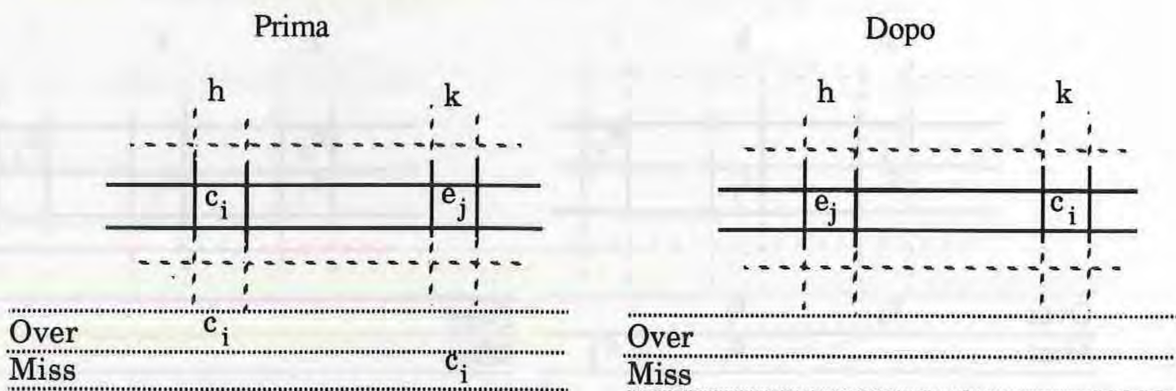


Fig.4.5 Scambio di due elementi

Passo 3: fintantoché esistono *cammini transitivi tra classi*, cioè cammini che connettono le classi $c_i, c_j, \ldots, c_t, c_z$, tali che

- $c_i \in$ **over(h)**, $c_i \in$ **miss(k)**, $c_j \in$ **over(k)**, .. , $c_t \in$ **miss(l)**, $c_z \in$ **over(l)**, $c_z \in$ **miss(h)**
- le classi $c_i, c_j, \ldots, c_t, c_z$ sono a coppie nella stessa riga

scambia ogni classe con la seguente nel cammino transitivo appartenente alla stessa riga.

L'effetto di una applicazione del passo 3 è mostrato nella figura 4.6, nel caso in cui la lunghezza del cammino transitivo è limitata a due. Dato che la complessità dell'algoritmo

di ricerca dei cammini transitivi cresce esponenzialmente con la lunghezza del cammino stesso, lo abbiamo implementato ponendo uguale a due la sua lunghezza massima.

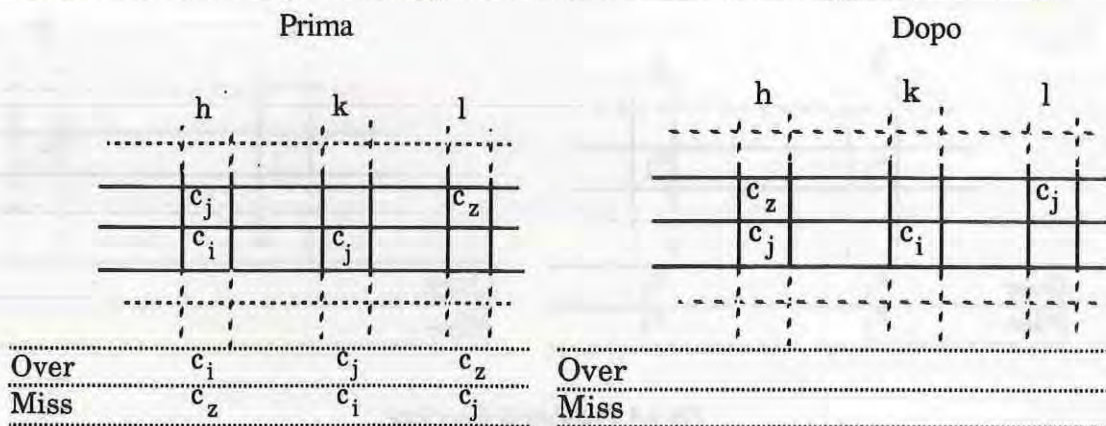


Fig.4.6 Scambio di classi in un cammino transitivo

Passo 4: fintantoché esistono *cammini transitivi tra elementi*, dove un elemento può essere una classe o un elemento mobile nel senso definito al passo 2, comportati come al passo 3 (permettendo la sostituzione di una classe con un elemento mobile).

L'effetto di una applicazione del passo 4 è mostrato nella figura 4.7.

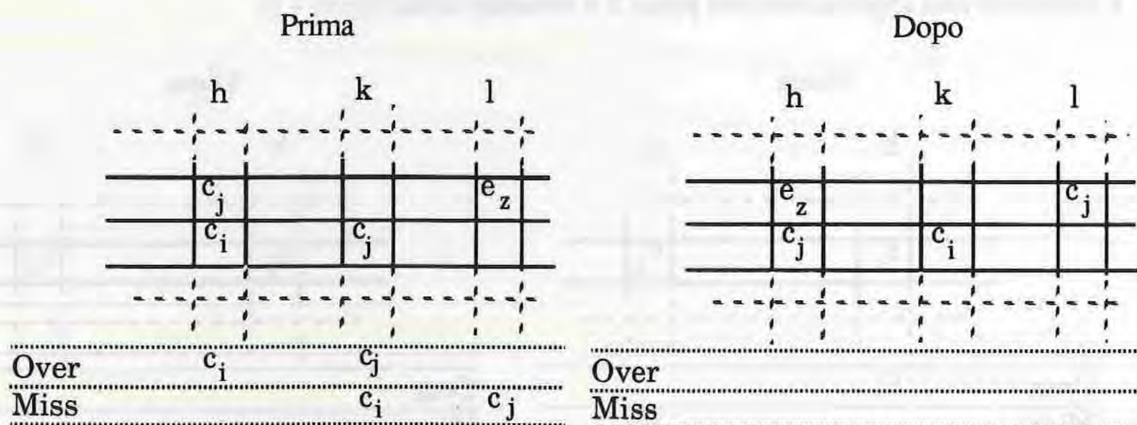


Fig.4.7 Scambio di elementi in un cammino transitivo

Riassumendo, l'aspettazione del numero di passi di calcolo causati da una applicazione degli operatori, quando applicati ad una popolazione di dimensioni N, è la seguente:

Riproduzione	$O(\mathbf{FF} \cdot N)$
Mutazione di ordine k	$O(N \cdot n \cdot m \cdot p_m)$
Crossover	$O(N \cdot m \cdot \mathbf{LFF} \cdot p_c)$
Filtro	$O(N \cdot m^6 \cdot n^3)$

La complessità del calcolo di **FF** e di **LFF** è

Fitness function	$\mathbf{FF} = O(m^2 \cdot n^2)$
Fitness function locale	$\mathbf{LFF} = O(m \cdot n^2)$

4.6 Risultati

Il programma, realizzato in linguaggio C, funziona su PC-Ibm compatibile con configurazione standard.

Il modello e il programma sono stati testati nella costruzione dell'orario di un grosso liceo scientifico di Milano. Ciò è stato fatto in collaborazione con gli insegnanti che normalmente si occupano della preparazione degli orari: la collaborazione è stata fondamentale, sia nella fase di progettazione (per la definizione delle specifiche) che in quella di convalida, che è tuttora in corso con risultati che appaiono decisamente soddisfacenti.

La scelta di volere un programma effettivamente utilizzabile ci ha impedito un confronto diretto con approcci alternativi al TTP: abbiamo infatti prestato un'attenzione molto maggiore ai possibili desideri/richieste degli insegnanti di quella che si può trovare nei lavori presenti in letteratura. Inoltre le differenze nell'organizzazione degli orari scolastici tra le varie nazioni fa si che ogni tentativo di soluzione del problema fatto da ricercatori di diversa nazionalità sia difficilmente confrontabile con gli altri.

Gli esperimenti effettuati si sono pertanto orientati alla valutazione dei seguenti aspetti:

(i) individuazione di un set di parametri per cui il comportamento dell'algoritmo sia buono;

(ii) verifica dell'effettiva convergenza dell'algoritmo verso valori bassi della funzione obiettivo;

(iii) confronto degli orari ottenuti con quelli generati da esperti umani.

Gli esperimenti effettuati sui parametri sono i seguenti:

(1) abilitazione/disabilitazione delle fasi 3 e 4 della procedura di filtro;

(2) costo delle inammissibilità: il valore è stato posto a 1000 (un ordine di grandezza maggiore del peso del livello 2) o a 500 (per facilitare la ricerca passando per zone di non ammissibilità)

(3) uso della strategia elitista (cioè un numero - compreso fra 1 e 5 - di copie del miglior individuo vengono ricopiate nella popolazione dei figli con probabilità uno)

(4) variazione del valore delle probabilità di mutazione e di crossover (p_{m1}, p_{mk}, p_c) e della probabilità di accettazione di un figlio peggiore del padre.

L'esperimento (1), i cui risultati sono ottenuti su tre prove, ha mostrato che la differenza tra le due modalità è molto lieve, sia per quanto riguarda il valore finale trovato (costo medio 205 con il filtro completamente attivo e 213 con il filtro parzialmente disattivato) che il tempo necessario per trovarlo (in media 592 iterazioni nel primo caso e 629 nel secondo). Il rallentamento dovuto all'utilizzo della versione più completa del filtro è risultato essere inferiore al 10%. Pertanto le due alternative possono essere considerate circa equivalenti.

I punti (2) e (3) sono stati testati in modo simile, eseguendo 10 esperimenti per ogni coppia di valori. I risultati sono riportati in tabella 4.2. In questa tabella le coppie di valori per il costo delle inammissibilità (500-1000) e per la strategia elitista[3] (1-5) sono testate in

---

[3] Il parametro relativo alla strategia elitista è $\#C_p$

10 esperimenti in ognuno dei quali gli altri parametri sono mantenuti costanti. Si può vedere come il valore 500 per il costo delle inammissibilità generi risultati leggermente migliori: la fitness è migliore di circa il 5% in 7 casi su 10. La tabella 4.2 mostra anche che la scelta del valore 5 per il parametro $\#C_p$ (che indica l'uso di 5 copie dell'individuo migliore) risulta essere più favorevole.

Tab.4.2 - Test sul costo delle inammissibilità e della strategia elitista

Test	Costo	#Cp	p_{m1}	p_{mk}	p_c	p_a	f.o.	generazione*
1	1000	1	.1	.1	.8	.9	196	852
	500	1	.1	.1	.8	.9	207	1177
2	1000	1	.3	.1	.5	.9	228	572
	500	1	.3	.1	.5	.9	208	817
3	1000	1	.3	.3	.5	.9	216	745
	500	1	.3	.3	.5	.9	211	798
4	1000	1	.8	.5	.3	.5	259	365
	500	1	.8	.5	.3	.5	249	629
5	1000	1	.8	.5	.3	.9	288	962
	500	1	.8	.5	.3	.9	263	884
6	1000	5	.1	.1	.8	.9	165	1515
	500	5	.1	.1	.8	.9	171	435
7	1000	5	.3	.1	.5	.9	222	598
	500	5	.3	.1	.5	.9	207	823
8	1000	5	.3	.3	.5	.9	198	1343
	500	5	.3	.3	.5	.9	192	665
9	1000	5	.5	.3	.5	.9	217	1426
	500	5	.5	.3	.5	.9	219	729
10	1000	5	.8	.5	.8	.9	229	864
	500	5	.8	.5	.8	.9	180	3633

(*) generazione alla quale ha avuto luogo l'ultima iterazione

Gli esperimenti relativi al punto (4) hanno mostrato che i valori migliori per i parametri sono: p_c=0.8, p_{m1}=0.1÷0.3, p_a=0.5.

Nelle fig. 4.8 e 4.9 presentiamo due evoluzioni del sistema, fermate dopo 5000 generazioni. La prima riguarda una prova con parametri di controllo: p_{m1}=0.5, p_c=0, p_a=0.5, penalità per sovrapposizioni e classi scoperte (inammissibilità)=1000. La seconda riguarda una prova con parametri di controllo: p_{m1}=0.5, p_c=0.2, p_a=1, penalità per sovrapposizioni e classi scoperte=500. In entrambi i casi è evidente come l'algoritmo converga verso un valore basso (ma maggiore di zero) della f.o. Il fatto che l'asintoto non abbia costo nullo deriva dalla parziale conflittualità degli obiettivi e dei desideri degli insegnanti, cioè dall'impossibilità (ben nota) di raggiungere la condizione "ideale" in un problema di questo tipo.

Si noti che il primo caso, con parametri molto più restrittivi per la ricerca (penalizza molto le inammissibilità, non permette crossover, accetta "con sospetto" le variazioni peggiorative) porta ad una convergenza precoce.

Dall'esperienza che si sta facendo emerge che il sistema proposto può essere uno strumento assai valido per sostituire l'uomo in questo compito difficile e noioso. Si può inoltre pensare che, una volta ottenuto un orario soddisfacente (che normalmente corrisponde a un minimo locale), sia poi facile per l'utente valutare la rispondenza

effettiva della soluzione proposta ed eventualmente effettuare piccole modifiche per tener conto di situazioni eccezionali, non previste nel modello.

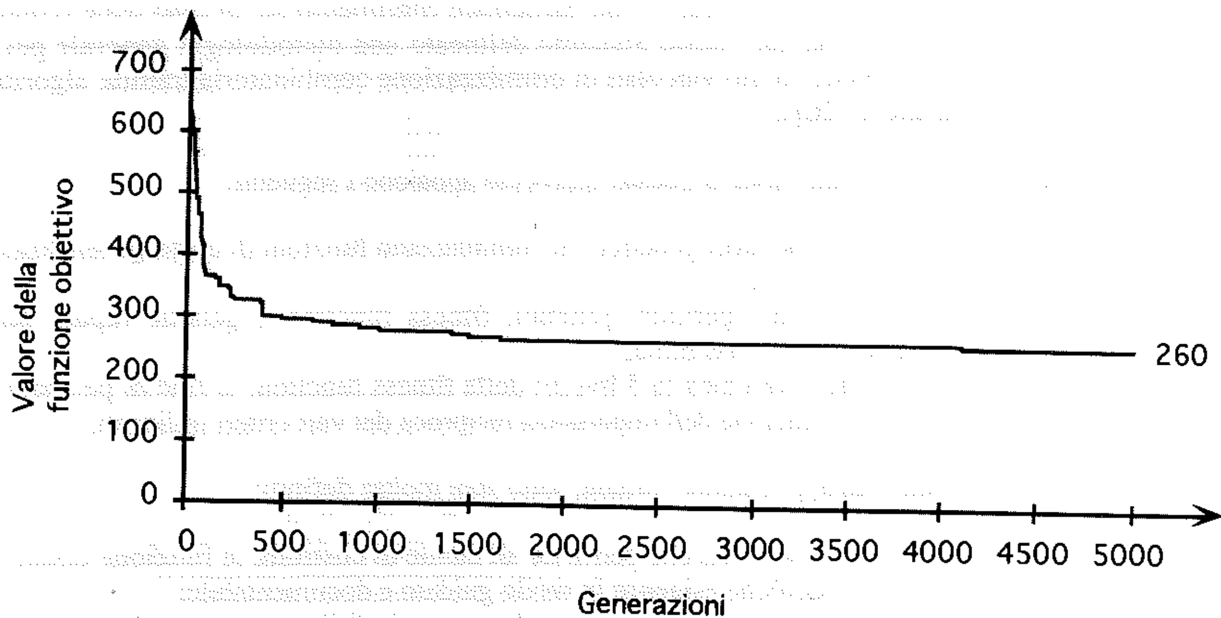


Fig. 4.8 - Creazione di un orario fermata dopo 5000 generazioni

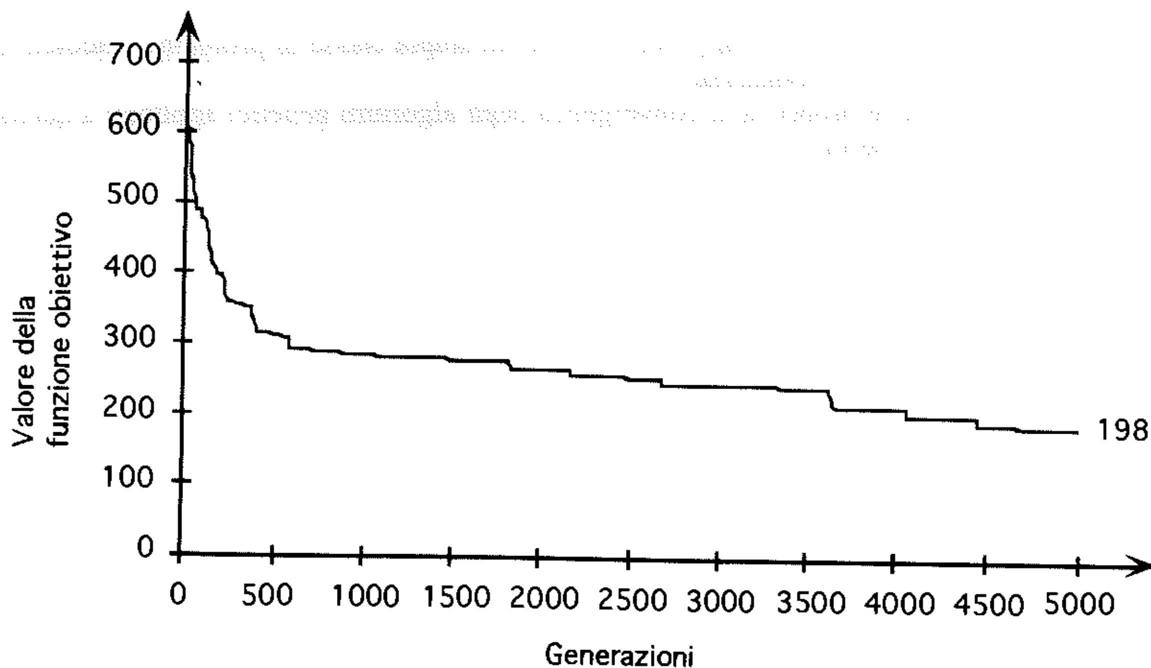


Fig. 4.9 - Creazione di un orario fermata dopo 5000 generazioni con parametri che permettono la continuazione della ricerca

4.7 Conclusioni e sviluppi futuri

In questo capitolo abbiamo descritto l'applicazione degli algoritmi genetici alla risoluzione del problema dell'orario, con particolare riferimento ad un caso reale (l'orario di un grosso liceo). Per far questo abbiamo delineato una metodologia generale per la risoluzione di problemi molto vincolati di ottimizzazione combinatoria tramite algoritmi genetici (vedi anche [20]).

Gli elementi generalizzabili di questo approccio appaiono i seguenti:

- la definizione di operatori genetici che minimizzano funzioni di costo generalizzato (con eventuali penalità);
- la collaborazione di operatori genetici, fitness function e genetic repair nella gestione delle non ammissibilità;
- la strutturazione gerarchica (a 3 livelli) della fitness function, al fine di permettere una facile individuazione dell'importanza reciproca dei vari criteri utilizzati;

Con riferimento all'applicazione trattata, sono stati inoltre definiti:

- un'interfaccia *menu-based* che permette all'utente di costruire la funzione obiettivo adatta alle sue specifiche esigenze in modo guidato e documentabile;
- un algoritmo di filtro in grado di rendere ammissibile un orario che non lo è, generandone uno "vicino" (la vicinanza essendo definita secondo una metrica apposita).

Possibili sviluppi futuri saranno:

- la generalizzazione dell'approccio a una più ampia classe di problemi vincolati di ottimizzazione combinatoria;
- lo studio delle proprietà di convergenza degli algoritmi genetici applicati a queste classi di problemi.

5

I sistemi a classificatori e il problema del robot autonomo

5.1 Introduzione
5.2 Quale architettura software?
5.3 Modifiche apportate al sistema a classificatori standard
5.4 ALECSYS: un sistema a classificatori parallelo
5.5 Valutazione sperimentale di ALECSYS
5.6 Lavori correlati
5.7 Conclusioni e sviluppi futuri

5.1 Introduzione

Questo capitolo è dedicato all'esposizione dei risultati ottenuti applicando i sistemi a classificatori (SC) al problema del controllo di un robot autonomo. Il modello SC è stato introdotto nella sezione 2.4 e pertanto verrà qui solo brevemente richiamato, nei limiti di quanto necessario per esporre le modifiche da noi apportate per renderlo più efficiente ed efficace per il compito in oggetto. Tali modifiche hanno portato allo sviluppo di un nuovo algoritmo per la soluzione del problema della valutazione delle regole che nelle nostre sperimentazioni è risultato più efficace di quelli proposti in letteratura. L'algoritmo viene presentato, insieme ai risultati di alcune simulazioni, nella sezione 5.2. La necessità di avere tempi di reazione molto rapidi ci ha portato alla progettazione di un sistema a classificatori parallelo - ALECSYS - su architettura MIMD (implementato su una rete di transputer). La struttura del sistema parallelo è esposta nella sezione 5.4, mentre nella sezione 5.5 vengono esposti in dettaglio i risultati di alcuni esperimenti eseguiti utilizzando un robot simulato. Obiettivo di questi esperimenti era la messa a punto e la valutazione del sistema software (in particolare l'organizzazione del parallelismo e le caratteristiche degli algoritmi) in vista di una sua applicazione al robot vero. La sezione 5.6 è dedicata ad una breve esposizione dei lavori che maggiormente hanno influenzato il lavoro presentato in questo capitolo. Infine la sezione 5.7 riporta alcuni commenti riguardo i risultati ottenuti e i possibili sviluppi futuri, con particolare riferimento alle possibilità di utilizzo del software presentato per controllare un robot vero [35], [8].

5.2 Quale architettura software?

Nel definire l'architettura software da utilizzare per implementare gli algoritmi di apprendimento automatico e di controllo del robot autonomo ci siamo posti come obiettivo di lungo termine quello di progettare un sistema che superasse alcune delle difficoltà finora incontrate dai sistemi tradizionali di intelligenza artificiale basati su una rappresentazione simbolica della conoscenza [36], [31], [37]. A tal fine abbiamo sviluppato il sistema ALECSYS [38] (A LEarning Classifier SYStem), un sistema di apprendimento automatico basato su sistemi a classificatori ed algoritmi genetici, che presenta i seguenti aspetti innovativi rispetto ai sistemi precedentemente proposti (vedi anche la sezione 5.6 nella quale sono presentati i lavori correlati):

(i) utilizzo di un'architettura di riferimento di derivazione naturale (cioè il modello di Tinbergen [80] di gerarchia degli istinti, vedi sezione 5.2.1);
(ii) utilizzo di un singolo sistema a classificatori per ogni comportamento di base (definito dal progettista) da apprendere;
(iii) utilizzo di un sistema a classificatori per ogni comportamento di coordinamento da apprendere (sono comportamenti di coordinamento quelli che si occupano di coordinare le interazioni tra comportamenti);
(iv) utilizzo di un sistema parallelo su due livelli (vedi sezione 5.4): il parallelismo di alto livello permette di far coincidere l'architettura logica dei punti (ii) e (iii) con l'architettura fisica di una rete di transputer; il parallelismo di basso livello permette di distribuire ogni singolo sistema a classificatori su più processori (nodi della rete transputer) in modo tale da ottenere tempi di risposta sufficientemente rapidi da permettere il funzionamento in tempo reale;
(v) introduzione di una serie di modifiche negli algoritmi di performance e di distribuzione del credito che migliorano l'efficienza del sistema.

5.2.1 Il modello di Tinbergen

Nel 1966 Tinbergen [80] propose un modello del comportamento animale sviluppato nell'ambito di ricerche in etologia.

Il modello può essere descritto come una gerarchia di moduli comportamentali, i così detti *centri di istinto*. Ogni centro di istinto è decomposto in centri di istinto di grana più fine posti ad un livello inferiore della gerarchia. I centri di istinto allo stesso livello della gerarchia competono per raggiungere lo stato di attivazione. Ad un dato livello della gerarchia solo i centri di istinto con un elevato grado di eccitazione possono mandare dei segnali eccitatori ai centri di istinto a loro direttamente collegati al livello inferiore. Lo stato di eccitazione globale di un centro di istinto è pertanto una funzione del livello dei segnali eccitatori provenienti dal livello superiore. Esso è inoltre funzione dei segnali che arrivano dai sensori - interni ed esterni (esempio di segnale interno può essere il dolore provocato da un organo malfunzionante) - e dallo stato motivazionale. Un centro di istinto diviene attivo solo se è superato un certo valore soglia che è funzione dello stato di eccitazione del centro di istinto stesso. In Fig.5.1 è riportata la gerarchia di istinti così come proposta da Tinbergen.

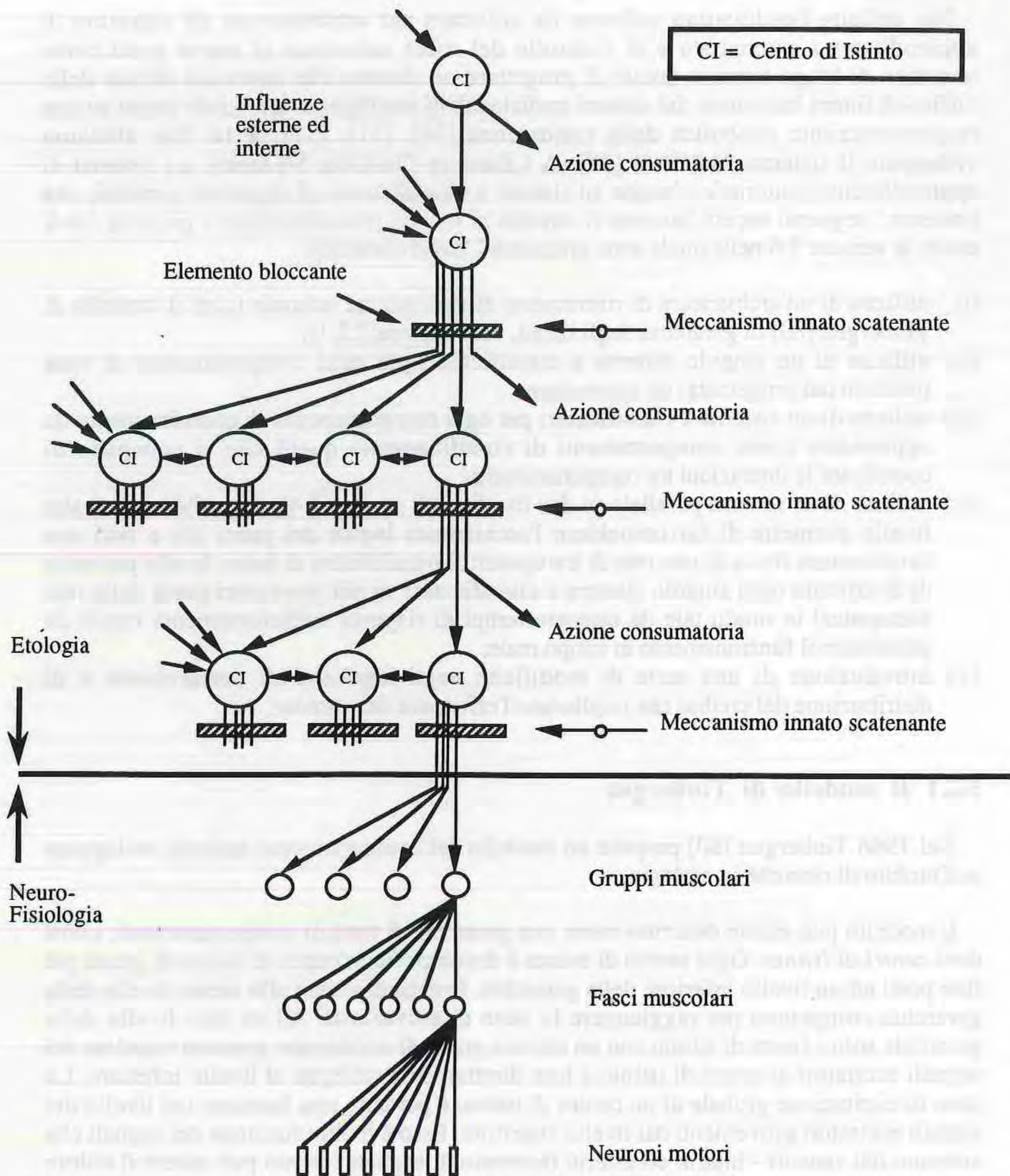


Fig.5.1 - Gerarchia degli istinti secondo Tinbergen

5.2.2 Il modello su cui si basa ALECSYS

Come detto, il modello presentato nella sottosezione precedente ci serve come punto di partenza nel definire l'architettura del nostro sistema (ALECSYS). Nel presentare il nostro modello faremo una distinzione tra un *modello completo* e un *modello corrente*: il primo rappresenta il nostro obiettivo di lungo periodo, mentre il secondo rispecchia lo stato attuale del sistema.

Il modello completo consiste di molti sistemi a classificatori (SC) che operano contemporaneamente. Ogni SC apprende un semplice comportamento per mezzo dell'interazione con l'ambiente, mentre il sistema nel suo complesso deve, come compito di apprendimento, imparare a coordinare i vari SC che lo compongono. La struttura gerarchica ci permette infatti di distinguere due tipi fondamentali di SC: uno che apprende come reagire a determinati input per mezzo di *sequenze comportamentali*, l'altro che impara a coordinare un gruppo di SC. Nel nostro modello i SC al livello più basso apprendono ad eseguire le sequenze comportamentali, mentre quelli di più alto livello imparano il coordinamento. Inoltre solo i SC di livello più basso hanno accesso diretto all'ambiente esterno attraverso i sensori e gli attuatori. La Fig.5.2 illustra l'organizzazione gerarchica dei sistemi a classificatori.

Il modello completo può essere caratterizzato nel modo seguente:

a) i SC operano in parallelo ad ogni livello della gerarchia;
b) ogni SC al livello più basso rappresenta una classe di possibili interazioni con l'ambiente;
c) ogni SC degli altri livelli rappresenta una possibile classe di interazioni tra SC al livello direttamente inferiore;
d) SC appartenenti ad uno stesso livello possono essere associati ad un SC comune al livello direttamente superiore se divengono attivi nella stessa situazione;
e) ogni SC riceve segnali eccitatori ed inibitori dai SC a lui collegati e calcola il suo stato di attivazione. Se questo è sufficientemente alto il SC diviene attivo e spedisce i messaggi appropriati ai SC associati;
f) SC allo stesso livello gerarchico competono per diventare attivi scambiandosi messaggi inibitori;
g) solo i SC al livello più basso hanno accesso diretto ai sensori ed agli attuatori;
h) i segnali eccitatori arrivano ad un SC da: sensori (solo per i SC di più basso livello), stimoli motivazionali, stimoli inibitori da altri SC dello stesso livello, stimoli eccitatori da SC del livello superiore;
i) i SC di livello più alto scambiano segnali eccitatori con i SC associati al livello immediatamente inferiore e viceversa;
j) ogni SC può essere associato con più di un SC del livello immediatamente superiore;
k) i segnali motori che arrivano agli attuatori da tutti i SC attivi sono sommati (secondo una somma pesata) e tradotti in una azione motoria.

Mentre il modello di Tinbergen è una descrizione di una data forma di organizzazione del comportamento, il nostro modello descrive un processo di costruzione di una struttura che dovrebbe essere capace di apprendere un insieme di comportamenti. Una caratteristica notevole del nostro modello è altresì la sua espandibilità: per mezzo dell'aggiunta di nuovi moduli SC è possibile, almeno in teoria, aumentare l'insieme dei compiti apprendibili.

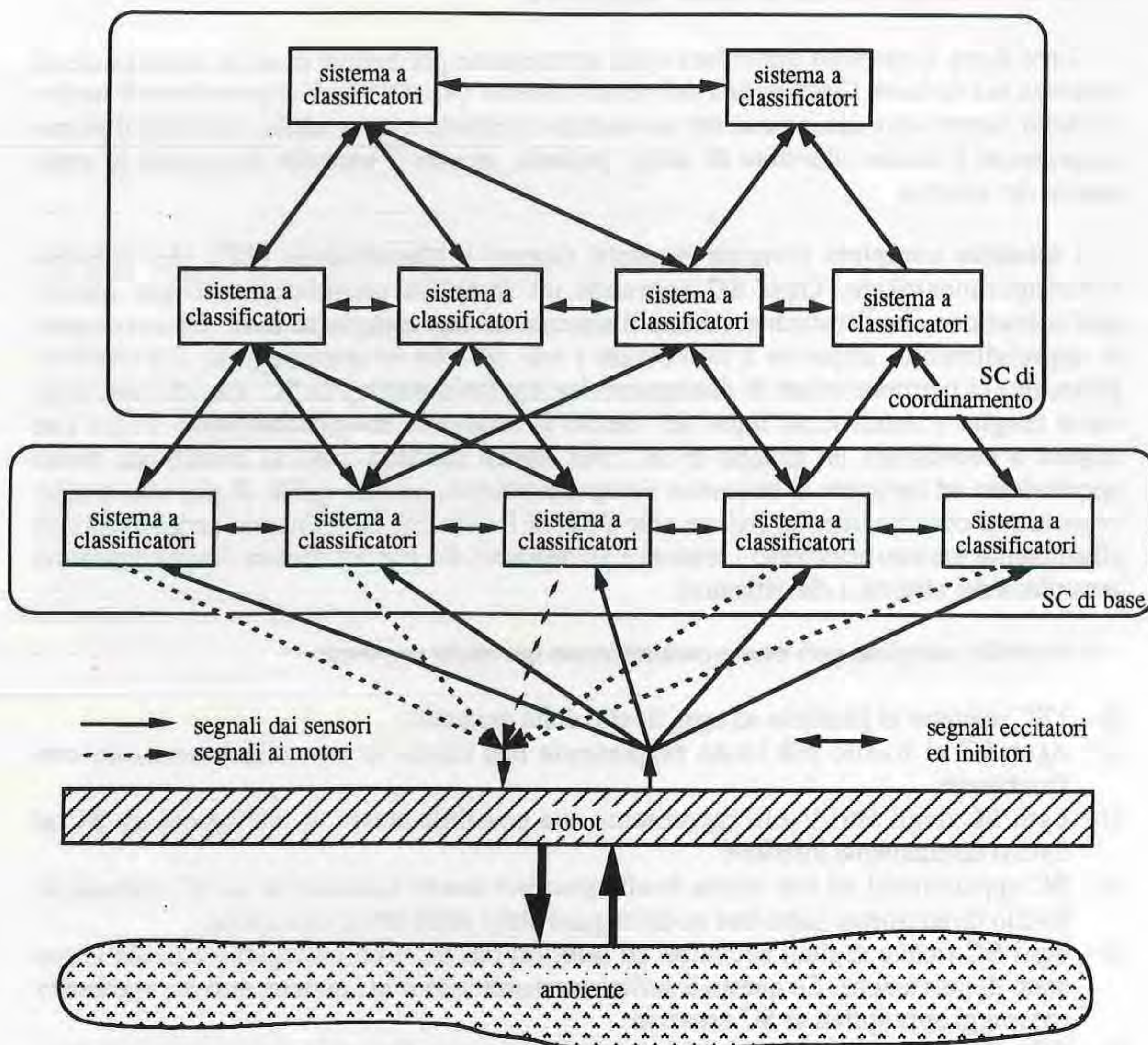


Fig.5.2 - Gerarchia di sistemi a classificatori

Allo stato attuale della sperimentazione il modello completo non è stato interamente implementato in ALECSYS . Alcune delle caratteristiche (e cioè le caratteristiche d, e, f, h, j) saranno introdotte in versioni future. Anche il numero di livelli utilizzati è per ora limitato a due. Nonostante queste limitazioni, come si vedrà nella sezione dedicata alla discussione dei risultati sperimentali, il comportamento del sistema presenta aspetti interessanti e degni di approfondimento. Nella Fig.5.3 è riportato un esempio in cui si utilizza una struttura a due livelli con un SC di secondo livello che coordina due SC di primo livello.

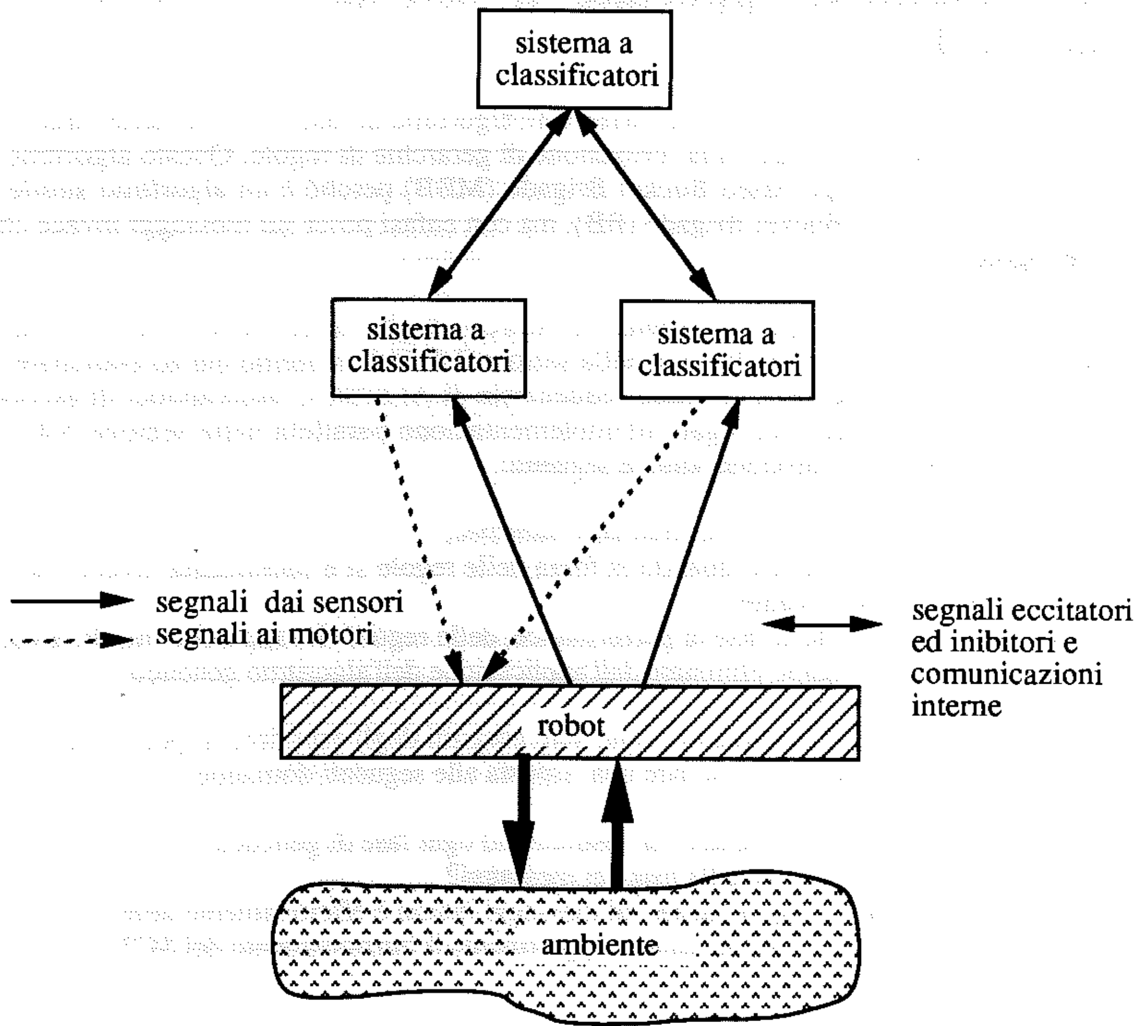


Fig.5.3 - Esempio di gerarchia a due livelli

5.3 Modifiche apportate al sistema a classificatori standard

In questa sezione illustriamo una variante all'algoritmo di distribuzione delle forze che risolve alcuni problemi legati alla formazione di gerarchie di regole. Questo algoritmo è stato chiamato Message-based Bucket Brigade (MBB) perché è un algoritmo simile a quello più noto detto Bucket Brigade (BB), ma con enfasi posta sui messaggi invece che sulle regole.

Inoltre presentiamo quegli aspetti di ALECSYS che differiscono dal sistema a classificatori standard come illustrato nella sezione 2.4. Ci limiteremo qui ad esaminare le innovazioni introdotte nella versione sequenziale di ALECSYS, riservandoci di esporre diffusamente gli aspetti più legati all'implementazione parallela nella sezione 5.4. Le principali innovazioni introdotte sono le seguenti:

- utilizzo di un nuovo operatore chiamato *mutespec*,
- attivazione della genetica[1] quando la forza delle regole si è stabilizzata invece che ad intervalli di tempo costante,
- eliminazione, durante la fase di performance, delle regole che appaiono inutili, invece di aspettare che vengano eliminate dall'applicazione dell'algoritmo genetico.

Per ognuna di queste innovazioni sono stati eseguiti degli esperimenti per valutarne la bontà. Abbiamo inoltre tentato di dare una risposta alle seguenti domande:

- qual è il numero di classificatori da sostituire ad ogni fase di genetica?
- qual è la lunghezza ottima della lista dei messaggi?
- l'utilizzo di catene di regole basate sui messaggi interni è effettivamente utile?
- quali operatori genetici influenzano maggiormente il funzionamento del SC?

5.3.1 L'algoritmo *Message-based Bucket Brigade*

In Fig.5.4 è riportato l'esempio di gerarchia di default che abbiamo introdotto nella sezione 2.4.4.

0 0 ; 0 0 → 0 0
* * ; * * → 1 1

Fig.5.4 - Esempio di gerarchia di default

Affinché tale gerarchia di regole funzioni correttamente ci deve essere un meccanismo che, nella fase di competizione per acquisire il diritto di appendere il messaggio a ML, favorisca la regola più specifica in quei casi in cui entrambe le regole sono soddisfatte e quindi attivate. Nella sezione 2.4.4 avevamo accennato ad un metodo proposto in letteratura per ottenere questo effetto: l'offerta fatta da una regola C_i durante la fase di competizione viene resa proporzionale non solo alla forza $\mathbf{For}_i^{(t)}$ della regola stessa, ma anche alla sua specificità $\mathbf{Spe}_i$. In questo modo si intendono favorire le regole più specifiche, dato che queste hanno un valore di $\mathbf{Spe}_i$ maggiore. Questo approccio però non porta sempre al risultato desiderato. Per dimostrare questa affermazione esaminiamo

[1] Con il termine "attivazione della genetica" intendiamo l'applicazione dell'algoritmo genetico all'insieme dei classificatori.

il comportamento a regime di una gerarchia di default formata da due regole, una specifica - indice s - e una generale - indice d (l'indice d sta per "default").

Sia

$$\mathbf{Off}_i^{(t)} = k\,\mathbf{For}_i^{(t)}\mathbf{Spe}_i \tag{5.1}$$

la quantità offerta dal classificatore C_i per partecipare alla fase di competizione, dove $0 \leq k \leq 1$ è una costante; le equazioni che regolano la variazione della forza del sistema sono (nell'ipotesi che il premio medio **Pre** sia uguale per le due regole: $\mathbf{Pre}_s = \mathbf{Pre}_d = \mathbf{Pre}$):

$$\mathbf{For}_s^{(t+1)} = \mathbf{For}_s^{(t)}(1 - k\mathbf{Spe}_s) + \mathbf{Pre} \tag{5.2}$$

$$\mathbf{For}_d^{(t+1)} = \mathbf{For}_d^{(t)}(1 - k\mathbf{Spe}_d) + \mathbf{Pre} \tag{5.3}$$

Supponendo che la dinamica arrivi a regime ("ss" sta per steady-state) il valore della forza all'istante t+1 è uguale a quello all'istante t e pertanto le due equazioni divengono:

$$\mathbf{For}_s^{ss} = \frac{\mathbf{Pre}}{k\mathbf{Spe}_s} \tag{5.4}$$

$$\mathbf{For}_d^{ss} = \frac{\mathbf{Pre}}{k\mathbf{Spe}_d} \tag{5.5}$$

sostituendo le (5.4) e (5.5) in (5.1) si ottiene

$$\mathbf{Off}_s^{ss} = k\,\mathbf{Spe}_s \frac{\mathbf{Pre}}{k\mathbf{Spe}_s} = \mathbf{Pre} \tag{5.6}$$

$$\mathbf{Off}_d^{ss} = k\,\mathbf{Spe}_d \frac{\mathbf{Pre}}{k\mathbf{Spe}_d} = \mathbf{Pre} \tag{5.7}$$

e pertanto

$$\mathbf{Off}_s^{ss} = \mathbf{Off}_d^{ss} \tag{5.8}$$

Le due regole non sono perciò in grado di gestire la gerarchia di default in modo corretto, dato che l'offerta fatta dalle due regole a regime è uguale. Come risultato secondario si ottiene anche che con la politica sopra proposta il valore di regime delle regole più generali risulta essere più elevato di quelle più specifiche. In [56] è stato proposto un metodo per risolvere questo problema che però funziona solo per gerarchie formate da due regole. Di seguito proponiamo un algoritmo che possiede la proprietà di creare gerarchie di default, basato sull'idea di associare la quantità chiamata "forza" ai messaggi invece che alle regole [39], [40], [41], [32].

L'algoritmo MBB

L'algoritmo MBB è un algoritmo derivato dal BB nel quale la forza invece di misurare l'utilità dei classificatori misura l'utilità dei messaggi. La fase di performance funziona come nel BB, le differenze risiedono nel funzionamento dell'algoritmo di distribuzione del credito. Sia $M_i^{(t)}$ la forza associata ad ogni messaggio generabile i (un messaggio è detto essere generabile se è nella parte azione di almeno una delle regole che compongono il sistema a classificatori). Durante la fase di competizione all'istante t ogni classificatore attivato offre una quantità data da

$$\text{M-Off}_c^{(t)}=M_a^{(t)}+\rho_1 M_{c1}^{(t)}+\rho_2 M_{c2}^{(t)} \tag{5.9}$$

dove $M_a^{(t)}$ è la forza associata al messaggio che rappresenta la parte azione del classificatore C, $M_{c1}^{(t)}$ e $M_{c2}^{(t)}$ sono rispettivamente le medie delle forze dei messaggi che soddisfano la prima e la seconda[2] condizione del classificatore C, ρ_1 e ρ_2 sono le specificità delle due condizioni.

Un messaggio che vince la competizione paga una quantità proporzionale alla sua forza (kM_i^t, dove k è una costante) ad entrambi gli insiemi di messaggi che hanno soddisfatto le condizioni del classificatore da cui è stato impostato. Ad esempio se all'istante t i messaggi r e s hanno soddisfatto la prima condizione del classificatore C, il messaggio q ha soddisfatto la seconda e C ha impostato il messaggio w, allora w pagherà $kM_w^{(t)}$ (cioè la sua forza diminuirà di $kM_w^{(t)}$). Questa quantità viene suddivisa fra i tre messaggi come segue: i messaggi r ed s ricevono $\frac{1}{4}kM_w^{(t)}$ ciascuno, mentre il messaggio q riceve $\frac{1}{2}kM_w^{(t)}$.

L'algoritmo MBB è il seguente.

Passo 0	•	Crea una lista di tutti i messaggi generabili e associa ad ognuno dei messaggi una forza iniziale; vuota ML, poni tutti i classificatori in stato non attivo e poi appendi a ML i messaggi ambientali.
Passo 1	•	Poni in stato attivo tutti i classificatori che hanno le condizioni soddisfatte dai messaggi che stanno su ML; per ogni classificatore si mantengono due liste contenenti i messaggi che hanno soddisfatto le due condizioni.
Passo 2	•	Per ogni classificatore attivo si calcola il valore $\text{M-Off}_c^{(t)}$ come da formula (5.9).
Passo 3	•	Si scelgono i messaggi da appendere alla ML con una probabilità proporzionale al valore $\text{M-Off}_c^{(t)}$.
Passo 4	•	Ogni messaggio i che ha vinto la competizione (e che quindi è stato appeso alla lista dei messaggi) paga la quantità $kM_i^{(t)}$ ai messaggi che hanno causato l'attivazione della regola che lo ha impostato.
Passo 5	•	Lo stato di tutti i classificatori è posto al valore non attivo.
Passo 6	•	Vengono appesi alla ML i nuovi messaggi ambientali e si ritorna al passo 1.

La caratteristica principale di MBB è che la sua applicazione fa sì che i classificatori più specifici vengano attivati con probabilità più alta di quelli più generali. Supponendo infatti che si arrivi a una situazione di regime, la forza di un messaggio m all'istante t+1 è data da

[2] Per semplicità presentiamo l'algoritmo MBB per il caso di classificatori con due condizioni. MBB può essere facilmente generalizzato a SC con classificatori con più di due condizioni.

$$M_m^{(t+1)} = M_m^{(t)} - kM_m^{(t)} + R_m \Rightarrow M_m^{ss} = \frac{R_m}{k}$$

dove R_m è il valor medio del premio ricevuto dal messaggio m. È evidente che a regime la forza M_m^{ss} del messaggio m non dipende dalla specificità, mentre la quantità offerta dal classificatore ne dipende, come risulta dalla equazione (5.9).

Si consideri l'esempio riportato in Fig.5.5. L'automa rappresenta l'ambiente nel quale agisce un sistema a classificatori composto dalle seguenti due regole:

0 * * ; 1 * * → 1 1 1
0 0 0 ; 1 * * → 1 0 0

Nell'automa la variabile S rappresenta lo stato. Lo stato prossimo è determinato una volta che sia noto il valore della variabile di input i (il valore di i viene deciso dal SC). Dopo che è avvenuta una transizione di stato l'automa ritorna al SC la variabile O che contiene il nome del nuovo stato al quale si è portato l'automa (O=S). Il SC riceve anche un premio dato dal valore della variabile R associata alla transizione di stato effettuata. Il SC utilizza la variabile O come messaggio ambientale: il valore di O più un bit di etichetta (per distinguere i messaggi esterni da quelli interni) vengono appesi alla lista MLE.

Supponiamo ad esempio che il SC dia alla variabile i il valore i=1 e che lo stato dell'automa sia S=11: allora lo stato prossimo sarà S=00. Una volta avvenuta la transizione di stato l'automa ritorna al SC la variabile O con il nome del nuovo stato in cui si trova l'automa (O=00). Inoltre il SC riceve il premio R=1.

Nel SC i bit delle condizioni hanno i seguenti significati: il primo bit serve a distinguere i messaggi interni da quelli esterni (1 interno; 0 esterno, cioè dall'ambiente); il secondo e terzo bit corrispondono all'output dell'automa; nella parte azione il primo bit viene di nuovo utilizzato come etichetta, mentre il secondo e terzo bit determinano il valore di i (00 per i=0 e 11 per i=1). Se ad esempio l'automa si trova nello stato S=01 il sistema a classificatori riceverà dall'ambiente il messaggio 001 e quindi[3] potrà attivarsi solo la regola più generale; questa regola determina l'assegnamento i=1 e quindi la transizione allo stato S=10 nell'automa. Se invece lo stato dell'automa fosse stato S=00 allora il messaggio ricevuto dal sistema a classificatori sarebbe stato 000 e quindi entrambi i classificatori avrebbero potuto essere attivati[4]. Un'analisi sommaria mostra che affinché il SC riceva il massimo premio deve impostare messaggi che facciano compiere all'automa un ciclo in quattro mosse. Pertanto la politica migliore è quella di scegliere, quando entrambi i classificatori siano attivati, quello più specifico[5]. Il problema che deve essere risolto dall'algoritmo di distribuzione delle forze è pertanto quello di distribuire i premi in modo tale che la regola più specifica sia favorita rispetto a quella generale.

---

[3] Si fa l'ipotesi che il sistema fosse già in funzione e che quindi sia sempre presente sulla ML un messaggio che inizia con il bit 1.

[4] Questo automa è particolarmente semplice, ed è stato scelto perché erano disponibili i risultati relativi alla performance di un sistema a classificatori uguale a quello da noi utilizzato relativamente a vari metodi di distribuzione del credito (i risultati sono stati da noi riprodotti e sono riportati nel seguito).

[5] Questa politica corrisponde a scegliere il valore di i uguale all'OR logico sui due bit che definiscono lo stato dell'automa.

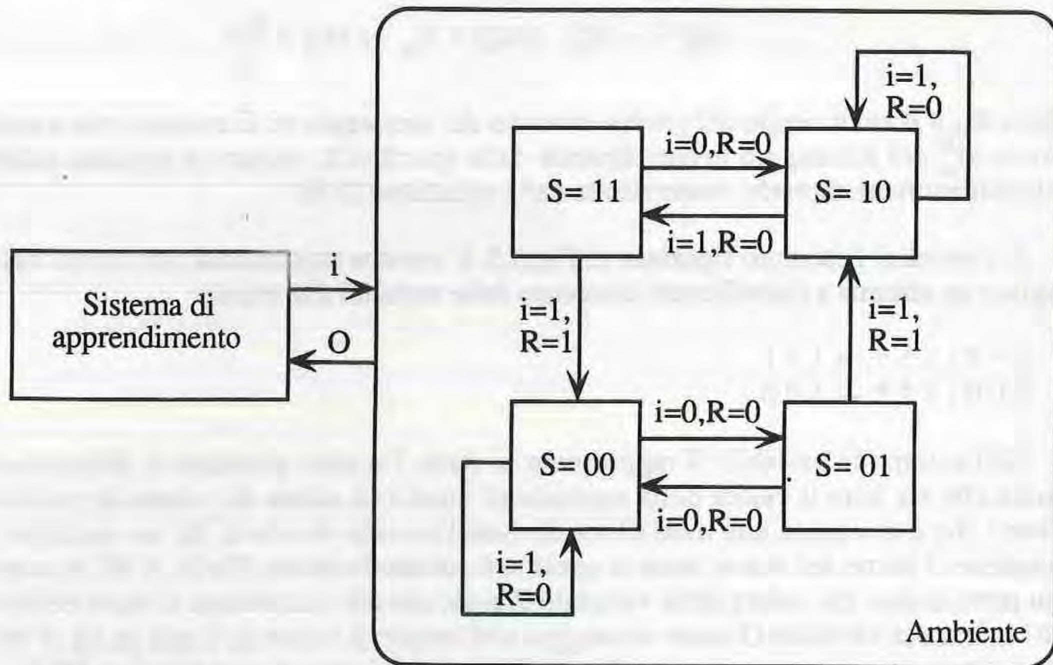


Fig. 5.5 - Automa a stati finiti nel quale il massimo premio si ottiene continuando a ciclare fra i suoi quattro stati

Nelle figure sono riportati i risultati relativi a performance del sistema e andamento della forza dei due classificatori nel caso di applicazione di diversi algoritmi: l'algoritmo bucket brigade standard (Fig.5.6-5.7, vedi anche sezione 2.4.4), l'algoritmo bucket brigade con specificità (Fig.5.8-5.9, questo algoritmo prevede che la specificità intervenga solo nel calcolo della quantità offerta e non in quello della quantità effettivamente pagata [56]) e l'algoritmo MBB (Fig.5.10).

Fig.5.6 - Performance del sistema usando l'algoritmo BB standard:
Forza media: 0.68
Deviazione standard: 0.21

Fig.5.7 - Andamento della forza dei due classificatori (algoritmo BB standard):

Forza media: regola generale = 22.27
regola specifica = 5.49

Deviazione standard: regola generale = 1.94
regola specifica = 0.38

Fig.5.8 - Performance del sistema usando l'algoritmo bucket brigade con specificità (SBB):
Forza media: 0.8
Deviazione standard: 0.17

Fig.5.9 - Andamento della forza dei due classificatori (algoritmo SBB):

Forza media:	regola generale = 8.9
	regola specifica = 4.43
Deviazione standard:	regola generale = 0.78
	regola specifica = 0.25

Fig.5.10 -Performance del sistema usando l'algoritmo message-based bucket brigade (MBB):
Forza media: 0.91
Deviazione standard: 0.12

Come si vede dai grafici l'algoritmo MBB è quello con livello di performance più elevato.

5.3.2 Innovazioni in ALECSYS

Prima di iniziare la progettazione di ALECSYS nella sua versione parallela, abbiamo studiato il comportamento del SC standard per cercare di identificare eventuali aspetti suscettibili di miglioramento. Nel seguito di questa sottosezione riportiamo la descrizione delle modifiche effettuate che sono risultate in un miglioramento delle prestazioni del sistema e i relativi risultati sperimentali. Tutte queste modifiche fanno parte integrante del sistema ALECSYS presentato nella sezione 5.4.

Gli esperimenti sono stati condotti, quando non altrimenti indicato, nel seguente ambiente sperimentale.

Si consideri un robot che si muove nel piano. Sui quattro lati del robot sono posti quattro sensori di luce in modo tale da dividere il piano in quattro semipiani parzialmente ricoprentesi. Le direzioni da cui arriva la luce sono chiamate con il corrispondente nome del segno cardinale (Nord-Ovest-Sud-Est). Se ad esempio la luce si trova nella posizione della Fig.5.11 allora saranno attivi i sensori di luce che percepiscono la luce a Nord e ad Est.

Nei messaggi dai detettori i quattro sensori sono rappresentati da un bit ciascuno: se il valore del bit è uno significa che il sensore corrispondente ha percepito il segnale luminoso, se è zero che non lo ha percepito. Per esempio il messaggio 0 1 1 0 significa che la luce si trova a Sud-Ovest[6].

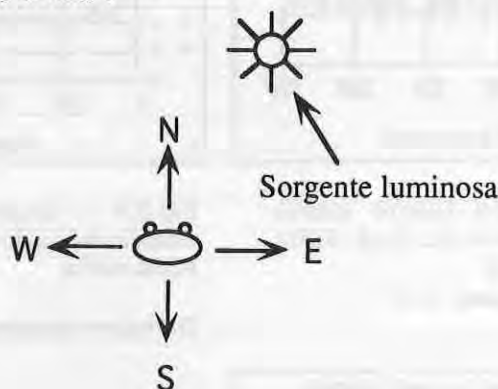


Fig.5.11 - Direzione di percezione della luce del robot

Il robot è in grado di muoversi a 3 velocità: semplice, doppia e nulla, nelle otto direzioni di Fig.5.12.

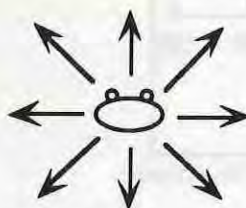


Fig.5.12 - Possibili direzioni di movimento del robot

Un classificatore è quindi una stringa di 15 bit: 5 bit per ognuna delle due condizioni più 5 bit per la parte messaggio. I 5 bit sono a loro volta suddivisi in 4 bit per il messaggio vero e proprio più un bit per l'etichetta[7]. Il problema è quello di apprendere un insieme di classificatori che permetta al robot di inseguire una sorgente luminosa mobile.

In tutti gli esperimenti, qualora non altrimenti specificato, abbiamo utilizzato una popolazione di 240 classificatori, una lista dei messaggi interni di lunghezza 10 e la seguente politica di premiazione: nessun premio se il robot si allontana dalla luce, premio se si avvicina. Chiaramente il robot può avvicinarsi alla sorgente luminosa facendo tre tipi di mosse: il premio massimo (18 punti) verrà dato se fa la mossa di massimo avvicinamento, il premio minimo (6 punti) se si avvicina secondo una delle altre due direzioni (vedi Fig.5.13).

Ulteriori dettagli su questo problema e sul modo in cui ALECSYS apprende una base di classificatori per la sua soluzione verranno dati nella sottosezione 5.5.3. Qui di seguito ci occuperemo solamente dell'effetto che l'introduzione delle modifiche da noi proposte ha sull'efficienza del sistema di apprendimento.

---

[6] Perciò non tutte le possibili combinazioni di bit hanno un senso compiuto: le combinazioni senza senso non possono comunque essere generate dai sensori, se non a causa di errori.

[7] L'etichetta serve a distinguere i messaggi esterni da quelli interni (vedi sezione 5.5).

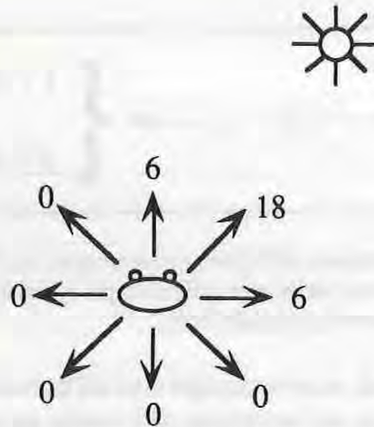


Fig.5.13 - Modalità di premiazione del robot

5.3.2.1 L'operatore *mutespec*

Un problema critico nei SC è causato dal fatto che un classificatore attivato in certi casi genera messaggi "utili" e altre volte messaggi "dannosi". Da questo comportamento risulta che tale classificatore assume un valore intermedio tra quello che assumerebbe se generasse solo messaggi utili e quello che assumerebbe se generasse solo messaggi dannosi: ciò determina un effetto negativo sulla performance del sistema (che infatti continua a mantenere un classificatore che genera, parzialmente, messaggi dannosi), e impedisce al classificatore stesso di raggiungere un valore di regime. Quest'ultimo aspetto, come vedremo nella sottosezione 5.3.2.2, complica la valutazione del raggiungimento dello stato di regime da parte di tutto il sistema e di conseguenza la determinazione dell'istante temporale in cui applicare la fase genetica dell'algoritmo di apprendimento. Per risolvere almeno parzialmente questo inconveniente introduciamo un operatore che chiamiamo *mutespec*.

L'idea base è di prendere in considerazione una stima della varianza nei premi ottenuti ad ogni attivazione. Questo valore è piccolo per quei classificatori che impostano messaggi sempre esatti oppure sempre sbagliati, alto per i classificatori rimanenti.

L'operatore *mutespec* prende un classificatore con elevata varianza, sceglie in modo casuale uno dei simboli "don't care" e aggiunge all'insieme dei classificatori due nuovi classificatori che presentano rispettivamente i simboli 0 e 1 al posto del simbolo # considerato (per far posto ai due nuovi classificatori vengono eliminati due classificatori fra quelli con forza più bassa).

Si consideri il seguente esempio: nel sistema a classificatori di Fig.5.14 c'è un classificatore che ha una condizione che contiene due simboli #. Supponiamo che il classificatore riceva un premio alto **Pre** ogni qualvolta la sua prima condizione è soddisfatta dal messaggio 1100 e prenda un premio basso **Pun** quando è soddisfatta dal messaggio 0110. Nell'ipotesi che i due messaggi si presentino con uguale frequenza il premio medio ricevuto dal classificatore sarà (**Pre**+**Pun**)/2. Questo valore non rispecchia l'utilità del classificatore in nessuno dei due casi (inoltre, come già detto, questa situazione mantiene elevato il valore della varianza della differenza tra premio e quantità pagata dal classificatore[8] disturbando così il meccanismo di valutazione del raggiungimento della situazione di regime).

---

[8] Si ricordi che la quantità pagata da una classificatore è proporzionale alla sua forza.

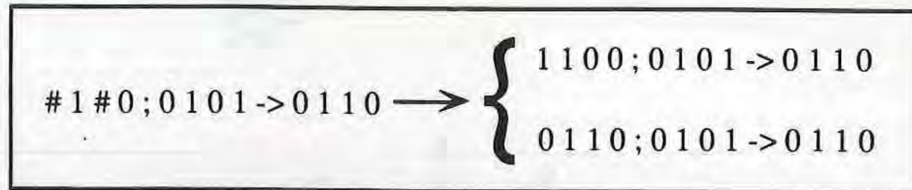


Fig.5.14 - Esempio di applicazione dell'operatore *mutespec*: un classificatore che contiene simboli # ed il cui valore della forza ha elevata varianza è utilizzato per generare due nuovi classificatori più specifici (il classificatore genitore rimane nella popolazione).

L'operatore *mutespec* può essere paragonato all'operatore di mutazione, orientato però all'aumento della specificità nel tentativo di trovare un classificatore più "preciso" nella fase di match. Quest'effetto è ottenuto con un'efficienza maggiore di quella ottenibile per mezzo del semplice operatore di mutazione. Infatti l'operatore di mutazione opera indistintamente su tutti i bit (sia che essi valgano zero, uno o don't care) e può operare, casualmente, sia in direzione di una maggiore che di una minore specificità del classificatore. Inoltre, a differenza della mutazione che modifica un classificatore, *mutespec* mantiene il classificatore generatore ed introduce due nuove regole (che vanno a sostituire due regole con valore di forza basso).

Abbiamo paragonato diverse combinazioni di operatori effettuando delle simulazioni i cui risultati sono riportati in Fig.5.15. Nel primo esperimento usiamo solo la genetica di background (riproduzione, crossover e mutazione), nel secondo usiamo la genetica di background più gli operatori di cover detector e cover effector, nel terzo la genetica di background più l'operatore mutespec e nel quarto tutti gli operatori. I risultati mostrano l'utilità dell'operatore mutespec, la cui applicazione migliora sia il livello di performance raggiunto che la velocità con cui tale livello viene raggiunto. È notevole osservare che la performance ottima (100% delle mosse corrette) viene ottenuta solo con l'utilizzo contemporaneo di tutti gli operatori.

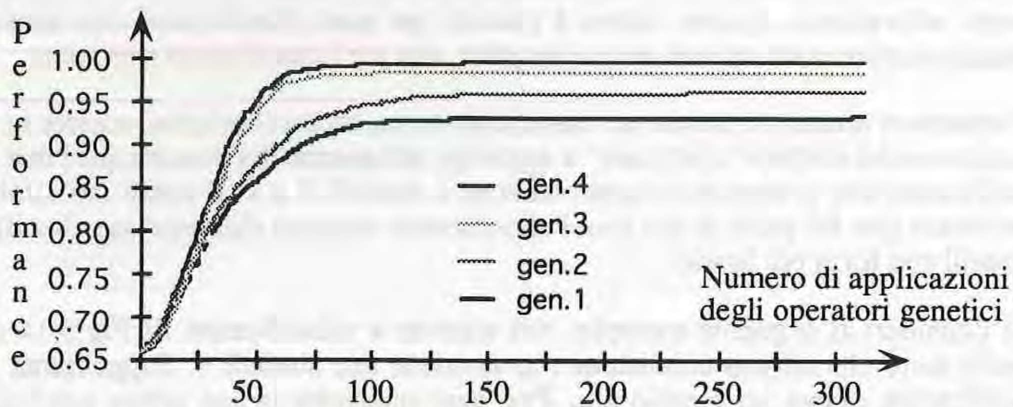


Fig.5.15 - Performance (numero di mosse corrette su numero di mosse totale) del SC al variare del tipo di operatori genetici utilizzati:

- gen.1 - solo genetica di background (chiamata ogni 400 cicli)
- gen.2 - genetica di background + operatori di cover detector e cover effector
- gen.3 - genetica di background + operatore mutespec
- gen.4 - genetica di background + operatori di cover detector e cover effector + operatore mutespec

5.3.2.2 Valutazione del raggiungimento della situazione di regime

Come visto nella sezione 2.4, l'algoritmo di bucket brigade ha la funzione di attribuire un valore detto forza ad ogni classificatore sperando che tale forza rispecchi l'utilità del classificatore stesso per il sistema di apprendimento. Tali forze rappresentano infatti la "fitness" utilizzata dall'algoritmo genetico, ed è quindi importante che queste forze rispecchino correttamente l'utilità dei classificatori cui sono associate. Affinché ciò avvenga è necessario non solo che la distribuzione dei premi avvenga in modo corretto, ma anche che all'algoritmo di distribuzione dei premi sia dato un tempo sufficiente perché i valori delle forze si stabilizzino sui valori di regime. In questa sezione ci occupiamo del secondo problema.

Normalmente l'algoritmo genetico viene applicato ogni NC cicli, dove NC è un parametro determinato sperimentalmente e il cui valore ottimo dipende dal problema in esame. La determinazione sperimentale di questo valore è onerosa e non è detto a priori che NC non sia variabile nel tempo (cioè l'intervallo ottimo tra due chiamate della genetica potrebbe non essere costante durante l'evolversi dell'apprendimento[9]): è quindi ragionevole tentare di introdurre un meccanismo automatico di valutazione del raggiungimento della situazione di regime che permetta di chiamare la genetica di background non appena tale stato è raggiunto. È chiaro che un modo per stabilire se il sistema a classificatori ha raggiunto l'equilibrio potrebbe essere quello di verificare che tutti i classificatori abbiano raggiunto valori stabili. Come abbiamo visto nella sezione precedente però vi possono essere classificatori il cui valore oscilla. Causa di queste oscillazioni potrebbe essere ad esempio il presentarsi di una situazione come quella riportata in Fig.5.16. In questo caso, che è simile a quello presentato in Fig.5.14, una applicazione dell'operatore mutespec potrebbe risolvere il problema. In realtà in classificatori in cui il numero di simboli # è maggiore di uno, l'operatore *mutespec* è costretto a scegliere in modo casuale un simbolo # da tramutare in 0 e 1, senza poter avere la certezza di aver scelto quello corretto. Pertanto alcuni classificatori potrebbero richiedere molto tempo prima di raggiungere l'equilibrio.

Classificatore oscillante: 0 1 # 1 ; 0 1 1 0 -> 1 1 1 1
ML -> 0 1 1 1 implica che il messaggio 1 1 1 1 è molto utile
ML -> 0 1 0 1 implica che il messaggio 1 1 1 1 non è molto utile

Fig.5.16 - Esempio di classificatore oscillante

È quindi necessario introdurre una misura che permetta di valutare il raggiungimento dello steady-state prescindendo dalla presenza di qualche classificatore oscillante. A tal fine introduciamo il concetto di *energia* $E_{sc}(t)$ di un sistema di n classificatori all'istante t.

$$E_{sc}^{(t)}=\sum_{i=1}^{n} \mathbf{For}_i^{(t)} \qquad (5.10)$$

dove $\mathbf{For}_i^{(t)}$ è la forza del classificatore i-esimo all'istante t.

L'utilizzo dell'energia permette di smorzare l'effetto di eventuali classificatori che continuano ad oscillare, nell'ipotesi in cui i classificatori oscillino in modo asincrono. A

[9] Ciò è sicuramente vero nel nostro sistema in quanto utilizziamo operatori come il *mutespec* (vedi sezione 5.3.2.1) o il cover effector (vedi sez.2.4.5) che possono influenzare la composizione dell'insieme dei classificatori in ogni momento in modo differente e non prevedibile.

regime il valore dell'energia è costante, dato che la quantità di premi che entra in SC è uguale alla quantità che ne esce. Pertanto il valore nullo della deviazione standard dell'energia è un indicatore del raggiungimento della situazione di regime.

In Fig.5.17 sono riportati gli andamenti tipici della deviazione standard dell'energia e della somma delle deviazioni standard dei singoli classificatori, mentre in tabella 5.1 sono riportati i rispettivi valori medi e deviazioni standard della deviazione standard. Si vede come il valore medio della deviazione standard dell'energia e la sua deviazione standard stessa siano molto minori di quelli della somma delle deviazioni standard dei singoli classificatori, e che quindi la deviazione standard dell'energia può essere proficuamente utilizzata per dare una valutazione del raggiungimento del regime.

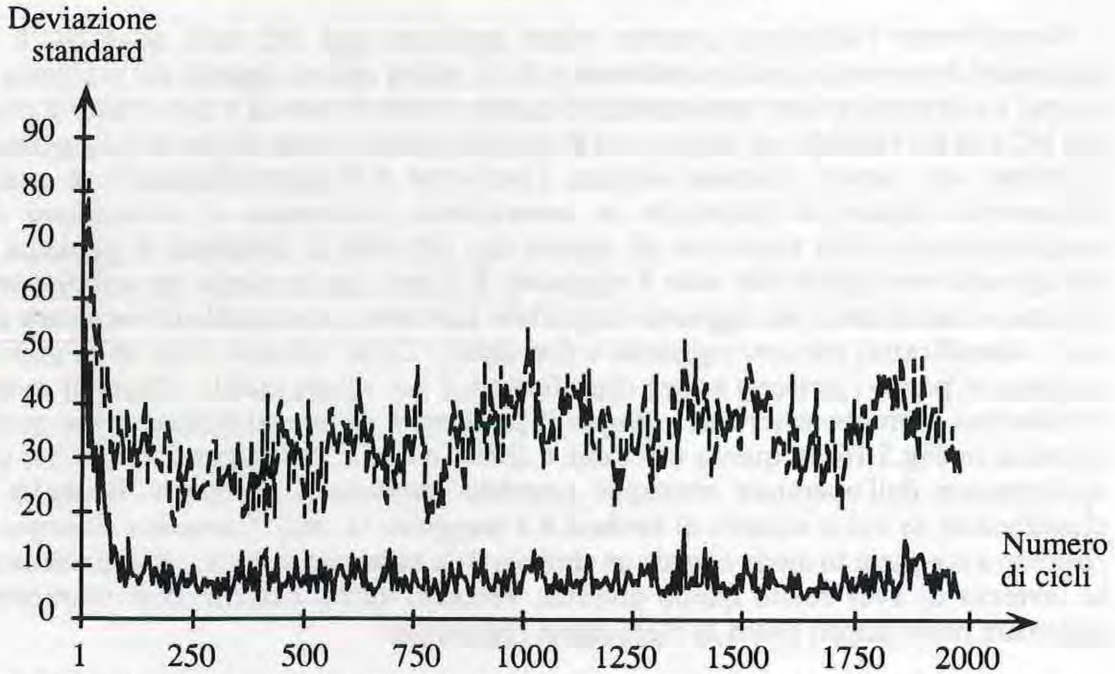


Fig.5.17 - Andamento tipico della deviazione standard dell'energia (in basso) e della somma delle deviazioni standard dei singoli classificatori (in alto).

Tab. 5.1 - Valore medio e deviazione standard[10] della deviazione standard dell'energia e della somma delle deviazioni standard dei singoli classificatori

	Valore medio	Deviazione standard
Deviazione standard dell'energia	7.36	2.36
Somma delle deviazioni standard dei singoli classificatori	33.61	7.32

In pratica quello che viene fatto è il porre come condizione di raggiungimento dello steady-state la seguente: SC è allo steady-state se negli ultimi k cicli il valore dell'energia si è mantenuto all'interno dell'intervallo [E_{Min} , E_{Max}]. I valori E_{Min} ed E_{Max} sono

[10] I valori sono stati calcolati escludendo i primi 250 cicli.

rispettivamente il minimo e il massimo valore raggiunti dall'energia nei k cicli precedenti quelli in esame[11].

5.3.2.3 Numero di regole da sostituire ad ogni fase di genetica

In Fig.5.18 è riportato l'andamento tipico dell'energia: i picchi rappresentano l'immissione di nuove regole da parte della genetica. Come si vede dalla Fig.5.18 il numero di cicli che intercorre tra due chiamate della genetica non è costante. L'ampiezza di questo intervallo dipende sia dal tipo di classificatori generati dall'AG che dal loro numero NR, che è pertanto un parametro importante da valutare.

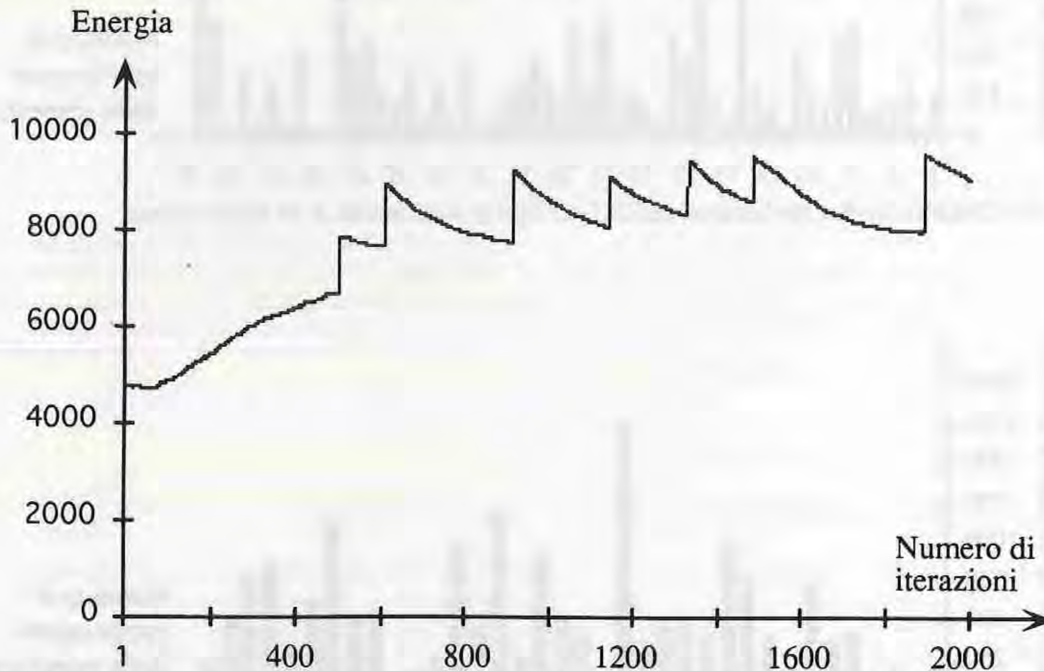


Fig.5.18 - Nel grafico è riportato l'andamento dell'energia. I picchi corrispondono alla sostituzione di classificatori giudicati inutili con classificatori creati dall'applicazione dell'algoritmo genetico. Si può osservare come l'algoritmo genetico venga chiamato quando il valore dell'energia si è stabilizzato.

Quando la genetica viene chiamata ad intervalli di tempo costanti è conveniente porre NR al valore più alto che non determini una caduta di performance del sistema (in questo modo si ha che il numero di classificatori testati è il massimo possibile rispettando il vincolo che il sistema continui a funzionare correttamente; se si introducesse un numero maggiore di classificatori nuovi si sarebbe costretti, per far loro posto, ad eliminare dei classificatori utili determinando così una probabile diminuzione di performance del sistema). Nel caso del sistema con genetica chiamata a regime (SSCS - Steady-State Classifier System) si hanno invece due effetti contrastanti: aumentando NR aumenta il numero di classificatori testati ad ogni genetica, ma nel contempo il sistema impiega (mediamente) più tempo per arrivare a regime e quindi l'algoritmo genetico viene applicato meno spesso. Riportiamo di seguito i risultati di esperimenti volti a valutare l'effettivo comportamento di SSCS. Nelle Fig.5.19, Fig.5.20, Fig.5.21 si può osservare l'andamento del numero di iterazioni intercorrente tra due chiamate della genetica per valori di NR rispettivamente NR=24, NR=48, NR= 72 (su una popolazione totale di 200 classificatori): al crescere di NR l'intervallo medio tra due genetiche aumenta. Inoltre,

[11] In questo modo si escludono i casi in cui $E_{cs}(t)$ sta diminuendo (perché il nuovo E_{Min} sarà minore di quello vecchio), quelli in cui $E_{cs}(t)$ sta aumentando (perché il nuovo E_{Max} sarà maggiore di quello vecchio), e quelli in cui $E_{cs}(t)$ oscilla eccessivamente.

nonostante la minor frequenza di attivazione dell'AG, il numero totale di classificatori testati risulta essere maggiore per NR grande.

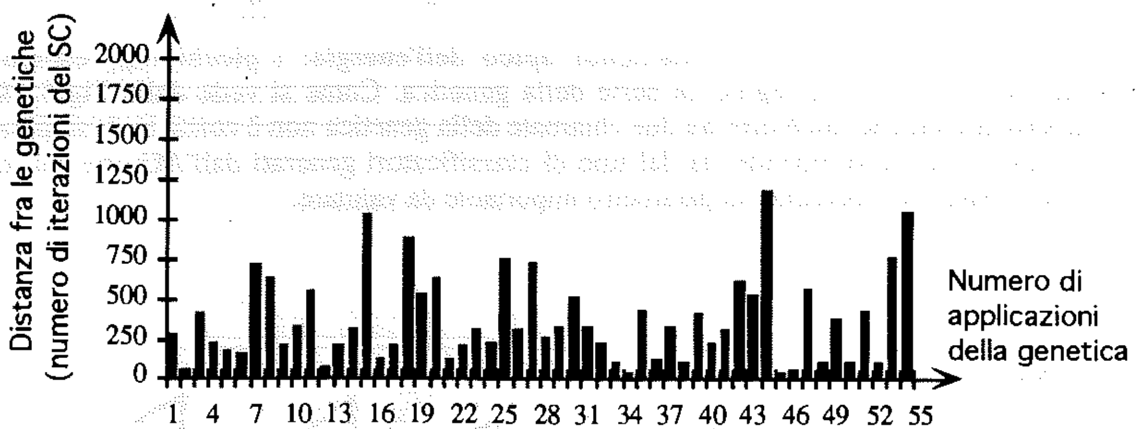


Fig.5.19 - Distanza fra due applicazioni dell'AG nel caso di inserimento di 24 regole nuove

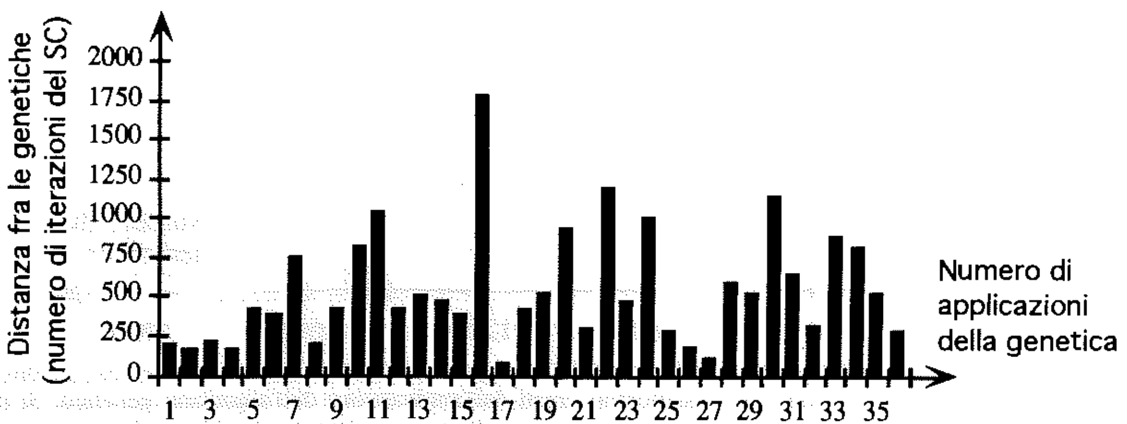


Fig.5.20 - Distanza fra due applicazioni dell'AG nel caso di inserimento di 48 regole nuove

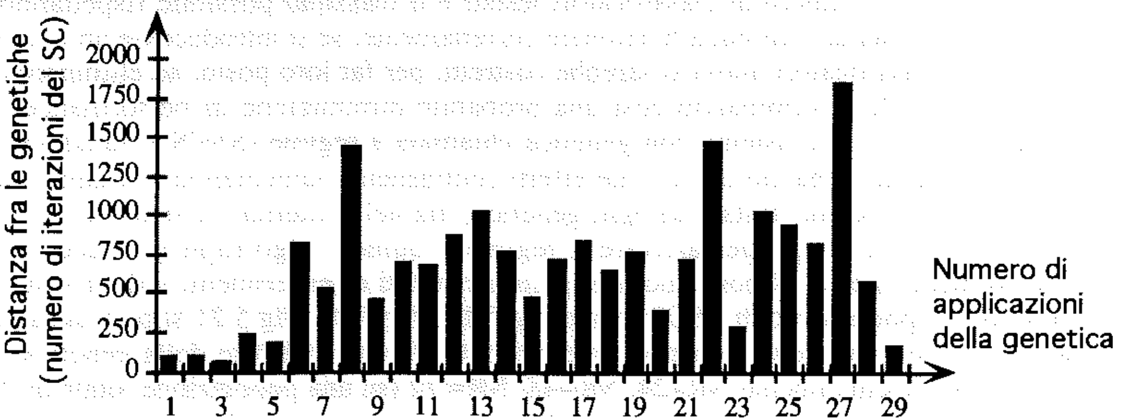


Fig.5.21 - Distanza fra due applicazioni dell'AG nel caso di inserimento di 72 regole nuove

Se consideriamo però la performance del sistema si vede che, tabella 5.2, la performance di SSCS ha un massimo per NR=48. Ciò è dovuto al fatto che nonostante

con NR=72 venga valutato un numero maggiore di classificatori, si verifica una diminuzione di efficienza nei transitori (a causa del maggior numero di classificatori che propongono azioni errate nel periodo immediatamente successivo ad una applicazione dell'AG) e vengono eliminati con più alta probabilità classificatori utili a causa del più alto numero di nuovi classificatori introdotti.

Per valutare se applicare la genetica quando il sistema è a regime sia effettivamente utile abbiamo confrontato SSCS con un SC standard (cioè con genetica chiamata ad intervalli fissi). Gli esperimenti sono stati fatti per vari valori del numero di regole nuove introdotte dalla genetica per entrambi i sistemi e utilizzando come intervallo tra due genetiche nel caso dell'algoritmo standard un valore buono determinato sperimentalmente. I risultati sono riportati in tabella 5.2. Si vede come la performance di SSCS risulti essere la migliore. Si può anche osservare che il numero medio di cicli tra due applicazioni dell'AG in SSCS risulta essere vicino al valore ottimo sperimentale ottenuto per SC standard.

Tab.5.2 - Performance del sistema a classificatori e di SSCS per diversi valori NR.

Tipo di sistema a classificatori	NR (numero di regole nuove introdotte dall'AG)	Numero di cicli tra due applicazioni dell'AG[12]	Numero totale di regole esplorate in 20000 cicli	Performance relativa[13] (fatta 100 la performance migliore)
SC standard	24	500	960	91.9
SSCS	24	366	1311	96.6
SC standard	48	500	1920	75.5
SSCS	48	544	1764	100
SC standard	72	500	2880	82.9
SSCS	72	676	2130	91.7

5.3.2.4 Utilizzo di un insieme ridotto di regole

È evidente, da un esame dell'algoritmo di performance presentato nella sezione 2.4.1, che la complessità della fase di matching di un ciclo dell'algoritmo è funzione lineare del numero di regole, della lunghezza della lista dei messaggi e del numero di condizioni in ogni regola. In questa sezione presentiamo un metodo per contenere tale complessità agendo sul numero totale di regole effettivamente utilizzate. L'idea consiste nell'inibire la fase di matching per quelle regole che assumono un valore della forza al di sotto di un certo valore soglia. In questo modo si ottiene il risultato di non rallentare inutilmente l'esecuzione dell'algoritmo valutando la possibilità di attivazione di regole che comunque avrebbero probabilità bassissima di impostare effettivamente un messaggio. Infatti, mentre allo stato iniziale tutte le regole hanno un valore di forza simile, man mano che procede la computazione, a causa dell'applicazione dell'algoritmo di distribuzione dei premi, alcune regole diventeranno più forti delle altre, aumentando così la probabilità di essere attivate. Viceversa le regole che non ricevono premi, a causa sia delle eventuali

---

[12] Nel caso del sistema SSCS viene riportato il numero medio di cicli. Nel caso di SC standard la genetica viene chiamata ogni 500 cicli (valore ottenuto sperimentalmente per il quale le prestazioni del sistema sono risultate le migliori).

[13] La performance è calcolata come la somma di tutti i premi ricevuti durante i 20000 cicli di durata dell'esperimento. Questo indice da una indicazione aggregata sia dell'andamento della curva (tempo necessario per arrivare alla performance massima) che del valore massimo di performance raggiunto.

punizioni che del meccanismo di tassazione[14], vedranno diminuire il valore della propria forza e conseguentemente diminuirà nel tempo la loro probabilità di vincere un'asta e di poter così influire attivamente sul comportamento del sistema di apprendimento automatico. Continuare a effettuare la fase di matching per queste regole si traduce quindi in uno spreco di capacità di calcolo. Si tratta quindi di stabilire il criterio da utilizzare per fissare il valore soglia al di sotto del quale le regole vengono escluse dalla fase di matching. L'utilità di un approccio di questo tipo è suggerita dai risultati sperimentali sulla distribuzione delle forze in una base di regole che si sta evolvendo; come si può vedere in Fig.5.22, la distribuzione tipica è (tranne che nelle fasi iniziali): poche regole forti, molte regole deboli.

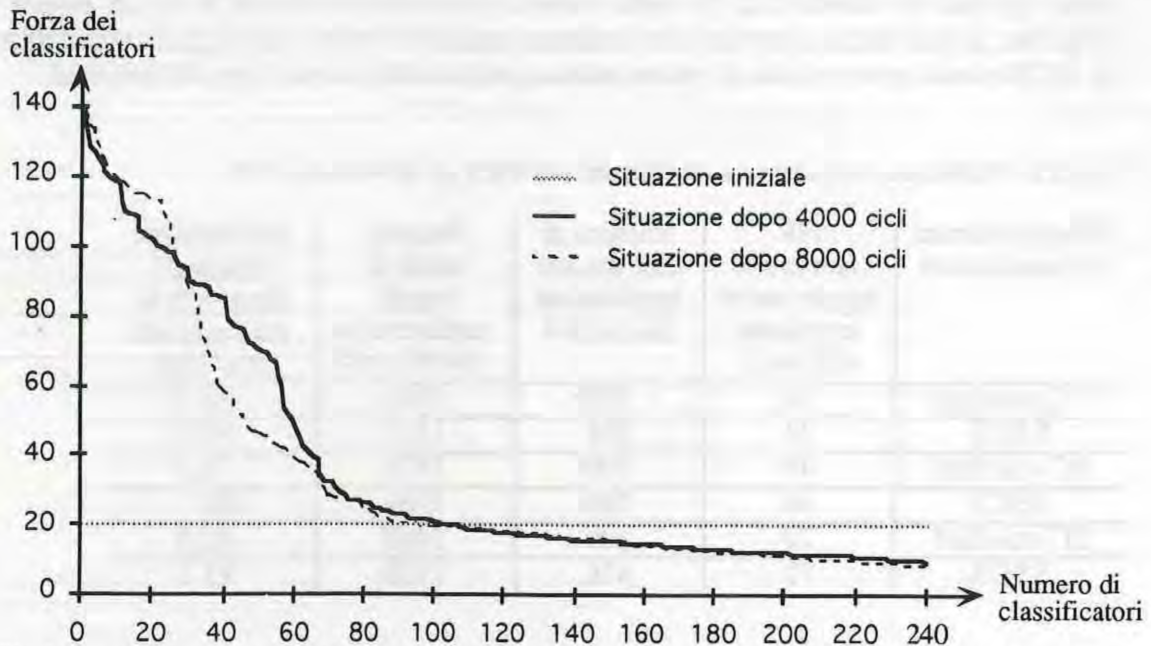


Fig.5.22 - Andamento tipico delle forze dei classificatori a regime: in ascissa sono riportati i classificatori in ordine di forza decrescente e in ordinata la relativa forza: dopo 4000 cicli circa 100 classificatori hanno una forza maggiore di quella iniziale.

Il meccanismo utilizzato per fissare il valore soglia è il seguente:

- le n regole vengono ordinate secondo la loro forza in senso decrescente;
- viene calcolata la forza media $\mathbf{m(t)} = \frac{\Sigma \mathbf{For}_c^{(t)}}{\mathbf{n}}$, dove $\mathbf{For}_c^{(t)}$ è la forza del classificatore C all'istante t;
- tutte le regole con $\mathbf{For}_c^{(t)} < k \cdot \mathbf{m}(t)$, con k costante arbitraria (nei nostri esperimenti i valori migliori sono risultati essere compresi fra 0.35 e 0.45; il valore utilizzato negli esperimenti riportati è k=0.4), vengono eliminate.

È da notare che il criterio adottato porta alla eliminazione di pochi classificatori nelle fasi iniziali e di molti nelle fasi più avanzate. Dal grafico di Fig.5.22 si vede infatti che mentre nella situazione iniziale della computazione tutti i classificatori hanno la stessa forza, e perciò nessuno di essi viene eliminato, dopo 4000 cicli la differenziazione tra le

[14] Tutti i classificatori pagano, ad ogni ciclo, una "tassa" proporzionale alla loro forza. Con questa tassa si penalizzano quei classificatori che non si attivano mai e che perciò continuerebbero indefinitamente ad occupare spazio nell'insieme dei classificatori senza essere di nessuna utilità.

forze è già notevole ed è ancora più accentuata dopo 8000. Ciò determina l'eliminazione di un buon numero di classificatori (il numero di classificatori effettivamente eliminato dipende ovviamente dal valore k della soglia sopra definita).

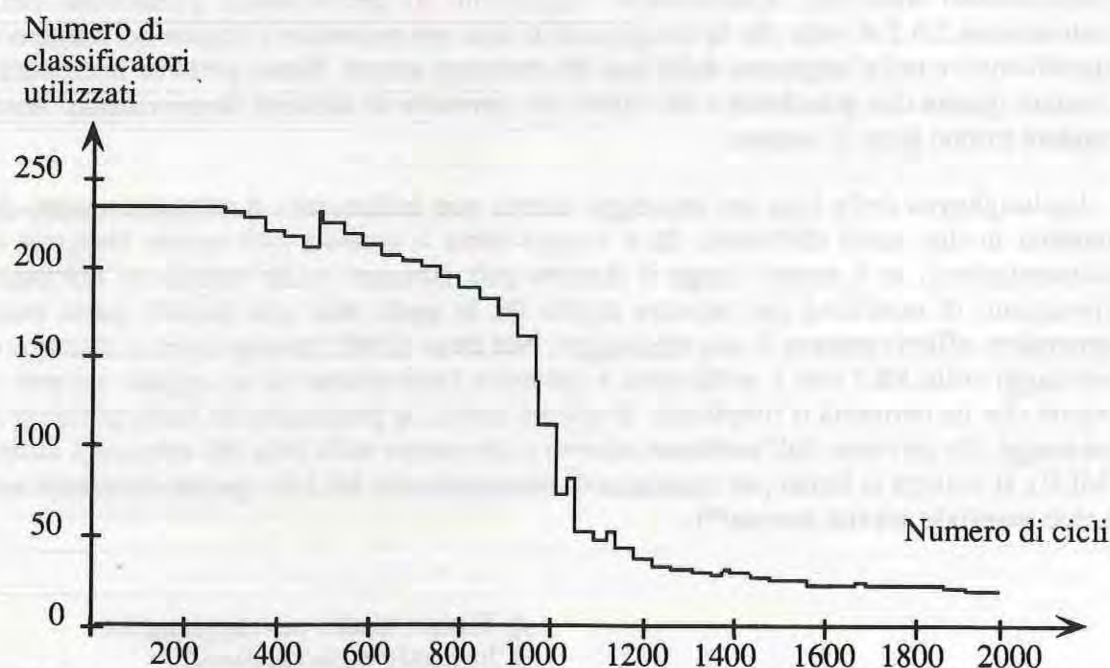


Fig.5.23 - Andamento tipico del numero di classificatori effettivamente utilizzati in funzione del numero di cicli.

Per quanto riguarda la performance del sistema abbiamo ottenuto i risultati riportati in tabella 5.3. Si può osservare come il sistema a numero di classificatori variabile dia risultati migliori per tutti gli indici osservati.

Tab.5.3 - Confronto tra sistema con numero di classificatori costante e sistema con numero di classificatori variabile (numero totale di cicli 8000)

	Performance massima raggiunta	Cicli per raggiungere performance 67 %	Tempo totale per fare 8000 cicli (sec)	Tempo medio per ciclo (sec)
Numero classificatori costante	78 %	3200	6061	0.75
Numero classificatori variabile	92 %	1800	3158	0.39

Abbiamo anche osservato che l'andamento del grafico della performance è, nel caso con numero di classificatori variabile, meno frastagliato: ciò è dovuto alla presenza di un minor numero di regole potenzialmente sbagliate.

5.3.2.5 Lunghezza ottima della lista dei messaggi interni

In questa sezione studiamo come varia la performance del sistema a classificatori al variare della lunghezza della lista dei messaggi interni[15] (MLI) e del numero di classificatori utilizzati. Esaminando l'algoritmo di performance presentato nella sottosezione 2.4.2 si vede che la complessità di una sua iterazione è lineare nel numero di classificatori e nella lunghezza della lista dei messaggi interni. Siamo pertanto interessati a limitare queste due grandezze a un valore che permetta di ottenere buoni risultati senza rendere troppo lento il sistema.

La lunghezza della lista dei messaggi interni può influenzare il comportamento del sistema in due modi differenti. Se è troppo corta il sistema può essere incapace di autosostenersi, se è troppo lunga il sistema può sprecare molto tempo ad effettuare operazioni di matching per attivare regole fra le quali solo una piccola parte potrà appendere effettivamente il suo messaggio. Nel caso di MLI troppo corta il numero di messaggi sulla MLI non è sufficiente a garantire l'attivazione di un uguale numero di regole che ne permetta il rimpiazzo. In questo modo, se prescindiamo dalla presenza di messaggi che arrivano dall'ambiente esterno e che vanno sulla lista dei messaggi esterni (MLE), il sistema si ferma per mancanza di messaggi sulla MLI (in questa situazione non è cioè possibile attività interna[16]).

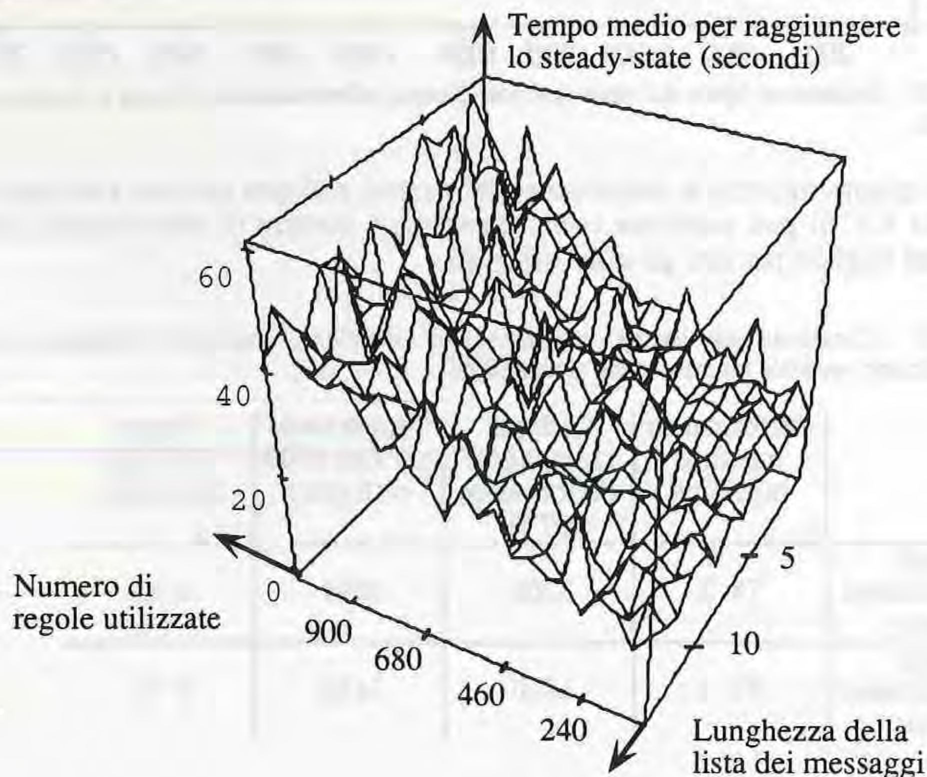


Fig.5.24 - Tempo medio (su cinque prove) necessario a raggiungere la situazione di regime a partire dalla popolazione iniziale in funzione del numero di regole nella popolazione e della lunghezza della lista dei messaggi interni

[15] Si ricorda che la lista dei messaggi ML, introdotta nella sezione 2.4.2, è la concatenazione di due liste: quella dedicata ai messaggi interni (MLI) e quella dedicata ai messaggi esterni (MLE).

[16] Condizione necessaria affinché i SC mantengano delle strutture dinamiche con cui creare un modello del mondo è che vi sia attività interna.

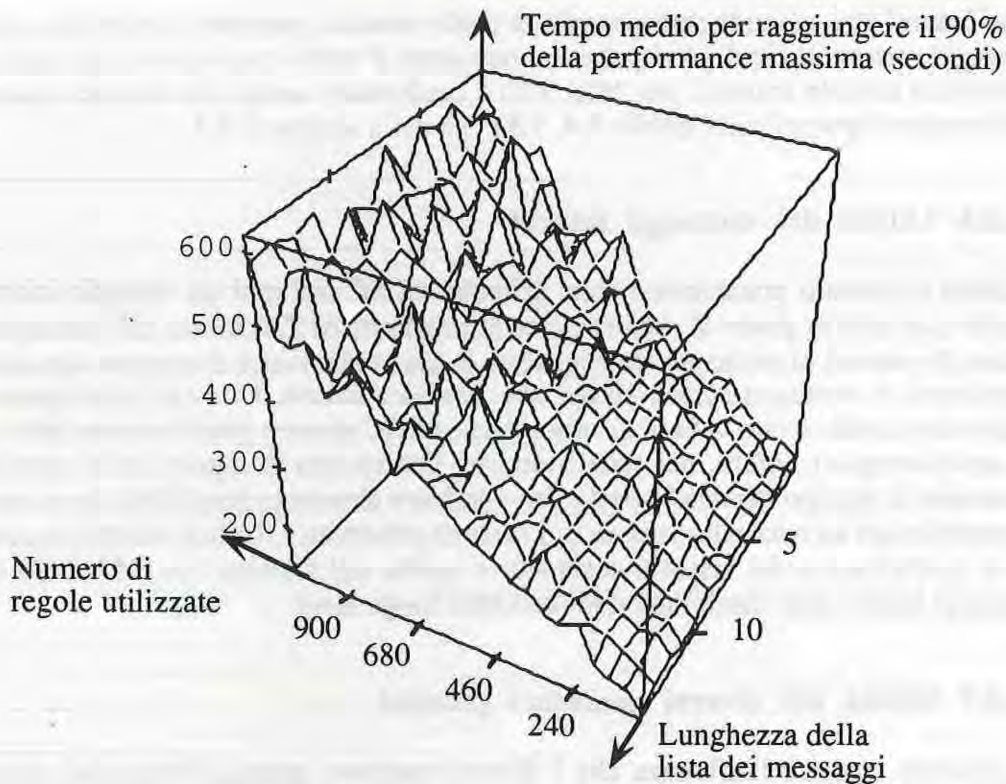


Fig.5.25 - Tempo medio (su cinque prove) necessario a raggiungere il 90% del valore di performance massimo

Una elevata cardinalità dell'insieme delle regole, rispetto ad una bassa, presenta i seguenti vantaggi:

- l'algoritmo genetico può introdurre un più elevato numero di classificatori senza che la performance istantanea ne sia eccessivamente disturbata,
- la più elevata quantità di informazione memorizzata nella popolazione di classificatori permette una ricerca più efficace e più veloce nello spazio dei classificatori possibili.

D'altra parte, come detto, l'effetto negativo risiede nell'incremento del tempo di esecuzione di un singolo ciclo.

Per valutare l'effetto che diverse combinazioni di valori di questi due parametri hanno sulla performance del sistema abbiamo effettuato alcuni esperimenti. In Fig.5.24 è riportato il numero di cicli necessario a raggiungere lo stato di regime al variare della lunghezza di MLI e del numero di regole utilizzate. Questo grafico mostra come il numero di cicli cresca al crescere del numero di regole utilizzato, mentre rimane costante al variare della lunghezza di MLI. Lo stesso andamento può essere osservato nel grafico di Fig.5.25 dove nell'asse verticale riportiamo il tempo necessario per raggiungere un valore fissato di performance (nel nostro caso il valore 0.9, con 1 performance massima).

La irrilevanza della lunghezza della MLI può essere giustificata dal fatto che per questo particolare compito i messaggi interni sembrano essere inutili (se non dannosi, vedi sezione 5.5.3). Pertanto i classificatori che appendono messaggi interni non sopravvivono e quindi il numero totale di messaggi interni generati dal sistema tende a essere molto basso. Ciò fa sì che all'aumentare della lunghezza della MLI il funzionamento del sistema non cambi (in pratica, la maggiore capacità della MLI rimane largamente inutilizzata). Unica eccezione si ha nella fase iniziale in cui sono presenti molti

classificatori che, essendo stati generati in modo casuale, generano un elevato numero di messaggi interni: in Fig.5.24 si può osservare come il tempo per arrivare allo steady-state dallo stato iniziale aumenti con MLI. Ciò è confermato anche dai risultati relativi alla performance riportati nelle tabelle 5.4, 5.5 e 5.6 della sezione 5.5.3.

5.3.2.6 Utilità dei messaggi interni

Come accennato precedentemente, affinché un SC sviluppi un modello interno del mondo (sia cioè in grado di memorizzare le regolarità dell'ambiente che percepisce per mezzo dei sensori in modo tale da migliorare la sua performance man mano che accumula esperienza), è necessario che sviluppi una dinamica interna. Con ciò intendiamo che le regole che costituiscono la base di conoscenza del SC devono poter formare dei cicli che si autosostengono. Infatti, nel caso contrario, l'unico tipo di regola che il sistema può apprendere è del tipo stimolo-risposta. Per verificare almeno la possibilità che si sia creato un modello del mondo nella sezione 5.5 saranno presentati i risultati relativi al confronto tra la performance del sistema completo e quella del sistema con ML nella quale i messaggi interni sono disabilitati (cioè con MLI lunga zero).

5.3.2.7 Utilità dei diversi operatori genetici

Abbiamo studiato l'influenza che i diversi operatori genetici hanno sul sistema di apprendimento. Gli operatori considerati sono la riproduzione, il crossover, la mutazione, e il cover effector. La genetica di background (riproduzione, crossover e mutazione) viene chiamata a regime. L'esperimento è stato fatto nel seguente modo: ad ogni classificatore abbiamo associato un'etichetta che indica il nome dell'ultimo operatore da cui è stato modificato. All'istante precedente la chiamata di una fase di genetica per ogni operatore viene contato il numero di classificatori da lui generato e la somma delle loro forze. I risultati sono dati in Fig.5.26 e in Fig.5.27, dalle quali si vede che l'operatore cover effector gioca un ruolo molto importante nella fase iniziale, mentre il crossover accresce la sua importanza solo più avanti. Questo comportamento è ragionevole perché si può immaginare che nella fase iniziale il sistema commetta molti errori[17] e che quindi il cover effector intervenga spesso, mentre più tardi, quando il numero di mosse errate diminuisce, possa emergere l'azione svolta dal crossover.

[17] Si ricordi che nel nostro sistema l'operatore di cover effector è attivato non solo in mancanza di messaggi verso l'ambiente, ma anche quando il messaggio causa una azione errata.

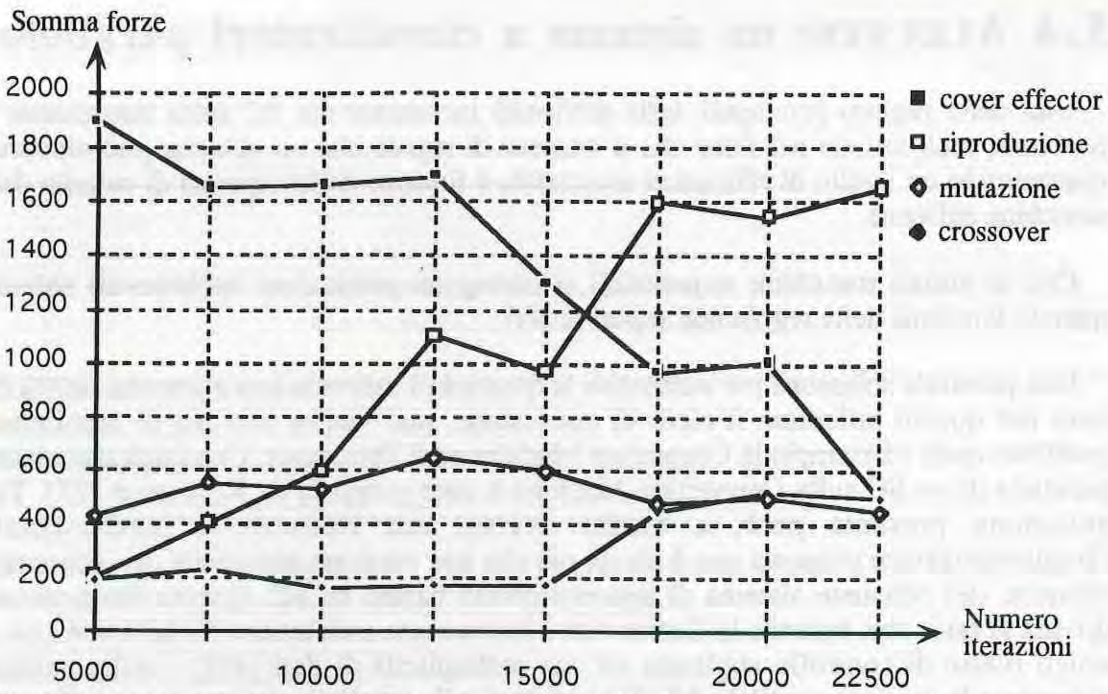


Fig.5.26 - Andamento nel tempo della somma delle forze dei classificatori generati dai diversi operatori

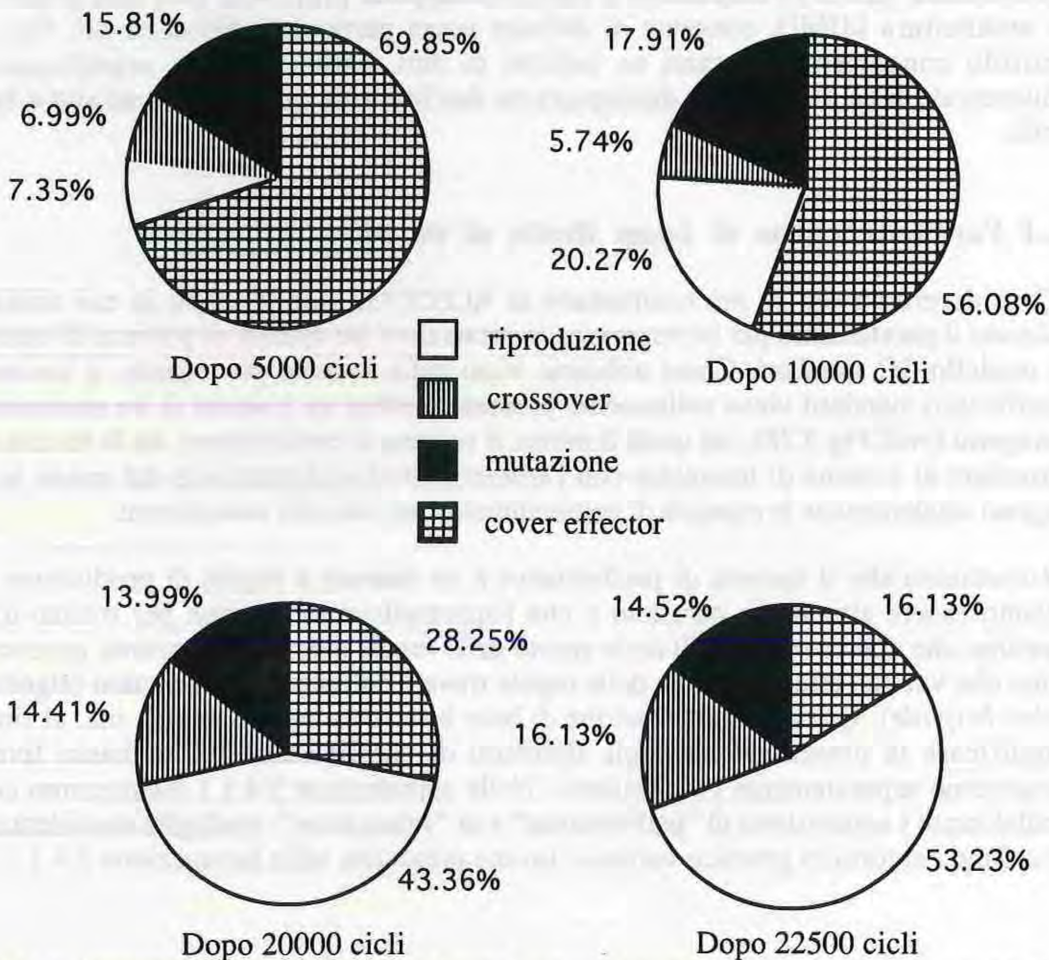


Fig.5.27 - Percentuale (della somma delle forze) dei classificatori generati dai diversi operatori in diversi istanti temporali

5.4 ALECSYS: un sistema a classificatori parallelo

Una delle ragioni principali delle difficoltà incontrate dai SC nella risoluzione di problemi reali risiede nel fatto che il numero di regole che un sistema può elaborare mantenendo un livello di efficienza accettabile è limitato dalle capacità di calcolo della macchina utilizzata.

Con le attuali macchine sequenziali si ottengono prestazioni ragionevoli soltanto quando l'insieme delle regole non superi le 300.

Una possibile soluzione per aumentare la quantità di informazioni elaborate, senza dovere per questo rallentare il ciclo di esecuzione, può venire dall'uso di architetture parallele, quali ad esempio la Connection Machine ed il Transputer. Una implementazione parallela di un SC sulla Connection Machine è stata proposta da Robertson [75]. Tale soluzione presenta però, a nostro avviso, una limitazione fondamentale: l'implementazione proposta non è niente più che una versione potenziata, ma comunque classica, del consueto sistema di apprendimento basato su SC. Questa limitazione è dovuta al fatto che, essendo la Connection Machine una architettura SIMD (cioè con un unico flusso di controllo applicato ad una molteplicità di dati [47]), risulta naturale progettare la versione parallela del SC basandosi sulla parallelizzazione dei dati. Essendo noi interessati allo sviluppo di un sistema più flessibile di quello standard, nel quale sia possibile inserire le caratteristiche architetturali viste nella sezione 5.2, abbiamo scelto di implementare ALECSYS utilizzando il sistema transputer [38], [43], [42] che, grazie alla sua architettura MIMD, consente di definire senza particolari difficoltà più flussi di controllo concorrenti, operanti su insiemi di dati distinti. Questa organizzazione architetturale ci ha permesso di distinguere tra due forme di parallelismo: ad alto e basso livello.

5.4.1 Parallelizzazione di basso livello di un SC

Considereremo ora la *microstruttura* di ALECSYS, cioè il modo in cui abbiamo utilizzato il parallelismo per incrementare le prestazioni (in termini di potenza di calcolo) del modello SC standard. Come abbiamo visto nella sezione precedente, il sistema a classificatori standard viene solitamente presentato come un insieme di tre sottosistemi interagenti (vedi Fig.5.28), dei quali il primo, il sistema di performance, ha la funzione di permettere al sistema di interagire con l'ambiente indipendentemente dal modo in cui vengono implementate le capacità di apprendimento nei due altri sottosistemi.

Ricordiamo che il sistema di performance è un sistema a regole di produzione che possono essere attivate in parallelo e che l'apprendimento avviene per mezzo di un algoritmo che ricerca regole utili nello spazio delle regole possibili (algoritmo genetico) e di uno che valuta l'effettiva utilità delle regole trovate dall'algoritmo genetico (algoritmo *bucket brigade*). Questa organizzazione di base è stata da noi mantenuta ma, al fine di semplificare la presentazione degli algoritmi di parallelizzazione di basso livello, discuteremo separatamente i vari sistemi. Nella sottosezione 5.4.1.1 mostreremo come parallelizzare i sottosistemi di "performance" e di "valutazione"; analoghe considerazioni riguardanti l'algoritmo genetico verranno invece presentate nella sottosezione 5.4.1.2.

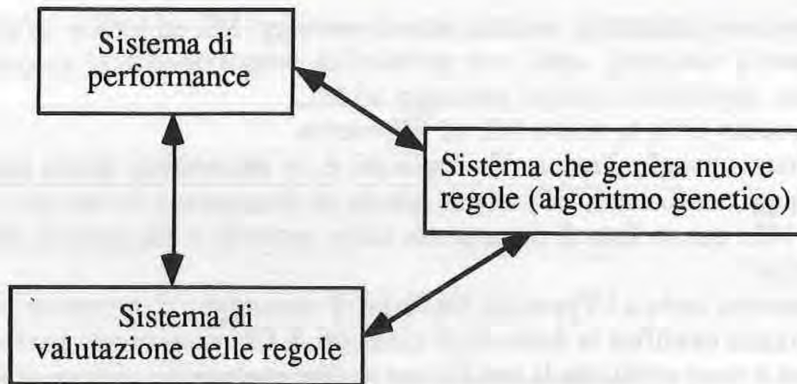


Fig.5.28 - Struttura di un SC classico

5.4.1.1 I sistemi di performance e di valutazione delle regole

Un ciclo base di esecuzione nella versione sequenziale di un SC può essere visto come il risultato dell'interazione tra due strutture dati: la lista dei messaggi (Message List, ML) e l'insieme dei classificatori (ClassiFier set, CF). Come si vede in Fig.5.29 l'elaborazione di tale ciclo può essere suddivisa tra i due processi concorrenti, MLprocess e CFprocess, interfacciati con l'ambiente mediante un processo di input (DTprocess, DeTector process) ed uno di output (EFprocess, EFfector process). Nel seguito riformuliamo l'algoritmo di performance unitamente a quello di valutazione delle regole mettendo in risalto gli aspetti legati alle comunicazioni.

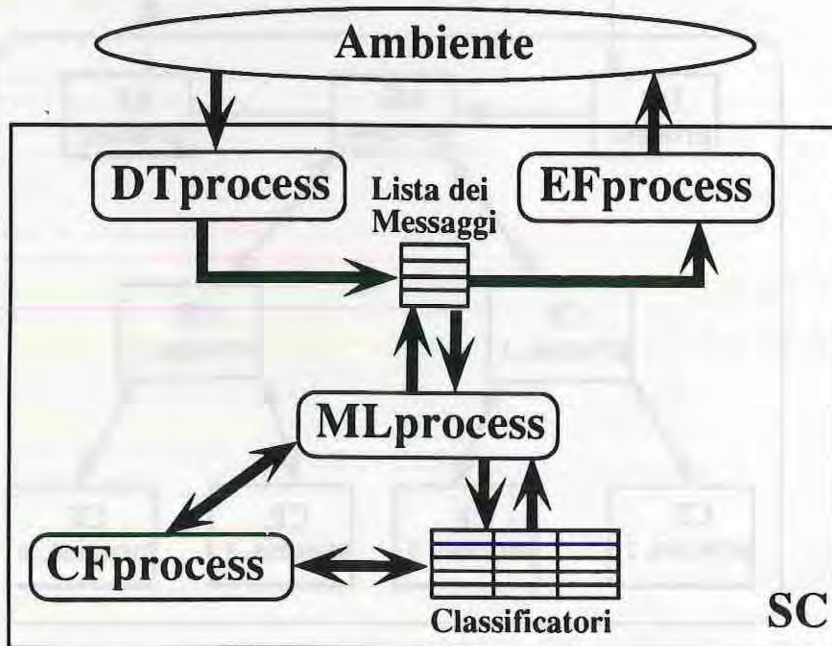


Fig.5.29 - SC visto come insieme di processi interagenti

1 - MLprocess riceve eventuali messaggi dall'ambiente tramite DTprocess e li pone nella lista di messaggi ML.
2 - MLprocess invia ML a CFprocess.
3 - CFprocess confronta ML e CF ("fase di matching") e calcola l'offerta fatta da ogni regola attivata.
4 - CFprocess invia a MLprocess la lista delle regole attivate.

5 - MLprocess cancella la vecchia lista di messaggi ML ed indice un'asta tra le regole attivate; i vincitori, scelti con probabilità proporzionale al valore della propria offerta, appendono i propri messaggi ad ML.
6 - MLprocess invia la nuova ML ad EFprocess.
7 - EFprocess sceglie l'azione da applicare e, se necessario, scarta da ML eventuali messaggi conflittuali; EFprocess calcola la ricompensa dovuta ad ogni messaggio nella ML; questa lista di ricompense viene restituita a MLprocess, assieme alla ML "residua".
8 - MLprocess invia a CFprocess l'insieme di messaggi e ricompense.
9 - CFprocess modifica le forze degli elementi di CF, assegnando le ricompense.
10 - Se non è stata verificata la condizione di fine esecuzione, ritorno al passo 1.

I passi 3 e 4 ("matching" e produzione di messaggi), possono essere eseguiti su ciascun classificatore in maniera indipendente. È perciò naturale spezzare CFprocess in un pacchetto di processi concorrenti CFprocess.1, ... , CFprocess.i, ... , CFprocess.n, ciascuno dei quali si occupa di una frazione n-esima di CF (vedi Fig.5.30). Quanto più elevato è il valore di n, tanto più intenso è il parallelismo. Quando n eguaglia la cardinalità di CF, ogni CFprocess.i gestisce un singolo classificatore: in questo caso abbiamo una versione di SC concorrente simile a quella implementata su Connection Machine da [75]. Nel nostro caso invece, dato che un'architettura parallela basata sui transputer è dotata di nodi con elevate capacità di calcolo, allochiamo circa 100-250 regole su ogni processore (nella sezione 5.4.2 daremo una motivazione sperimentale a questo intervallo di valori).

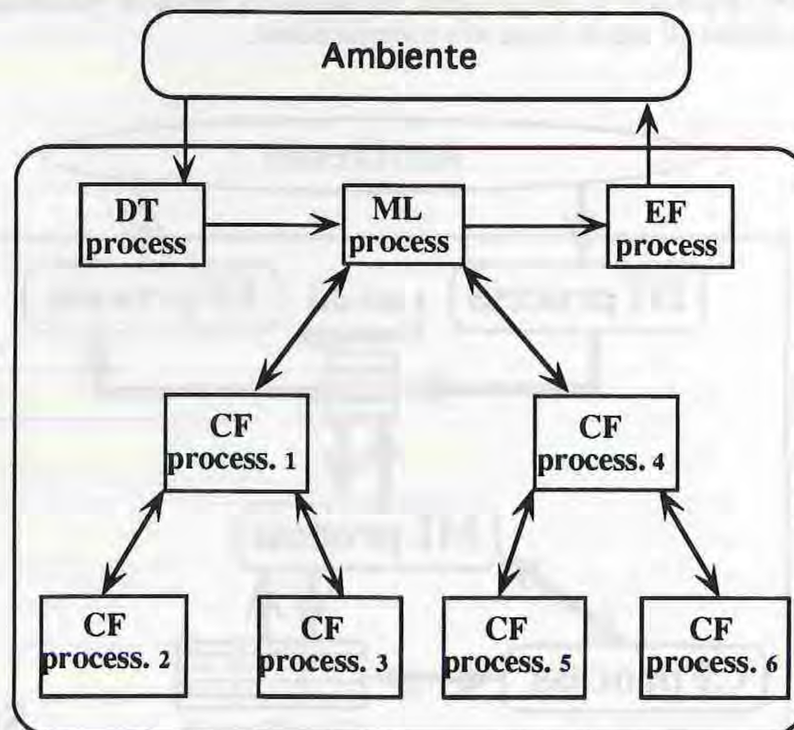


Fig.5.30 - Esempio di scomposizione di CF process in tanti processi CFprocess.i

Oltre ai su citati passi 3 e 4, ci sono molti altri passi dell'algoritmo che possono essere parallelizzati: se propaghiamo la lista dei messaggi ML da MLprocess all'i-esimo CFprocess e ritorno, possiamo ottenere una elaborazione parallela della fase di valutazione su ciascun processo CFprocess.i (il meccanismo di asta tra le regole attivate viene distribuito secondo una struttura gerarchica, e lo stesso approccio viene applicato alla ridistribuzione delle ricompense).

5.4.1.2 Il sistema di generazione delle regole

Vediamo ora come distribuire gli Algoritmi Genetici (Genetic Algorithms, GA) su una rete di processori transputer. Ad un primo processo, GAprocess, può essere affidata la selezione, all'interno degli elementi di CF, di quegli individui che dovranno essere replicati e di quelli che dovranno essere eliminati. Sarà invece compito di CFprocess (frammentato nel consueto pacchetto di CFprocess.i), applicare gli operatori genetici, in base alle decisioni di GAprocess: ciascun CFprocess.i si occuperà della frazione di popolazione CF assegnatagli.

MLprocess rimane inoperoso durante le operazioni di GAprocess, mentre GAprocess viene disattivato quando MLprocess è al lavoro. È quindi naturale allocare i processi MLprocess e GAprocess su di un singolo processore.

Una tipica forma dell'algoritmo genetico sequenziale applicato a problemi di apprendimento automatico può essere la seguente:

1 - All'interno di CF vengono selezionati due insiemi, uno composto di regole da replicare (classificatori *genitori*), e uno di regole da eliminare (posizioni che verranno occupate dai *figli* generati).
2 - I *genitori* vengono accoppiati a due a due.
3 - Ad ogni coppia viene applicato l'operatore crossover, che genera una nuova coppia di regole (*figli*).
4 - I *figli* subiscono una operazione di mutazione.

Il passo 1 può essere visto come un'asta, la quale può essere distribuita sulla rete di processori secondo un meccanismo di "raccolta e diffusione gerarchica" simile a quello utilizzato per propagare la lista dei messaggi. Il passo 2 (formazione delle coppie) è stato meno facile da parallelizzare in quanto richiede una unità direttiva centrale. Il passo 3 è stato il più arduo da parallelizzare, a causa dell'elevato numero di comunicazioni richieste, sia tra MLprocess e l'insieme di CFprocess.i, che fra gli stessi CFprocess.i. Il passo 4 è un tipico esempio di elaborazione dati su base locale, particolarmente adatto alla elaborazione concorrente.

L'algoritmo genetico parallelizzato risulta pertanto:

1 - Ciascun CFprocess.i seleziona, all'interno del sottoinsieme di CF affidatogli, m regole da replicare ed m da rimpiazzare (nota: m è un parametro di sistema).
2 - Ciascun CFprocess.i invia verso MLprocess alcuni dati riguardanti i classificatori selezionati al passo precedente, instaurando così un'asta gerarchica basata sui valori della forza; questo meccanismo si conclude nella scelta finale di 2m individui, selezionati tra la popolazione CF complessiva.
3 - Ad ogni CFprocess.i che contenga un *genitore*, MLprocess invia i seguenti dati:
    identificatore del *genitore*,
    identificatori dei due *figli*,
    punto di crossover (CrossOver Point, COP);
  allo stesso tempo, ad ogni CFprocess.i che contenga un *figlio*, MLprocess invia i seguenti dati:
    identificatore del *figlio*,
    identificatori dei due *genitori*,
    COP.
4 - Tutti i CFprocess.i che ospitano dei *genitori* nella propria frazione di CF inviano una copia del *genitore* al CFprocess.i che ospita il corrispondente classificatore

figlio; quest'ultimo processo dovrà poi applicare gli operatori genetici di crossover e mutazione.

5.4.2 Parallelizzazione di alto livello di un SC

Ciò che abbiamo presentato nella sezione precedente (al pari di quanto fatto da altri ricercatori, ad es. si vedano [14], [15], [75], [48], [74]), non è altro che una parallelizzazione del modello di SC classico, senza introdurre alcun significativo miglioramento, al di là della riduzione dei tempi medi di esecuzione. Questo approccio mostra la propria debolezza ogniqualvolta si applichi un SC a problemi multi obiettivo, come sembra sfortunatamente essere il caso nella maggior parte dei problemi reali. I SC classici trovano arduo il risolvere questo tipo di problemi perché tali sistemi non possiedono meccanismi espliciti per trattare insiemi distinti di regole, ciascuno dei quali dedicato alla risoluzione di un particolare obiettivo. Sembra che il modello SC richieda qualche ulteriore accorgimento per affrontare l'aumento di complessità causato dalla presenza contemporanea di molti obiettivi. Per queste ragioni un SC parallelizzato a basso livello, benché più veloce e perciò in grado di gestire popolazioni più ampie, risulta comunque di scarsa utilità pratica.

Inoltre insorgono problemi di scalabilità: l'aggiunta di un nodo alla rete transputer che implementa il SC provoca un aumento dei carichi di comunicazione in misura maggiore del corrispondente incremento di potenza di calcolo. Ne deriva quindi che aggiungere processori alla rete esistente risulta via via meno efficace.

Come abbiamo visto nella sezione 5.2, un modo migliore di trattare i problemi complessi potrebbe essere il codificarli come un gruppo di sottoproblemi più semplici, ciascuno dei quali affidato ad un SC diverso.

In ALECSYS la rete di processori viene partizionata in sottoinsiemi, ciascuno dei quali avente propria dimensione e topologia. Ad ogni sottoinsieme viene allocato un singolo SC, eventualmente distribuito secondo una parallelizzazione di basso livello (vedi Fig.5.31).

Ognuno di questi SC impara a risolvere uno specifico sottoproblema, in base agli ingressi che riceve: ciascun SC percepisce l'ambiente circostante attraverso i propri detettori mentre l'interfaccia di uscita è ovviamente la medesima per tutti i SC.

Il sistema parallelo presentato in questa sezione, ALECSYS, è stato da noi implementato con l'unica finalità di fornirci di uno strumento flessibile e potente per poter proseguire la nostra ricerca riguardante il controllo di un robot mobile. Pertanto non era nostro obiettivo lo studio di quelle caratteristiche dell'algoritmo che potrebbero migliorarne la performance da un punto di vista strettamente legato alle metodologie di progettazione parallela. Per alcune decisioni progettuali (ad esempio che topologia di collegamento utilizzare) ci siamo avvalsi dell'esperienza del gruppo di Pisa [14], [15]. L'unico parametro che abbiamo indagato è stato il numero ottimo di classificatori da mettere in ogni nodo transputer.

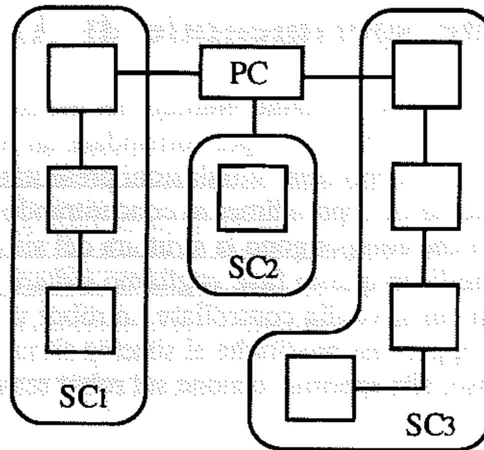


Fig.5.31 - Parallelizzazione di alto e basso livello: una rete transputer nella quale tre SC concorrenti (parallelismo di alto livello) vengono mappati rispettivamente su tre, uno e quattro nodi (parallelismo di basso livello).

In Fig.5.32 sono riportati i risultati ottenuti nel caso di singolo sistema SC parallelizzato con parallelismo di basso livello ed applicato al caso in cui un semplice robot deve imparare a seguire una sorgente luminosa (vedi sezione 5.5): i risultati dell'esperimento indicano che le prestazioni del sistema raggiungono il massimo utilizzando 200 classificatori su ciascuno dei 9 processori impiegati. Questo sembra confermare, relativamente al caso di un SC parallelo, l'ipotesi proposta da Goldberg per il caso sequenziale riguardo l'esistenza di una dimensione ottimale per la popolazione di classificatori [55].

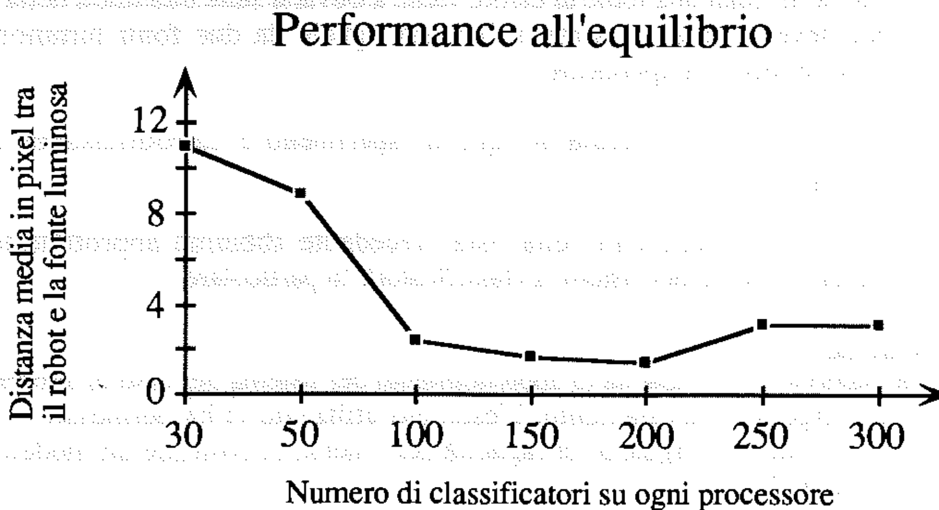


Fig.5.32 - Prestazioni a regime in funzione del numero di classificatori; la misura delle prestazioni è data dalla distanza media (espressa in pixel) tra il robot e la fonte luminosa (media degli ultimi 5000 cicli in una simulazione di 20000 cicli)

5.5 Valutazione sperimentale di ALECSYS

In questa sezione presentiamo i risultati ottenuti per mezzo di una serie di esperimenti effettuati utilizzando il sistema ALECSYS per comandare un robot simulato che si muove in ambiente bidimensionale. Scopo delle simulazioni presentate in questa sezione è quello di mettere a punto il sistema di apprendimento concentrando la propria attenzione sulla progettazione dei moduli che compongono la struttura del sistema stesso e sulle modalità con le quali interconnetterli (in questa fase vogliamo evitare tutti i problemi pratici che nascono nel momento in cui si voglia controllare un robot vero). Una volta compreso il funzionamento e fatte le opportune modifiche al sistema simulato si potrà passare, con più alta probabilità di successo, alla sperimentazione sul robot reale.

5.5.1 Esperimenti in un ambiente bidimensionale

In questa prima fase sperimentale abbiamo limitato la nostra attenzione ad un sistema di apprendimento composto da massimo tre sistemi a classificatori, testando le capacità di apprendimento del robot simulato su un insieme di compiti di difficoltà crescente. Gli esperimenti effettuati, in ordine di difficoltà, sono i seguenti:

(a) il robot deve imparare a seguire una luce che si muove in un piano bidimensionale (lo schermo del computer);
(b) il robot (che in questo caso è una specie di camaleonte artificiale) deve imparare a seguire la luce e contemporaneamente deve imparare a cambiare il proprio colore in accordo con quello dello sfondo;
(c) il robot deve imparare a seguire la luce e ad evitare delle fonti di calore (che rappresentano degli oggetti pericolosi), imparando quindi a gestire le situazioni di conflitto nelle quali una fonte di calore viene a trovarsi sulla traiettoria della luce;
(d) il robot deve imparare a stare a distanza uguale da due fonti luminose che si muovono in modo indipendente.

L'architettura software utilizzata per questi esperimenti è un sottoinsieme di quella presentata in Fig.5.2.

Per ognuno degli esperimenti della lista precedente abbiamo approfondito aspetti diversi del funzionamento dei sistemi a classificatori. In particolare:

Esperimento (a)

Valutazione delle capacità di apprendimento del sistema nel caso di apprendimento di un singolo comportamento. L'esempio utilizzato ci ha permesso di dare una prima valutazione riguardo la capacità del sistema di costruire un modello interno del mondo[18].

Esperimento (b)

(b1) Valutazione del sistema di parallelizzazione detto *parallelo* rispetto a quello detto *distribuito*. Nel sistema parallelo entrambi i comportamenti (inseguimento della luce e cambiamento del colore) devono essere appresi da un singolo sistema a classificatori (eventualmente parallelizzato, con parallelismo di basso livello). Nel sistema distribuito ogni comportamento è appreso da un diverso sistema a classificatori (eventualmente parallelizzato, con parallelismo di basso livello). Inoltre, in questo secondo caso, abbiamo valutato due politiche di premio

[18] Vedi sezione 5.3.2.6.

alternative: nella prima, chiamata di *premio singolo*, lo stesso premio viene dato ad entrambi i sistemi a classificatori (indipendentemente da quale sia stato il responsabile principale dell'azione); nella seconda, detta di *premio doppio*, il premio è specifico per il sistema a classificatori che ha prodotto l'azione.
(b2) Abbiamo inoltre valutato l'utilità dei premi negativi (punizioni) in ognuno dei tre tipi di architettura visti sopra.

Esperimento (c)

Valutazione delle capacità di apprendimento del coordinamento. In questo caso oltre all'utilizzo di un sistema a classificatori per apprendere ad inseguire una sorgente di luce e di un sistema a classificatori per apprendere ad evitare oggetti pericolosi, abbiamo utilizzato un terzo sistema per valutare la capacità di apprendere il meta-compito di coordinamento nel caso di situazioni di conflitto, nelle quali il robot deve imparare ad aggirare gli oggetti pericolosi pur continuando a seguire la sorgente luminosa (al compito "evitare gli oggetti pericolosi" viene attribuita un importanza relativa maggiore che al compito "seguire la luce"). In questo esperimento ci siamo limitati all'osservazione qualitativa dei risultati, lasciando all'esperimento (d) l'analisi quantitativa.

Esperimento (d)

Valutazione del compito di apprendimento del comportamento di coordinamento nelle diverse architetture: parallela, distribuita vettoriale e distribuita gerarchica[19]. Obiettivo dell'apprendimento è imparare un insieme di regole che permettano al robot di perseguire contemporaneamente due obiettivi contrastanti: inseguire due sorgenti di luce che si muovono in modo indipendente. Abbiamo confrontato il sistema che apprende con tre classificatori organizzati in struttura gerarchica con un sistema gerarchico vettoriale e con un sistema parallelo.

5.5.2 Caratteristiche generali del robot simulato

Prima di iniziare la descrizione dei risultati, definiamo alcune caratteristiche generali riguardanti sensori ed attuatori del robot. Il robot ha tre tipi di sensori: sensore di luce, sensore di calore e sensore di colore. I primi due sono sensibili alla posizione (non alla intensità) della luce o del calore. Il robot ha quattro sensori di luce e quattro di calore posizionati sui quattro lati, in modo tale da dividere il piano in quattro semipiani parzialmente ricoprentesi. I sensori di luce sono uguali a quelli descritti nella sottosezione 5.3.2, e i sensori di calore sono organizzati in modo analogo.

Il sensore di colore è sensibile solo al colore dello sfondo nella zona su cui poggia il robot nell'istante considerato (il numero di differenti colori cui è sensibile è 8).

Come per il robot presentato nella sottosezione 5.3.2. le possibili velocità di movimento sono 3: semplice, doppia e nulla, nelle otto direzioni di Fig.5.12.

Il robot è inoltre capace di cambiare il suo colore.

[19] L'architettura distribuita gerarchica è in questo caso composta da tre moduli: due SC per i due comportamenti di base e un SC per il coordinamento. Nell'architettura distribuita vettoriale il coordinamento è invece un modulo che esegue la somma delle due azioni proposte dagli SC di base. L'architettura viene detta vettoriale perché le azioni proposte dagli SC di base sono movimenti (vettori) e l'azione somma è perciò la somma vettoriale delle azioni proposte.

Le regole hanno il formato di Fig.5.33, dove il numero di bit e il relativo significato varia nei diversi esperimenti.

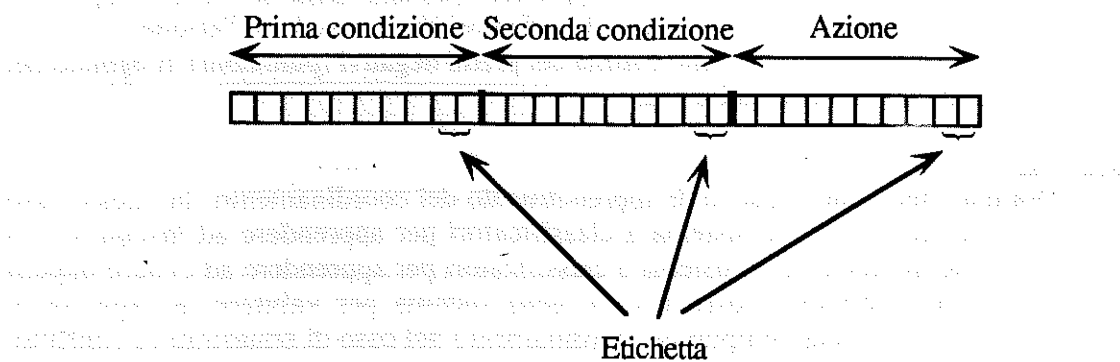


Fig.5.33 - Formato delle regole

In ogni esperimento due bit sono dedicati all'etichettatura (tag) del messaggio:

10 significa che il messaggio arriva dai detettori,
01 significa che il messaggio va agli effettori,
00 significa che il messaggio è stato generato all'istante precedente da una delle regole (è cioè un messaggio interno),
11 significa che il messaggio è responsabile per l'ultima azione effettuata (cioè è stato spedito agli effettori all'istante precedente ed ha vinto la competizione con eventuali altri messaggi per gli effettori, determinando pertanto l'azione effettivamente svolta dal robot).

5.5.3 Esperimento (a): inseguimento di una luce

In questo esperimento abbiamo voluto valutare la capacità del sistema di apprendere a seguire una sorgente luminosa. La base di regole che controlla il robot viene generata all'istante iniziale in modo del tutto casuale e viene poi sottoposta a modifica da parte degli algoritmi di distribuzione del credito e di creazione di nuove regole (algoritmo genetico). In Fig.5.34 è riportato un esempio del problema. Ad ogni ciclo il robot percepisce la posizione della luce per mezzo dei sensori e propone una direzione di movimento. Se la direzione di movimento fa diminuire la distanza tra il robot e la sorgente luminosa il robot viene premiato, se la distanza non varia il robot riceve ugualmente un premio (ma più basso: questo premio serve a far sì che non vi siano comportamenti strani quando il robot ha ormai raggiunto la luce; in questo caso infatti il comportamento ottimo è quello di continuare a seguire la luce, stando ad una distanza idealmente nulla), altrimenti viene punito.

Il sistema apprende abbastanza velocemente (poche decine di secondi sul nostro sistema usando 3 transputer[20]). Come si vede dalla Fig.5.35, dopo una fase iniziale di circa 800÷1000 cicli (un ciclo è l'intervallo che intercorre tra due mosse successive del robot), la performance raggiunta, misurata in distanza del robot dalla fonte di luce, è molto buona.

---

[20] Il sistema transputer utilizzato è una scheda FAST9 della Quintek; ogni scheda è dotata di 9 transputer T800 a 25 MHz con 1 Mb di RAM ciascuno.

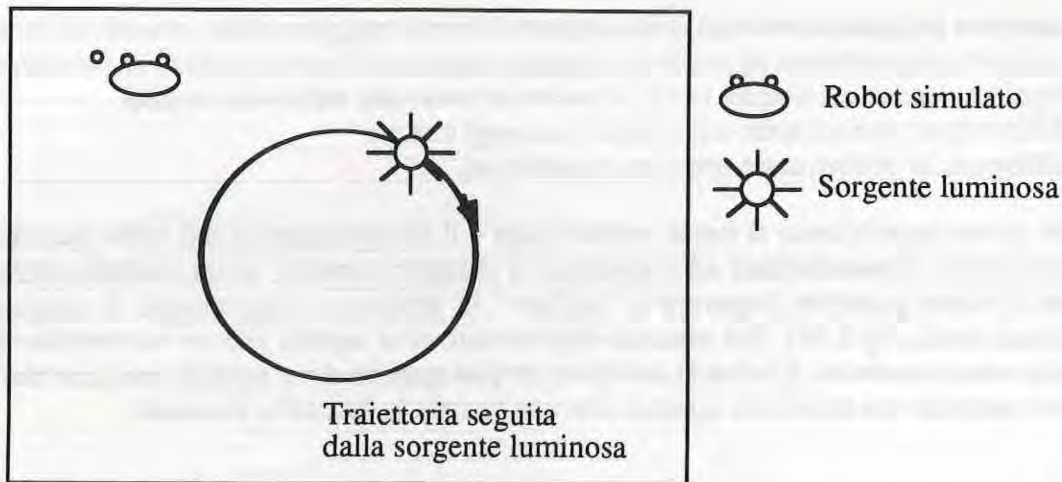


Fig.5.34 - Ambiente di simulazione nel quale il robot deve imparare a seguire un segnale luminoso

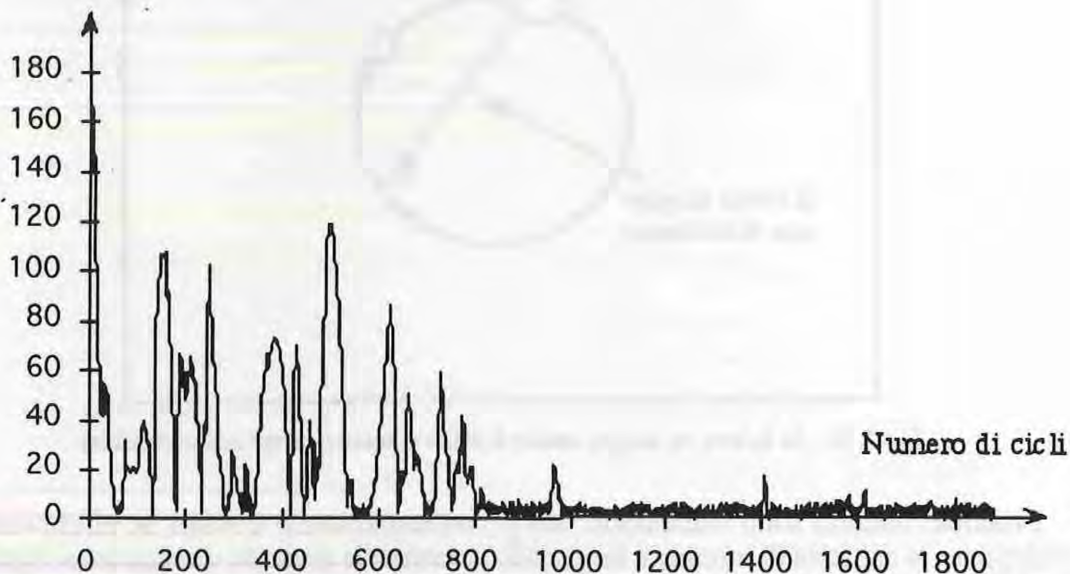


Fig.5.35 - Distanza - in pixel - del robot dalla sorgente di luce

In vista dell'esperimento (c1), abbiamo posto il robot in un ambiente contenente alcune fonti di calore. Il robot deve imparare a muoversi in questo ambiente mantenendosi lontano dalle sorgenti di calore. I premi e le punizioni vengono dati secondo una logica simmetrica a quella usata nell'esperimento con la luce: quando il robot si avvicina alla sorgente di calore viene punito, mentre viene premiato quando vi si allontana. Una importante differenza con il sistema precedente consiste nel fatto che il robot riceve premi o punizioni solo quando si trova al di sotto della distanza di percezione della sorgente di calore. Ciò fa sì che il tempo richiesto per apprendere sia maggiore (infatti parte del tempo speso dal robot per muoversi non viene utilizzato dall'apprendimento perché il robot non percepisce calore). Nondimeno le simulazioni hanno mostrato come anche in questo caso il robot, pur impiegando più tempo, impari il compito in modo corretto.

Abbiamo anche tentato di valutare se il robot fosse o meno in grado di costruire un modello interno del mondo per mezzo dei seguenti esperimenti:

- ambiente nel quale la velocità della sorgente di luce è maggiore della velocità del robot;
- confronto fra ambiente nel quale la sorgente luminosa si muove secondo una traiettoria regolare rispetto e ambiente in cui si muove secondo una traiettoria casuale.
- differenza in performance con o senza messaggi interni;
- differenza in performance con o senza punizioni;

Nel primo esperimento si vuole vedere quale è il comportamento del robot quando è "fisicamente" impossibilitato ad apprendere il compito corretto; se un modello interno esiste il robot potrebbe imparare a "tagliare" la strada per raggiungere la sorgente luminosa (vedi Fig.5.36). Nel secondo esperimento ci si aspetta che, se un modello del mondo viene costruito, il robot si comporti meglio quando deve seguire una luce che si muove secondo una traiettoria regolare che non quando la traiettoria è casuale.

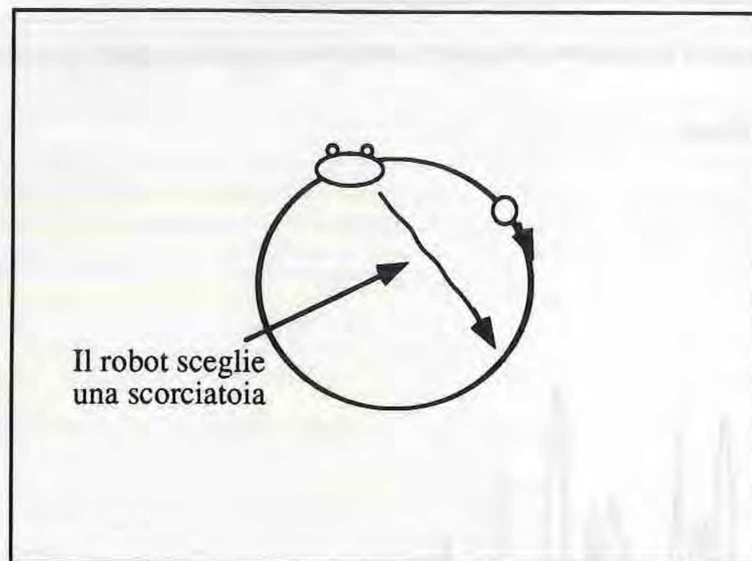


Fig.5.36 - Se la luce va troppo veloce il robot si muove lungo una scorciatoia

I risultati ottenuti sono interessanti: nel primo esperimento il robot ha effettivamente sviluppato la capacità di muoversi lungo delle scorciatoie in modo da riuscire a seguire la luce sebbene questa si muova ad una velocità superiore alla sua.

I risultati del secondo, terzo e quarto esperimento sono riportati nelle tre tabelle 5.4, 5.5 e 5.6. In ogni tabella sono riportati, per ogni combinazione di politica di premio e di lunghezza di MLI, i valori di performance ottenuti dopo centodiecimila cicli e, fra parentesi, il numero di cicli necessario per raggiungere il livello di performance 0.8. Gli esperimenti indicano che la performance del robot non è influenzata dal modo in cui si muove la sorgente luminosa. I messaggi interni risultano essere, in questo particolare compito, dannosi aumentando notevolmente, in tutti i casi, il tempo necessario per raggiungere buoni livelli di performance. Da ultimo si può vedere che l'utilizzo di punizioni accelera l'apprendimento e contemporaneamente permette il raggiungimento di livelli di performance più elevati.

Tab.5.4: Risultati esperimenti con luce che si muove in modo casuale.

	MLI = 10	MLI = 0
Solo premi	0.82 (≈ 100000)	0.9 (≈ 40000)
Premi e punizioni	0.96 (≈ 11000)	0.97 (≈ 2000)

Tab.5.5: Risultati esperimenti con luce che si muove con traiettoria circolare.

	MLI = 10	MLI = 0
Solo premi	0.83 (≈ 90000)	0.86 (≈ 60000)
Premi e punizioni	0.96 (≈ 10000)	0.97 (≈ 2000)

Tab.5.6: Risultati esperimenti con luce che si muove con traiettoria rettilinea.

	MLI = 10	MLI = 0
Solo premi	0.82 (≈ 100000)	0.88 (≈ 35000)
Premi e punizioni	0.92 (≈ 12000)	0.94 (≈ 2000)

I risultati di questi esperimenti indicano che per questo compito il sistema non sviluppa un modello interno del mondo. Ciò è ragionevole quando si consideri la natura tipicamente "stimolo-risposta" del compito da apprendere. I risultati del primo esperimento non devono fuorviarci in quanto, sebbene consistenti con la presenza di un modello interno, possono essere spiegati anche altrimenti. Ad una analisi attenta infatti, risulta che l'effetto osservato può essere spiegato utilizzando solo regole di tipo stimolo risposta.

5.5.4 Esperimento (b): un robot *camaleonte*

Come già detto, questo esperimento vuole valutare la differenza in performance tra un sistema *parallelo* ed uno *distribuito*. Nel nostro esempio il compito che deve essere appreso dal *robot-camaleonte* è imparare a seguire un segnale luminoso e contemporaneamente a cambiare colore in modo tale da avere lo stesso colore del terreno su cui si sta muovendo (vedi Fig.5.37). Il compito complessivo è in questo caso composto da due compiti che risultano essere tra di loro completamente indipendenti. Possiamo pertanto pensare di utilizzare un sistema dove entrambi i compiti vengono appresi da un singolo sistema a classificatori (è questo il caso di *sistema parallelo* dove il parallelismo utilizzato è quello detto di basso livello) oppure uno dove ogni compito è appreso da un SC specifico (è questo il caso di *sistema distribuito* dove il parallelismo utilizzato è quello detto di alto livello[21]). È chiaro che nel caso di sistema parallelo lo spazio delle possibili regole è molto più grande (perché una regola avrà condizioni con un maggior numero di bit a causa della maggiore informazione di cui tenere conto): ci aspettiamo perciò che l'apprendimento, se c'è, sia più lento.

---

[21] Rimane sempre possibile parallelizzare ognuno dei SC che compongono il sistema utilizzando parallelismo di basso livello.

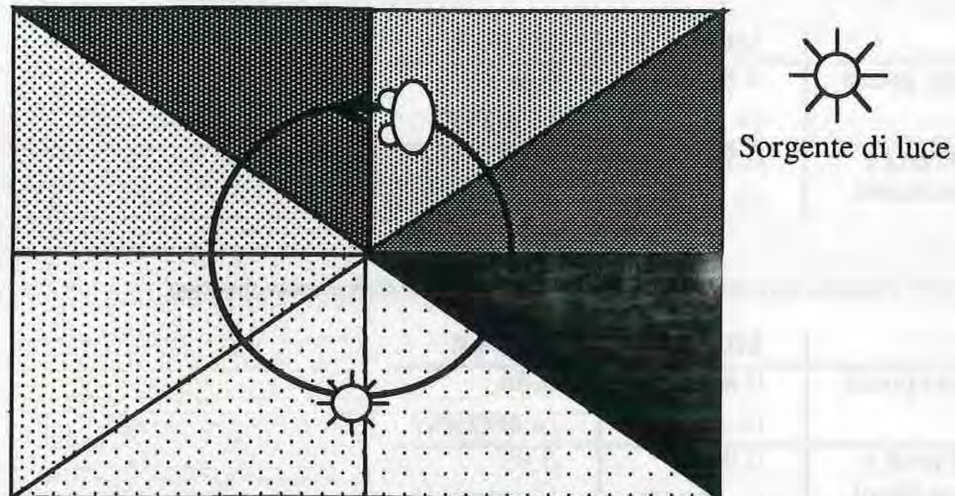


Fig.5.37 - Ambiente di apprendimento per il robot-camaleonte: il terreno è suddiviso in otto settori triangolari di colore diverso; la sorgente di luce segue un'orbita circolare

Nel caso di sistema distribuito abbiamo anche valutato due metodologie secondo le quali dare premi o punizioni: nella prima, detta di *premio unico*, un singolo premio viene dato ad entrambi i SC senza distinguere chi fosse l'artefice della azione per cui il premio è stato generato. Nella seconda, detta di *premio doppio*, il premio viene dato al sistema che ha effettivamente generato l'azione. Pertanto se ad esempio il robot segue correttamente la luce ma sbaglia il colore, con il primo metodo entrambi i SC prenderanno lo stesso premio (che sarà meno elevato che non nel caso che entrambe le azioni di inseguimento luce e cambio colore fossero state corrette), mentre con il secondo metodo il SC che si occupa di seguire la luce prenderà un premio mentre quello che si occupa del colore prenderà una punizione.

La politica del premio doppio dovrebbe essere la migliore in un caso come quello considerato in questo esperimento (nel quale i diversi compiti sono completamente indipendenti), ma potrebbe non esserlo in casi più complicati, nei quali l'azione effettuata è il risultato della composizione di più azioni proposte. Inoltre riteniamo sia in generale interessante valutare le capacità di apprendimento nel caso in cui il solo criterio di valutazione sia quello del grado di raggiungimento del comportamento globale desiderato.

Per aiutare il lettore a comprendere le caratteristiche dei sistemi studiati in questa sezione, riportiamo nel seguito, prima della esposizione dei risultati sperimentali, la struttura delle regole e il significato dei messaggi che arrivano dai detettori e che vanno agli effettori per i due sistemi parallelo e distribuito.

5.5.4.1 Descrizione delle componenti del sistema parallelo

In Fig.5.38 è riportato il formato delle regole e di seguito il significato delle loro parti nel caso del sistema parallelo.

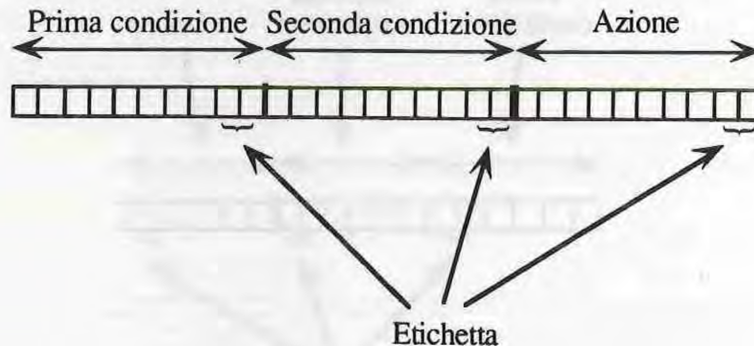


Fig.5.38 - Formato delle regole nel sistema parallelo

A seconda dei bit di etichettatura il significato dei messaggi è differente; per i messaggi dai detettori (etichetta 10) il significato è il seguente:

- primo e secondo bit: etichetta;
- bit 3-6: detezione luce;
- bit 7-9: detezione colore;
- bit 10: a disposizione.

Per i messaggi agli effettori (etichetta 01):

- primo e secondo bit: etichetta;
- bit 3-5: tre bit per determinare le otto possibili direzioni di movimento;
- bit 6: movimento; può essere attivo o no. Nel primo caso il robot si muove nella direzione definita dai bit 3-5, nel secondo sta fermo;
- bit 7-9: tre bit per determinare gli otto possibili colori;
- bit 10: cambiamento colore; può essere attivo o no. Nel primo caso il robot cambia colore secondo quanto definito dai bit 7-9, nel secondo non cambia colore.

Per i messaggi interni (etichetta 00) non è possibile dare una interpretazione del significato dei bit mentre i messaggi con etichetta 11 hanno lo stesso significato dei messaggi agli effettori.

5.5.4.2 Descrizione delle componenti del sistema distribuito

Nel sistema distribuito le condizioni delle regole, e quindi i messaggi, sono più corti (vedi Fig.5.39), in quanto devono rappresentare solo un sottoinsieme dell'informazione rappresentata nel sistema parallelo. Le regole hanno il seguente formato:

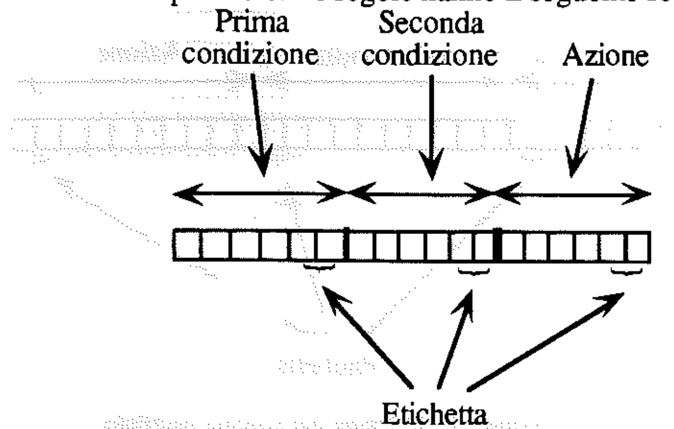


Fig.5.39 - Formato delle regole nel sistema distribuito

Di nuovo il significato dei bit dipende dalla etichetta. Inoltre dipende dal SC cui la regola appartiene. Nel caso di appartenenza al SC che si occupa dell'inseguimento della luce si ha:

per i messaggi dai detettori (etichetta 10)

- primo e secondo bit: etichetta;
- bit 3-6: detezione luce;

per i messaggi agli effettori (etichetta 01):

- primo e secondo bit: etichetta;
- bit 3-5: tre bit per determinare le otto possibili direzioni di movimento;
- bit 6: movimento; può essere attivo o no. Nel primo caso il robot si muove nella direzione definita dai bit 3-5, nel secondo sta fermo.

Nel caso di appartenenza al SC che si occupa del cambio di colore si ha:

per i messaggi dai detettori (etichetta 10)

- primo e secondo bit: etichetta;
- bit 3-5: detezione colore;
- bit 6: a disposizione.

per i messaggi agli effettori (etichetta 01):

- primo e secondo bit: etichetta;
- bit 3-5: tre bit per determinare gli otto possibili colori;
- bit 6: cambio di colore; può essere attivo o no. Nel primo caso il robot cambia colore secondo quanto definito dai bit 3-5, nel secondo non cambia colore.

Per i messaggi interni (etichetta 00) vale per entrambi i sistemi quanto detto nel caso del sistema parallelo.

5.5.4.3 Risultati delle simulazioni

Gli esperimenti effettuati, come detto, sono volti a valutare i seguenti aspetti:

(b1) differenza nelle prestazioni dei sistemi parallelo, distribuito con premio unico e distribuito con premio doppio;
(b2) differenza nelle prestazioni tra il caso in cui si utilizzano solo premi e quello in cui si utilizzano premi e punizioni.

In entrambi gli esperimenti abbiamo utilizzato la modalità senza messaggi interni (MLI=0).

Esperimento (b1)

I risultati di questo esperimento hanno mostrato che, come ci si aspettava, l'architettura migliore per questo tipo di compito è quella distribuita con premio doppio, anche se la differenza con l'architettura distribuita con premio singolo è molto lieve. La tabella tabella 5.7 riporta, per ogni tipo di sistema, la performance per i due comportamenti di cambiamento colore e di inseguimento della luce separatamente, e quella della loro composizione. La performance è misurata come numero medio di mosse corrette rispetto alle mosse totali negli ultimi 1000 cicli. Nella valutazione del compito completo "Colore e Luce" una mossa è stata giudicata corretta solo se erano corretti sia il cambiamento di colore che il movimento verso la luce (i due comportamenti sono cioè composti con la funzione logica And). Il numero totale di cicli effettuato è 80000. Come si può osservare, per ogni tipo di comportamento la performance migliora passando da sistema parallelo a sistema distribuito con premio unico per raggiungere il valore massimo con sistema distribuito con premio doppio.

Tab.5.7 - Confronto tra i tre tipi di architettura parallela proposti. Nelle colonne è riportata la performance (si usano premi e punizioni, vedi esperimento (b2)).

	Colore	Luce	Colore e Luce
Sistema parallelo	0.27	0.34	0.18
Sistema distribuito con premio unico	0.93	0.95	0.87
Sistema distribuito con premio doppio	0.95	0.97	0.92

Esperimento (b2)

In questo esperimento abbiamo confrontato le seguenti politiche di premio:

- solo premi (il premio vale 10 per ogni mossa corretta: il premio massimo sarà quindi 20, il minimo 0),
- premi e punizioni (il premio vale 10 per ogni mossa corretta, la punizione vale -15 per ogni mossa errata: in questo caso quindi i valori ritornati al camaleonte possono essere 20, -5, -30),

per ognuno dei tre tipi di architettura considerati. I risultati per il sistema parallelo, riassunti in tabella 5.8, ci dicono che sia nel caso di utilizzo di solo premi che di premi e punizioni si ha un basso livello di apprendimento per entrambi i compiti.

Tab.5.8 - Confronto tra diverse tipologie di premiazione - Architettura parallela - Performance media delle ultime 1000 iterazioni dopo 80000 iterazioni.

	Colore	Luce	Colore e Luce
Sistema parallelo premi-punizioni	0.27	0.34	0.18
Sistema parallelo solo premi	0.25	0.27	0.13

Le cose migliorano nel sistema distribuito con premio unico: utilizzando premi e punizioni si ha un netto miglioramento della performance (vedi Tab.5.9). Nel sistema distribuito con premio doppio si ha un netto miglioramento della performance nel caso di soli premi (vedi Tab.5.10), mentre questa risulta essere la migliore in assoluto nel caso di utilizzo di premi e punizioni.

Tab.5.9 - Confronto tra diverse tipologie di premiazione - Architettura distribuita con premio unico - Performance media delle ultime 1000 iterazioni dopo 80000 iterazioni.

	Colore	Luce	Colore e Luce
Sistema distribuito premi-punizioni	0.93	0.95	0.87
Sistema distribuito solo premi	0.40	0.50	0.35

Tab.5.10 - Confronto tra diverse tipologie di premiazione - Architettura distribuita con premio doppio - Performance media delle ultime 1000 iterazioni dopo 80000 iterazioni.

	Colore	Luce	Colore e Luce
Sistema distribuito premi-punizioni	0.95	0.97	0.92
Sistema distribuito solo premi	0.85	0.87	0.76

5.5.5 Esperimento (c): seguire una luce evitando oggetti pericolosi

In questo esperimento abbiamo posto il robot in un ambiente in cui si presentano situazioni contraddittorie, per cui uno degli obiettivi (ad esempio l'inseguire la luce) contrasta con l'altro (evitare la sorgente di calore). Questo accade ogni qualvolta il robot si trova in una situazione come quella riportata in Fig.5.40.

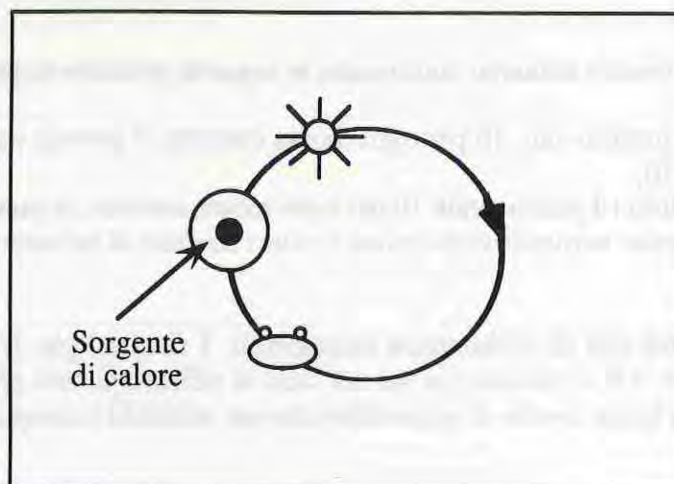


Fig.5.40 - Ambiente di apprendimento per il robot nel caso di due comportamenti interagenti

Il formato delle regole è del tutto equivalente a quello presentato negli esperimenti precedenti fatte le opportune modifiche per tenere conto delle caratteristiche specifiche del nuovo compito.

Il sistema che apprende questo compito è del tipo distribuito con tre SC (vedi Fig.5.41): uno per l'inseguimento della luce (SC-luce), uno per imparare ad evitare il calore (SC-calore) e il terzo per apprendere il coordinamento (SC-coordinamento).

Fig.5.41 - Sistema gerarchico di coordinamento a due livelli.

Ogni volta che SC-luce o SC-calore impostano un messaggio per gli effettori, essi lo spediscono anche a SC-coordinamento: questo interviene solo quando riceve un messaggio da entrambi i sistemi. In questi casi SC-coordinamento prende il sopravvento e l'azione effèttuata non è più decisa dai due sistemi di più basso livello, bensì dal coordinatore. In questo caso SC-coordinamento riceve direttamente un premio che utilizza per apprendere la giusta politica di coordinamento, cioè ad aggirare la fonte di calore cercando di limitare l'allontanamento dalla sorgente luminosa.

Gli esperimenti effettuati con questa architettura sono stati molto soddisfacenti: il comportamento osservato è quello desiderato, cioè il robot impara a seguire la luce fintanto che non si avvicina a zone in cui una sorgente di calore gli ostacola la strada (vedi Fig.5.42). A questo punto il comportamento scelto è uno dei seguenti: (i) il robot gira intorno alla sorgente di calore (comportamenti a e b in figura 5.42) e poi riprende a seguire la luce, (ii) il robot si ferma per un po' davanti alla sorgente di calore, fintanto che la luce non si è allontanata a sufficienza e poi riprende a seguirla (comportamento c in figura 5.42).

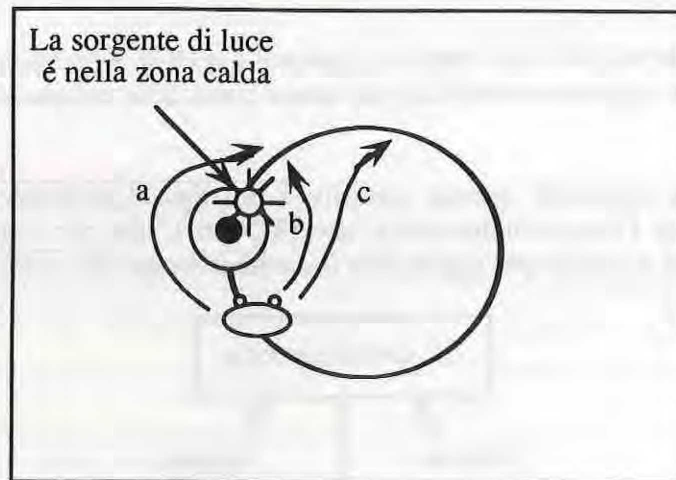


Fig.5.42 - Interazione tra comportamento di inseguimento della luce e comportamento di allontanamento dal calore

5.5.6 Esperimento (d): seguire due sorgenti luminose (stando alla minor distanza possibile dalle due)

Questo esperimento pone al robot un problema simile a quello presentato nella sottosezione precedente: obiettivo dell'apprendimento è imparare un insieme di regole che permettano al robot di perseguire contemporaneamente due obiettivi contrastanti. A differenza del caso precedente però, qui il robot ha sempre accesso a tutta l'informazione (nel caso luce-calore il robot percepiva la presenza di uno dei due stimoli, il calore, solo quando era abbastanza vicino). Inoltre è più facile misurare il livello di performance, dato che il raggiungimento dell'obiettivo coincide con lo stare su un punto qualunque dell'asse del segmento che unisce le due luci in movimento[22].

L'esperimento più significativo fatto con questo ambiente è stato il confrontare fra loro i seguenti diversi tipi di architettura:

- distribuita gerarchica,
- distribuita vettoriale,
- parallela.

Il compito di inseguimento è in questo caso reso più difficile dal fatto che il SC riceve informazioni non solo riguardo la posizione delle due luci, ma anche riguardo la loro distanza. Un messaggio ambientale da uno dei sensori sarà quindi composto da 4 bit per la detezione della direzione di provenienza della luce, 6 bit per la distanza della luce e due bit di tag.

L'architettura distribuita gerarchica (vedi Fig.5.41) prevede due SC per le due luci da seguire (ognuno è specializzato nell'inseguimento di una delle due luci) ed un SC per coordinare le azioni proposte. L'architettura distribuita vettoriale è uguale a quella distribuita gerarchica tranne che per il sistema coordinatore che in questo caso è sostituito

[22] Nel caso precedente del problema luce-calore la difficoltà nel misurare il livello di performance era dovuta al fatto che i due obiettivi erano effettivamente contrastanti e, quando in presenza di calore, l'obiettivo evitare la sorgente calda doveva assumere un importanza relativa maggiore dell'obiettivo seguire la luce. Per arrivare ad una misura della performance avremmo comunque potuto definire una funzione obiettivo, ad esempio che valutasse come risultato massimo il percorrere i bordi della sorgente di calore continuando a seguire la luce. Noi ci siamo accontentati di una osservazione qualitativa del comportamento ottenuto, rimandando l'analisi quantitativa a questa sottosezione.

da un coordinatore esplicito che opera una somma vettoriale delle due mosse proposte (la lunghezza dei vettori è proporzionale alla forza della regola che ha proposto la mossa). Nella versione parallela un unico sistema a classificatori apprende contemporaneamente a seguire le due luci.

La politica di premio utilizzata è la seguente: nel caso di mossa corretta il sistema prende il premio +40, in caso di mossa sbagliata la punizione -50. Nel caso di sistema distribuito lo stesso premio viene dato ai due sistemi a classificatori (siamo cioè nel caso di architettura distribuita con premio unico presentata in sezione 5.5.4).

Non abbiamo utilizzato messaggi interni (MLI=0).

I risultati ottenuti mostrano che il livello di performance del sistema parallelo è più basso di quello di entrambi i sistemi gerarchici. Questo fatto può essere spiegato dal fatto che lo spazio di ricerca nel caso del sistema parallelo è molto più grande che in quello del sistema distribuito; infatti nel sistema gerarchico ogni SC di basso livello[23] riceve dei messaggi di 12 bit e pertanto per ogni SC lo spazio delle possibili regole è $3^{36} \approx 1.5 \cdot 10^{17}$, mentre per il sistema parallelo i messaggi sono lunghi 22 bit[24] e lo spazio delle possibili regole è $3^{66} \approx 2 \cdot 10^{31}$.

Il confronto fra il sistema vettoriale e quello gerarchico presenta due aspetti interessanti: da una parte, come ci si poteva attendere, il sistema vettoriale, non dovendo apprendere la politica di coordinamento, arriva più velocemente a livelli elevati di performance; il secondo risultato, questa volta inaspettato, è che alla lunga (dopo circa 60000 cicli - circa 6 ore di calcolo) il SC di coordinamento apprende una strategia di coordinamento più efficiente della somma vettoriale pesata.

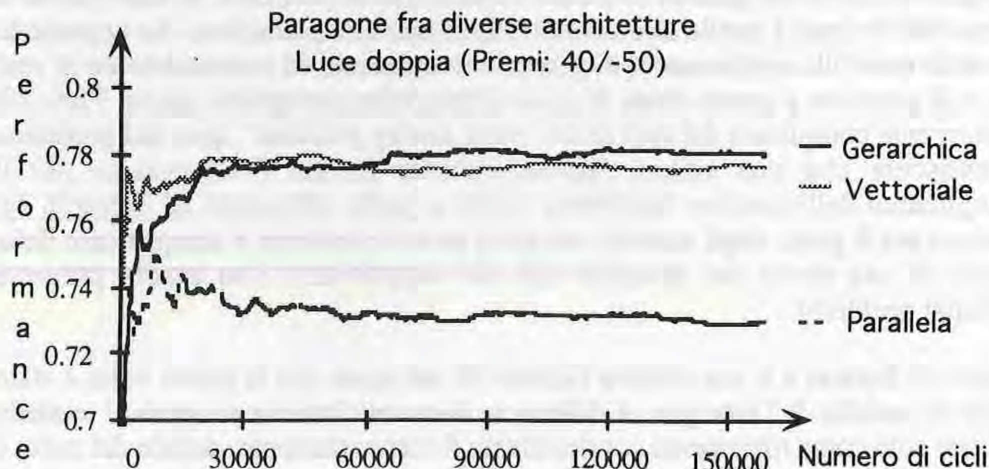


Fig.5.43 - Paragone fra architettura gerarchica, a somma vettoriale e parallela (le curve sono il risultato medio su cinque prove effettuate partendo da differenti popolazioni iniziali generate in modo casuale).

[23] Lo spazio di ricerca del SC coordinatore viene trascurato perché i messaggi che riceve sono lunghi solo 6 bit: 3 per la direzione di movimento proposta, 1 che dice se muoversi o no e due di tag; la sua complessità risulta pertanto essere di ordini di grandezza inferiore a quella dei due sistemi di basso livello.
[24] In questo caso abbiamo 10 bit per ogni luce più 2 bit di tag.

5.6 Lavori correlati

Il lavoro presentato in questo capitolo trova il suo background culturale in due settori di ricerca. Da una parte c'è l'interesse per la valutazione dell'utilità di un approccio all'apprendimento automatico basato sull'utilizzo di tecniche evolutive come gli algoritmi genetici (il che giustifica la presenza di questa ricerca nella presente tesi), dall'altra la convinzione che l'utilizzo di alcune idee prese da studi etologici sul comportamento degli animali possa aiutare a progettare una struttura adeguata ad affrontare problemi della complessità di quelli che si possono porre ad un robot autonomo, con qualche probabilità di successo.

Mentre dalle ricerche in etologia il nostro lavoro ha preso degli spunti isolati, il settore algoritmi genetici e in particolare il lavoro di Holland ([61], [63], [62], [10]), hanno avuto una grande importanza. I modelli sviluppati da Holland formano il substrato concettuale del nostro lavoro. La nostra ricerca è comunque stata influenzata anche da altri lavori tra cui alcuni, che hanno ricoperto un ruolo particolarmente importante, sono citati qui di seguito.

(i) Il lavoro di Wilson [88] sull' "*Animat* problem", cioè sul problema che deve essere affrontato e risolto da un animale artificiale che deve apprendere come fare a sopravvivere nel suo ambiente. Wilson propone il seguente ragionamento per spiegare perché questo sia un compito difficile: l'informazione che giunge al robot attraverso i suoi sensori dall'ambiente esterno è difficile da classificare perché l'Animat non ha conoscenza a priori sul come associare situazioni ambientali ad azioni appropriate che lo portino più vicino al raggiungimento dell'obiettivo prefissato (cioè alla sopravvivenza). Il tutto è complicato dalla supposta assenza di un insegnante che possa guidare l'Animat durante questo processo. In altre parole il problema dell'Animat è quello di costruire delle classi di equivalenza che coprano lo spazio delle possibili combinazioni di percezioni sensoriali, ed eventualmente di stati interni, e di associare a queste classi di equivalenza delle appropriate azioni. Tutto ciò è ulteriormente complicato dal così detto "*stage setting problem*", cioè dal problema di riconoscere che una azione apparentemente inutile è necessaria per il raggiungimento dell'obiettivo (problema simile a quello affrontato ad esempio dai programmi per il gioco degli scacchi, nei quali però il problema è semplificato dalla esistenza di una teoria del dominio che noi supponiamo non essere presente nell'Animat problem).

(ii) La ricerca di Booker e il suo sistema GOFER [9] nel quale per la prima volta è stato utilizzato il modello di Tinbergen. A differenza di quanto fatto da noi però, il modello è utilizzato solo come riferimento per descrivere il comportamento globale del robot e non è rappresentato esplicitamente nel sistema. Tutti i comportamenti (esplorazione, avvicinamento, fuga) sono incorporati in un singolo sistema a classificatori, mentre nel nostro sistema i comportamenti sono raggruppati e implementati per mezzo di diversi SC. Questa struttura ci ha permesso di risolvere e superare alcuni dei problemi riportati da Booker (ad esempio competizione tra classificatori che realizzano comportamenti completamente differenti, scomparsa di particolari sequenze di classificatori non rilevanti in una determinata situazione, insufficiente distinzione tra messaggi di coordinamento e messaggi azione). Un'ulteriore differenza consiste nel fatto che GOFER utilizza una strategia detta "winner-take-all", cioè l'azione effettuata è quella proposta dalla regola vincente, mentre noi utilizziamo delle tecniche di mediazione (ad esempio tramite l'utilizzo del SC di coordinamento).

(iii) Il sistema CSM (Classifier System with Memory) di Zhou [89]. Nel suo lavoro Zhou affronta il problema della memoria di breve e di lungo termine. Si pone cioè il problema di come utilizzare l'esperienza passata per semplificare l'attività di risoluzione dei problemi in presenza di situazioni nuove. Il suo approccio è quello di costruire un sistema nel quale accanto alla memoria di breve termine, rappresentata dalle usuali regole con forze associate, esiste una memoria di lungo termine rappresentata da insiemi di regole ottenute per mezzo di un algoritmo di generalizzazione che tiene conto dell'esperienza acquisita per risolvere una determinata classe di problemi. Ogni volta che si presenta un problema nuovo l'Animat inizializza la sua base di regole utilizzando quegli insiemi di regole appartenenti alla memoria di lungo termine che meglio sembrano adattarsi al problema; dopodiché il sistema continua a funzionare come i SC tradizionali: il vantaggio evidente è che la base di regole utilizzata in presenza di un problema nuovo (cioè che non può essere risolto con la base di regole in quel momento presente) non deve essere rigenerata a partire da quella attuale o da una nuova generata in modo casuale.

5.7 Conclusioni e sviluppi futuri

I sistemi a classificatori sono un modello di calcolo particolarmente adatto alla risoluzione del problema che si pone ad un animale artificiale (Animat [88]): imparare a sopravvivere in un ambiente dato cercando di massimizzare una funzione che esprime lo stato di benessere, basandosi su un unico tipo di informazione chiamato *rinforzo*. Il *rinforzo* è un premio o una punizione che viene dato all'Animat come conseguenza di una azione effettuata in una data situazione ambientale.

In questo capitolo abbiamo affrontato numerosi aspetti riguardanti:

- lo studio di una architettura adeguata alla soluzione dei problemi tipici proponibili ad un Animat (sezione 5.2),
- gli aspetti tecnici del funzionamento dei SC (sezione 5.3),
- la parallelizzazione di un SC secondo diverse modalità (sezione 5.4),

Per ognuno degli aspetti indagati sono stati sviluppati dei programmi di simulazione che sono stati utilizzati per una serie di esperimenti i cui risultati sono presentati nelle sezioni 5.3 e 5.5.

I risultati ottenuti permettono di concludere che l'applicazione di sistemi di apprendimento automatico basati su algoritmi genetici e sistemi a classificatori a problemi di robotica è un settore di ricerca molto promettente. L'utilità di questi metodi nel caso dell'apprendimento di semplici compiti è stata dimostrata. L'aspetto più interessante sul quale lavorare nel prossimo futuro sarà la progettazione di sistemi gerarchici più complessi di quelli qui presentati e la loro realizzazione su robot veri. A tal fine sono in corso esperimenti con un piccolo robot autonomo denominato AutonoMouse dotato di capacità sensoriali ed attuatoriali simili a quelle del robot simulato negli esperimenti di questo capitolo (sensori di luce, di campo magnetico, di contatto, di suoni e due motori per il controllo di ruote indipendenti).

6

Bibliografia

1. Abramson, A. Constructing School Timetables using Simulated Annealing: Sequential and Parallel Algorithms. *Management Science 37*, 1 (1991), 98-113.

2. Akkonyunlu, E.A. A Linear Algorithm for Computing the Optimum of University Timetable. *Computer Journal 16*, 4 (1973), 347-350.

3. Antonisse, H.J. A New Interpretation of Schema Notation that Overturns the Binary Encoding Constraint. In *Proceedings Third International Conference on Genetic Algorithms*, Morgan Kaufmann, George Mason University, June 1989.

4. Bäck, T., Hoffmeister, F., and Schwefel, H.P. A survey of evolution strategies. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, UCSD - San Diego - CA, 1991.

5. Berman, L. and Hartmanis, J. On isomorphism and density of NP and other complete sets. *SIAM Journal on Computing 1*(1977), 305-322.

6. Bersini, H. Hints for Adaptive Problem Solving Gleaned from Immune Networks. In *Proceedings First International Conference on Parallel Problem Solving from Nature (PPSN I)*, Schwefel, H.P. and Männer, R., Springer-Verlag, 1990.

7. Bersini, H. and Varela, F.J. The Immune Recruitment Mechanism: a Selective Evolutionary Strategy. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, UCSD - San Diego - CA, July 1991.

8. Bonarini, A., Dorigo, M., and Maniezzo, V. AutonoMouse: an Experiment in Grounded Behaviors. In *Proceedings of GAA91 - Second Italian Workshop on Machine Learning*, Bari - Italy, March 1991.

9. Booker, L.B. Classifier Systems that Learn Internal World Models. *Machine Learning 3*, 3 (1988), 161-192.

10. Booker, L.B., Goldberg, D.E., and Holland, J.H. Classifier Systems and Genetic Algorithms. *Artificial Intelligence 40*, 2 (1989), 235-282.

11. Boyd, S.C., Pulleyblank, W.R., and Cornuejols, G., *Travel*, Software Package - Carleton University, January 1989.

12. Bruschi, D., Joseph, D., and Young, P., *A Structural Overview of NP Optimization Problems*, Newsletter of the ESPRIT II BRA - Project no.3075 (ALCOM), May 1991, Vol.2 No.1.

13. Burkard, R.E. Quadratic Assignment Problems. *European Journal of Operational Research 15*(1984), 283-289.

14. Camilli, A. and Di Meglio, R. *Classifiers systems in massively parallel architectures*, Master's thesis, University of Pisa, Italy, 1990.

15. Camilli, A., Di Meglio, R., Baiardi, F., Vanneschi, M., Montanari, D., and R. Serra, Classifier System Parallelization on MIMD architectures. Tech. Rept. 3/17 CNR, March, 1990.

16. Carbonell, J.G., Michalsky, R.S., and Mitchell, T.M. *Machine Learning - An Artificial Intelligence Approach*, Springer-Verlag (1984).

17. Caruana, R.A. and Schaffer, J.D. Representation and hidden bias: Gray vs binary coding for genetic algorithms. In *Proceedings Fifth International Conference on Machine Learning,* Morgan Kaufmann, San Mateo, CA, 1988, pp. 153-161.

18. Chahal, N. and de Werra, D. An Interactive System for Constructing Timetables on a PC. *European Journal of Operational Research 40,* 1 (1989), 32-37.

19. Colorni, A., Dorigo, M., and Maniezzo, V. Genetic Algorithms: A New Approach to the Time-Table Problem. In *NATO series on Combinatorial Optimization,* Springer-Verlag, 1990.

20. Colorni, A., Dorigo, M., and Maniezzo, V. Genetic Algorithms and Highly Constrained Problems: The Timetable Case. In *Proceedings First International Conference on Parallel Problem Solving from Nature (PPSN I),* Schwefel, H.P. and Männer, R., Springer-Verlag, 1990, pp. 55-59.

21. Colorni, A., Dorigo, M., and Maniezzo, V. On the Use of Genetic Algorithms to Solve the Time-Table Problem. Tech. Rept. 90-60, Politecnico di Milano - Department of Electronics, Italy, December, 1990.

22. Colorni, A., Dorigo, M., and Maniezzo, V. Positive feedback as a search strategy. Tech. Rept. 91-16, Politecnico di Milano - Department of Electronics, Milano - Italy, November, 1991.

23. Colorni, A., Dorigo, M., and Maniezzo, V. Distributed optimization by ant colonies. In *Proceedings First European Conference on Artificial Life,* MIT Press/Bradford Books, Paris, December 1991.

24. Colorni, A., Dorigo, M., and Maniezzo, V. Il modello 'Formiche': un approccio distribuito ai problemi di ottimizzazione combinatoria. In *Proceedings of the Annual Conference of the Operational Research Society of Italy,* Riva del Garda, Italy, September 1991.

25. Colorni, A., Dorigo, M., and Maniezzo, V. Gli algoritmi Genetici e il Problema dell'Orario. *Rivista di Ricerca Operativa* (1992).

26. Csima, J. and Gotleib, C.C. Tests on a Computer Method for Construction of School Timetables. *Communications of the ACM* 7, 3 (1961), 160-163.

27. Davis, L. and Ritter, F. Schedule Optimization with Probabilistic Search. In *Proceedings Third IEEE Conference on Artificial Intelligence Applications,* February 1987.

28. De Jong, K.A. and Spears, W.M. Using Genetic Algorithms to Solve NP-Complete Problems. In *Proceedings Third International Conference on Genetic Algorithms,* Morgan Kaufmann, June 4-7 1989.

29. Denebourg, J.L., Pasteels, J.M., and Verhaeghe, J.C. Probabilistic Behaviour in Ants: a Strategy of Errors?. *Journal of Theoretical Biology 105*(1983), 259-271.

30. Dorigo, M. Genetic Algorithms: The State of the Art and Some Research Proposals. Tech. Rept. 89-58, Politecnico di Milano, Milano - Italy, December, 1989.

31. Dorigo, M. and Schnepf, U. A Bootstrapping Approach to Robot Intelligence: First Results. Tech. Rept. 90-68, Politecnico di Milano - Department of Electronics - PM-AI & R Project, Italy, December, 1990.

32. Dorigo, M. Machine Learning: An Approach Based on Classifier Systems and Evolutionary Algorithms. Tech. Rept. 90-43, Politecnico di Milano - Department of Electronics - PM-AI & R Project, Italy, October, 1990.

33. Dorigo, M. and Maniezzo, V. Translating Evolution Strategies in GA Terms. Tech. Rept. 91-7, Politecnico di Milano - Department of Electronics - PM-AI & R Project, Italy, 1991.

34. Dorigo, M. and Maniezzo, V. Evolution Strategies and Genetic Algorithms in Optimization. In *Proceedings of the Annual Conference of the Operational Research Society of Italy,* Riva del Garda, Italy, September 1991.

35. Dorigo, M. and Sirtori, E. A Learning Environment for Robots. In *Proceedings of GAA91 - Second Italian Workshop on Machine Learning,* Bari - Italy, March 1991.

36. Dorigo, M. and Schnepf., U. Organisation of Robot Behaviour Through Genetic Learning Processes. In *Proceedings of ICAR - Fifth International Conference on Advanced Robotics,* IEEE, Pisa, Italy , June 1991.

37. Dorigo, M. and Schnepf, U. Genetics-based Machine Learning and Behaviour Based Robotics: A New Synthesis. Tech. Rept. 91-44, Politecnico di Milano - Department of Electronics - PM-AI & R Project, Italy, February, 1991 (to appear on: *IEEE Transactions on Systems, Man and Cybernetics*).

38. Dorigo, M. and Sirtori, E. Alecsys: A Parallel Laboratory for Learning Classifier Systems. In *Proceedings of Fourth International Conference on Genetic Algorithms,* Morgan Kaufmann, UCSD - San Diego - CA, July 1991.

39. Dorigo, M. Message-Based Bucket Brigade: An Algorithm for the Apportionment of Credit Problem. In *Proceedings of the European Working Session on Learning,* Springer-Verlag, Porto, Portugal, March 1991.

40. Dorigo, M. New Perspectives about Default Hierarchies Formation in Learning Classifier Systems. In *Proceedings of II Italian Congress on Artificial Intelligence,* Springer-Verlag, Palermo, Italy, October 1991.

41. Dorigo, M. Default Rule Hierarchies in Genetics-based Machine Learning . Tech. Rept. 91-2, Politecnico di Milano - Department of Electronics - PM-AI & R Project, Italy, January, 1991.

42. Dorigo, M. and Sirtori, E. A Parallel Distributed Environment for Genetics-based Machine Learning. Tech. Rept. 91-15, Politecnico di Milano - Department of Electronics - PM-AI & R Project, Italy, April, 1991.

43. Dorigo, M. Using Transputer to increase Speed and Flexibility of Genetics-based Machine Learning Systems. *Microprocessing and Microprogramming, Euromicro Journal, North Holland* (1992).

44. Eilon, S., Watson-Gandy, T.H., and Christofides, N. Distribution management: mathematical modeling and practical analysis. *Operational Research Quarterly 20*(1969), 37-53.

45. Even, S., Itai, A., and Shamir, A. On the Complexity of Timetable and Multicommodity Flow Problems. *SIAM Journal of Computing 5*, 4 (1976), 691-703.

46. Fitzgerald, T.D. and Peterson, S.C. Cooperative foraging and communication in caterpillars. *Bioscience 38*(1988), 20-25.

47. Flynn, M.J. Some Computer Organizations and their Effectiveness. *IEEE Transaction on Computers C-21*, 9 (September 1972), 948-960.

48. Forrest, S. Parallelism in Classifier Systems. In *Proceedings of the First International Conference on Parallel Problem Solving from Nature (PPSN I)*, Springer-Verlag, Dortmund - Germany, October 1990.

49. Garey, M.R. and Johnson, D.S. *Computers and Intractability*, Freeman & C. (1979).

50. Gianoglio, P. Application of Neural Networks to Timetable Construction. In *Proceedings of the third International Workshop on Neural Networks & Their Applications*, November 1990.

51. Glover, F. Tabu Search — Part I. *ORSA Journal on Computing 1*, 3 (1989), 190-206.

52. Glover, F. Tabu Search — Part II. *ORSA Journal on Computing 2*, 1 (1990), 4-32.

53. Goldberg, D.E. and Lingle, R. Alleles, Loci, and the Traveling Salesman Problem. In *Proceedings of the First International Conference on Genetic Algorithms*, Morgan Kaufmann, Carnegie-Mellon University - Pittsburg - PA, July 1985.

54. Goldberg, D.E. *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley, Reading, Massachussets (1989).

55. Goldberg, D.E. Sizing populations for serial and parallel Genetic Algorithms. In *Proceedings Third International Conference on Genetic Algorithms*, Morgan Kaufmann, George Mason University, June 1989 , pp. 70-79.

56. Goldberg, D.E. and Smith, R.E. Reinforcement Learning with Classifier Systems. In *Proceedings of Congress on AI, Simulation and Planning in High Autonomous Systems*, University of Arizona, IEEE Computer Society Press, Tucson, March 1990.

57. Goss, S., Beckers, R., Denebourg, J.L., Aron, S., and Pasteels, J.M. *Behavioural Mechanisms of Food Selection*, Springer-Verlag: Berlin, Vol. G20, NATO ASI (1990).

58. Grefenstette, J.J. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics SMC- 16*, 1 (1986), 122-128.

59. Hertz, A. *La coloration des sommets d'un graphe et son application a la confection d'horaires*, Ph.D. dissertation, Ecole Polytechnique Fed. de Lausanne , Switzerland, 1989.

60. Hertz, A. and de Werra, D. Informatique et horaires scolaires. *Output 12*(1989), 53-56.

61. Holland, J.H. *Adaptation in natural and artificial systems*, Ann Arbor: The University of Michigan Press (1975).

62. Holland, J.H., Holyoak, K.J., Nisbett, R.E., and Thagard, P.A. *Induction, Processes of Inference, Learning and Discovery*, The MIT press, Cambridge, MA (1986).

63. Holland, J.C. *Escaping brittleness: The possibilities of general purpose learning algorithms applied to parallel rule-based systems,* Morgan Kaufmann: Los Altos, CA, Machine Learning II (1986).

64. Johnson, D.S. Local Optimization and the Traveling Salesman Problem. In *Proceedings of the 17th International Colloquium Warwick University,* Paterson, M.S., Springer-Verlag, 1990, pp. 446-461.

65. Joseph, D. and , Young, P. Some remarks on witness functions for polynomial reducibilities in NP. *Theoretical Computer Science 39*(1985), 225-237.

66. Koopmans, T.C. and Beckmann, M.J. Assignment Problems and the Location of Economic Activities. *Econometrica 25*(1957), 53-76.

67. Lawrie, N.H. An Integer Programming Model for a School Time-tabling Problem. *Computer Journal 12*(1969), 307-316.

68. Lin, S. and Kernighan, B.W. An effective Heuristic Algorithm for the TSP. *Operations Research 21*(1973), 498-516.

69. Mühlenbein, H. Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization. In *Proceedings Third International Conference on Genetic Algorithms,* Morgan Kaufmann, Carnegie Mellon University, June 1989.

70. Newell, A. and Simon, H.A. *Human Problem Solving,* Prentice-Hall (1972).

71. Post, E. Formal reductions of the general combinatorial problem. *American Journal of Mathematics 65*(1943), 197-268.

72. Rechenberg, I. *Evolutionsstrategie,* Fromman-Holzbog, Stuttgart, Germany (1973).

73. Reinelt, G., *TSPLIB 1.0,* Institut für Mathematik, Universität Augsburg, Germany, January 1990, .

74. Riolo, R.L. and Robertson, G.G. A Tale of Two Classifier Systems. *Machine Learning 3*, 2/3 (1988), 139-159.

75. Robertson, G.G. Parallel Implementation of Genetic Algorithms in a Classifier System. In *Proceedings of the Second International Conference on Genetic Algorithms,* Lawrence Erlbaum, MIT - Cambridge - MA, July 1987.

76. Schaffer, J.D., Caruana, R.A., Eshelman, L.J., and Das, R. A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization. In *Proceedings Third International Conference on Genetic Algorithms,* Morgan Kaufmann, George Mason University, June 1989 .

77. Schoemaker, P.J.H. The quest for optimality: A positive heuristic of science? *Behavioral and Brain Sciences 14*, 2 (June 1991).

78. Schwefel, H.P. *Numerical Optimization of Computer Models,* Wiley, New York (1981).

79. Shapiro, J.A. Bacteria as multicellular organisms. *Scientific American* (1988), 82-89.

80. Tinbergen, N. *The Study of Instincts,* Oxford University Press (1966).

81. Traub, J.F., Wasilkowski, G.W., and Wozniakowski, H. *Information-Based Complexity,* Academic Press, Inc. (1988).

82. Varela, F., Coutinho, B., Dubire, D., and Vaz, N. *Theoretical Immunology*, Addison Wesley: New Jersey, Vol. 2 , SFI Series on the Science of Complexity (1988).

83. Varela, F., Sanche, V., and Coutinho, B. *Evolutionary and Epigenetic order from complex systems: A Waddington Memorial Volume,* Edinburgh University Press (1989).

84. Varela, F. and Stewart, J. Dynamics of a class of immune networks. *Journal of theoretical Biology 144*(1990), 93-101.

85. de Werra, D. A Few Remarks on Chromatic Scheduling. Tech. Rept. Dep. of Mathematics, Ecole Polytechnique Federale de Lausanne, Switzerland, 1985.

86. de Werra, D. An Introduction to Timetabling. *European Journal of Operational Research* (1985), 151-162.

87. Whitley, D., Starkweather, T., and Fuquay, D. Scheduling Problems and Travelling Salesmen: the Genetic Edge Recombination Operator. In *Proceedings Third International Conference on Genetic Algorithms,* Morgan Kaufmann, George Mason University, 1989.

88. Wilson, S. Classifier Systems and the Animat Problem. *Machine Learning 2*, 3 (1987), 199-228.

89. Zhou, H.H. CSM: A Computational Model of Cumulative Learning. *Machine Learning 5*, 4 (1990), 383-406.