UNIVERSITA' DEGLI STUDI DI PADOVA

Facoltà di Scienze Statistiche

Corso di laurea specialistica in Statistica e Informatica

TESI DI LAUREA

Algoritmi Genetici con applicazioni ai giochi ed ai problemi di ottimizzazione

Relatore
Prof. Stuart Coles

Candidato
Daniele Rampoldi

Anno Accademico 2005/2006

Indice

1 Gli Algoritmi Genetici **7**

1.1 Principi di base 9

1.1.1 Codifica 10

1.1.2 Funzione di valutazione 11

1.1.3 Riproduzione selettiva 11

1.1.4 Crossover e Mutazione 13

1.2 Un primo esempio 15

2 La Teoria degli schemi **19**

2.1 Il processo evolutivo in dettaglio 21

2.2 Teorema degli Schemi 26

2.3 Ipotesi dei blocchi costitutivi 27

3 Aspetti pratici **31**

3.1 Scelta della codifica 32

3.1.1 Codifiche ad hoc 33

3.1.2 Mapping dei valori ridondanti 34

3.2 Funzione Fitness e Selezione 36

3.2.1 Funzione Fitness 36

3.2.2 Fitness e Selezione 38

3.2.3 Tecniche di selezione dei genitori 39

3.2.4 Elitismo e Steady-State Reproduction 41

3.3 Crossover 42

3.3.1 2 - point Crossover 43

3.3.2 Crossover Uniforme 44

3.3.3 Quale tecnica è la migliore? 45

3.3.4 Altre tecniche di Crossover 46
3.4 Mutazione ed Evoluzione Naive 46

4 Aspetti avanzati **49**
4.1 Inversione e Ordinamento 49
4.2 Epistasis 50
4.2.1 Deception (inganno) 52
4.2.2 Affrontare l'epistasi 52
4.3 Operatori Dinamici 53
4.4 Nicchia e Specificazione 54
4.4.1 Restricted Mating 55
4.5 Duplicità e Dominio 55
4.6 Tecniche basate sulla conoscenza 56
4.7 Diversi algoritmi genetici 57
4.8 Confronto con altre tecniche 57
4.8.1 Esplorazione e sfruttamento 58
4.8.2 Vantaggi offerti dagli algoritmi genetici 59

5 Il Master Mind **63**
5.1 Giocare a Master Mind 63
5.2 Strategia 64
5.2.1 Scelta della Miglior Prova 64
5.2.2 L'operatore di crossover 65
5.2.3 L'operatore di mutazione 66
5.2.4 Consistenza 67
5.3 Un esempio pratico 67
5.4 Spiegazione del codice R 70
5.4.1 Parametri di input e variabili 71
5.4.2 La correzione delle probabilità 73
5.4.3 Consistenza delle prove 74
5.5 Analisi Statistica sulla bontà dell'algoritmo 75
5.5.1 Il codice alternativo 76
5.5.2 Confronto 78

6 Il Problema del Commesso Viaggiatore **83**
6.1 Codifica 85
6.1.1 Rappresentazione con vettori 85
6.1.2 Rappresentazione con matrici 87
6.2 Scelta delle prove 95
6.3 L'operatore di crossover 96
6.3.1 PMX (Partially-Mapped Crossover) 96
6.3.2 OX (Order Crossover) 97
6.3.3 CX (Cycle Crossover) 97
6.3.4 OBX (Order-Based Crossover) 98
6.3.5 ERX (Edge Recombination Crossover) 99
6.4 L'operatore di mutazione 100
6.5 Spiegazione del codice R 102
6.5.1 Valori di input e di output 102
6.5.2 Aspetti particolari 103
6.6 Un esempio pratico 104

A Codice R del Master Mind **107**
A.1 Codice R commentato 107

B Codice R del TSP **125**
B.1 Codice R commentato 125

C Codice R del Master Mind Alternativo **135**
C.1 Codice R commentato 135

Bibliografia **149**

Ringraziamenti **155**

Capitolo 1

Gli Algoritmi Genetici

Gli Algoritmi Genetici sono una famiglia di tecniche di ottimizzazione che si ispirano alla teoria della selezione naturale di Charles Darwin che regola l'evoluzione biologica. I sistemi biologici esibiscono un alto grado di robustezza ed efficienza nel risolvere una vasta serie di problemi essenziali per la loro sopravvivenza, ma, al contrario dei sistemi artificiali creati dall'uomo, essi non sono il frutto di un disegno progettuale, bensì sono il risultato di un processo evolutivo basato sulla riproduzione selettiva degli individui migliori, sulla ricombinazione genetica dei loro cromosomi e su alcune mutazioni casuali.
Anche se l'esatto funzionamento dei meccanismi evolutivi è ancora oggetto di discussione, vi è un generale accordo su alcuni principi di base che contraddistinguono lo sviluppo filogenetico degli organismi viventi:

- l'evoluzione naturale opera sui cromosomi degli individui invece che sugli individui stessi, ovvero su di una codificazione genetica (il 'genotipo') delle caratteristiche fisiche dell'organismo (il 'fenotipo');

- il processo di selezione naturale favorisce la riproduzione generazionale dei cromosomi che hanno dato luogo agli organismi più efficienti dal punto di vista adattativo: esso è quindi il momento in cui il fenotipo esercita una influenza sul genotipo, anche se questa influenza è solo indiretta perché non prevede la trascrizione diretta nel genotipo delle caratteristiche acquisite dal fenotipo nel corso della sua vita;

- i meccanismi biologici della riproduzione costituiscono il cuore del processo evolutivo: la combinazione dei codici genetici dei due genitori e l'intro-

duzione di piccole mutazioni casuali dà luogo alla creazione di nuove strutture genetiche la cui bontà adattiva (e di conseguenza la probabilità di riproduzione) assume un significato solo in relazione alle capacità di sopravvivenza degli individui corrispondenti;

- l'evoluzione naturale opera su popolazioni di individui attraverso un processo ciclico e generazionale che non possiede memorie storiche, ma si basa esclusivamente su contingenze ambientali definite a livello dell'interazione fra ogni singolo individuo e il proprio ambiente ecologico.

Il funzionamento dell'evoluzione naturale è stato paragonato da Richard Dawkins al lavoro di un orologiaio cieco[1]: benché le modalità operative siano basate essenzialmente su dei processi casuali senza alcuna finalità definita a priori, i sistemi biologici possiedono un grado di complessità e di robustezza perfettamente adeguato ai compiti che essi devono svolgere per poter sopravvivere.
I sistemi biologici possiedono dunque molte caratteristiche di robustezza, di auto-organizzazione, di adattamento e di efficienza che sono altamente desiderabili se catturate ed incorporate nei sistemi artificiali creati dall'uomo. Tuttavia i metodi tradizionali di costruzione dei sistemi artificiali difficilmente riescono ad ottenere risultati eguagliabili alle prestazioni degli esseri viventi anche nei compiti più semplici, come ad esempio la navigazione, la ricerca di un rifugio, la fuga da un predatore o il riconoscimento di oggetti. Uno dei motivi di questo fallimento risiede probabilmente nel fatto che i metodi tradizionali si basano su procedimenti analitici (isolamento del problema, definizione teorica, identificazione delle variabili e derivazione formale di una soluzione specifica) che sono molto diversi dai principi di sviluppo dei sistemi biologici.
Gli algoritmi genetici invece si basano su procedimenti molto simili a quelli impiegati dall'evoluzione naturale e possiedono due precise finalità: il tentativo di comprendere meglio i meccanismi di sviluppo filogenetico dei sistemi viventi attraverso un processo simulativo di sintesi e il desiderio di sfruttare questi procedimenti per la creazione di sistemi artificiali che esibiscano le stesse caratteristiche di robustezza, efficienza e flessibilità degli organismi viventi nel-

[1] Dawkins, R. 'The Blind Watchmaker' (1986), London, Penguin Books; trad. it. 'L'orologiaio cieco' (1988), Milano, Rizzoli.

l'eseguire compiti difficilmente risolvibili con i metodi tradizionali.

L'avvento degli elaboratori elettronici ha rappresentato una svolta rivoluzionaria nella storia della scienza e della tecnologia, che sta accrescendo enormemente la nostra capacità di predire e controllare la natura in forme che erano a malapena immaginabili soltanto mezzo secolo fa. Per molti, il coronamento di questa rivoluzione sarà la creazione, sotto forma di programmi per il calcolatore, di nuove specie di esseri intelligenti e addirittura di nuove forme di vita.
In effetti, l'obbiettivo di creare l'intelligenza artificiale e la vita artificiale può venire fatto risalire alle origini dell'era informatica: già i primi informatici inseguivano il sogno di instillare nei programmi l'intelligenza, la capacità di duplicarsi e di imparare, nonché quella di controllare l'ambiente circostante.
Il tentativo di costruire modelli per il cervello ha dato origine al settore delle reti neurali, quello di imitare l'apprendimento umano al settore dell'apprendimento automatico e, infine, la simulazione dell'evoluzione biologica ha dato vita al campo della computazione evolutiva, di cui gli algoritmi genetici sono l'esempio più importante.

Gli algoritmi genetici sono applicabili ad un'ampia varietà di problemi d'ottimizzazione non indicati per gli algoritmi classici, compresi quelli in cui la funzione obbiettivo è discontinua, non derivabile, stocastica, o fortemente non lineare.

1.1 Principi di base

Gli algoritmi genetici operano su di una popolazione di cromosomi artificiali che vengono fatti riprodurre selettivamente in base alle prestazioni dei fenotipi a cui danno origine rispetto al problema da risolvere. Durante il processo riproduttivo le repliche dei cromosomi degli individui migliori vengono accoppiate casualmente e parte del materiale genetico viene scambiato, mentre alcune piccole mutazioni casuali alterano localmente la struttura del codice genetico. Le nuove strutture genetiche vanno quindi a rimpiazzare quelle dei loro genitori

dando luogo ad una nuova generazione di individui. Il processo continua fino a quando nasce un individuo che rappresenta una soluzione accettabile per il problema in esame.
Gli algoritmi genetici si basano quindi su tre operatori principali: la riproduzione selettiva degli individui migliori, la ricombinazione genetica (crossover) e le mutazioni casuali dei cromosomi. Prima di analizzare con maggior dettaglio il funzionamento di questi operatori, osserviamo che vi sono due importanti strumenti necessari allo sviluppo di un algoritmo genetico: la codifica genetica e la funzione di valutazione.

1.1.1 Codifica

Per 'codifica genetica' ci si riferisce al tipo di rappresentazione che viene utilizzata per identificare le soluzioni del problema nei cromosomi artificiali. Un cromosoma artificiale è una sequenza di simboli: per questo motivo viene comunemente definito con il nome di 'stringa genetica'.
Un tipo di codice usato molto frequentemente è quello binario: in questo caso il cromosoma di ciascun individuo della popolazione è una stringa di lunghezza finita composta di simboli binari. Vi sono molti altri tipi di codifica possibile: un cromosoma può essere composto da numeri reali oppure da una serie finita di simboli appartenente ad un qualsiasi alfabeto arbitrario. Il funzionamento dell'algoritmo genetico non è compromesso dal tipo di rappresentazione perché gli operatori genetici si limitano a selezionare le stringhe corrispondenti ai fenotipi migliori e a ricombinarne i vari pezzi a prescindere dal tipo di materiale su cui essi lavorano.
La scelta del tipo di codifica è però importante per il tipo di problema che si vuole risolvere: non esiste una codifica che vada bene per tutti i problemi, né esistono delle regole generali che permettano di fare delle scelte ottimali.
Il problema della rappresentazione genetica e delle regole di decodificazione da genotipo a fenotipo è quindi molto importante per poter sfruttare al meglio le potenzialità di ricerca dell'algoritmo genetico.

1.1.2 Funzione di valutazione

La funzione di valutazione serve per giudicare le prestazioni di ciascun fenotipo rispetto al problema che vogliamo risolvere: essa fornisce un valore numerico per ciascun individuo proporzionale alla bontà della soluzione offerta. Questa particolare funzione svolge un ruolo analogo a quello dell'ambiente fisico per gli organismi biologici, in quanto misura le prestazioni dell'individuo: per questo motivo viene comunemente definita 'funzione di fitness'.
Una caratteristica interessante degli algoritmi genetici è che la funzione di fitness può assumere vari gradi di complessità a seconda delle conoscenze disponibili: l'unico requisito è che essa permetta un ordinamento delle stringhe genetiche in base alle loro prestazioni sul problema da risolvere.

1.1.3 Riproduzione selettiva

Una volta definito il tipo di rappresentazione genetica e la funzione di fitness, il primo passo consiste nella creazione di una popolazione iniziale di stringhe genetiche. Solitamente la popolazione iniziale è composta da stringhe casuali. Ciascuna stringa di questa generazione iniziale viene a turno decodificata e valutata in base alla funzione di fitness.
Il processo di riproduzione selettiva consiste nella creazione probabilistica di un numero di copie di ciascuna stringa proporzionale al valore di fitness ottenuto dal fenotipo corrispondente. Ricopiare ciascuna stringa in proporzione al proprio valore di fitness significa che le stringhe che hanno riportato un valore di fitness maggiore avranno una probabilità maggiore di produrre uno o più figli: l'operatore di riproduzione selettiva svolge dunque un ruolo simile alla legge di sopravvivenza del più forte in natura.
Vi sono diversi modi di realizzare al computer la riproduzione selettiva probabilistica: il metodo più diffuso fa ricorso all'utilizzo di una *ruota della fortuna truccata* (Figura 1.1).
Consideriamo la situazione comune in cui manteniamo una popolazione di dimensione costante ad ogni generazione e in cui rimpiazziamo completamente tutti i membri della popolazione ad ogni ricambio generazionale. La ruota della fortuna avrà dunque tante caselle quante sono gli individui della popolazione,

ma la dimensione di ciascuna casella sarà proporzionale ai valori di fitness di ciascun individuo.
La riproduzione selettiva consiste nel far girare la ruota tante volte quanti sono gli individui da generare e nel creare ogni volta una copia della stringa corrispondente alla casella in cui si trova alla fine l'indice della ruota.

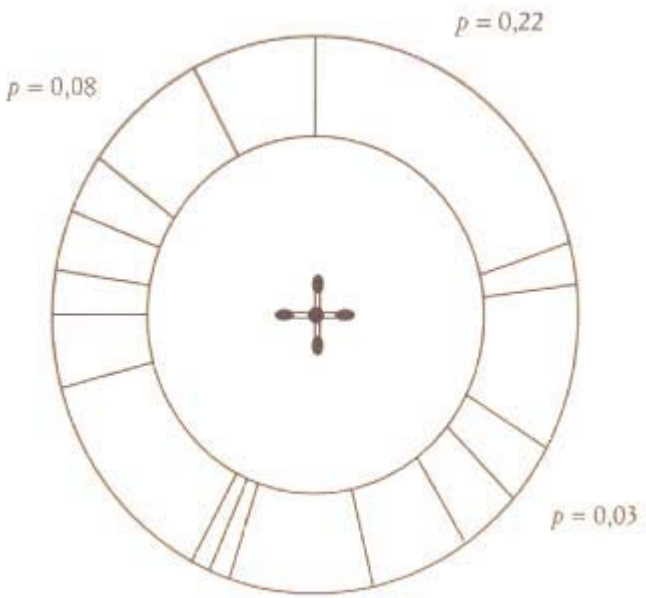

Figura 1.1: *La ruota della fortuna truccata: l'ampiezza di ciascuna casella è proporzionale al valore di fitness della stringa genetica corrispondente. Stringhe con valori di fitness maggiori avranno una probabilità maggiore di essere estratte una o anche più volte.*

Formalmente il processo di valutazione e riproduzione selettiva avviene assegnando al fenotipo x_i di ciascuna stringa un valore di fitness f_i in questa maniera:

$$f_i = \Phi(x_i) \ ,$$

dove $\Phi(x_i)$ è la funzione di valutazione. Questo valore viene utilizzato per

calcolare la probabilità P_i della stringa di essere selezionata per la riproduzione (che corrisponde all'ampiezza della casella della ruota della fortuna truccata)

$$P_i = \frac{f_i}{\sum_i^N f_i} \quad ,$$

pesando il valore di fitness della stringa per la somma dei valori di fitness di tutte le stringhe della popolazione. Il valore atteso del numero di figli per ciascuna stringa è dato da NP_i, ma il numero effettivo di figli si ottiene facendo ricorso ad un generatore di numeri casuali.

1.1.4 Crossover e Mutazione

Le nuove stringhe così create vengono poi disposte a coppie in modo casuale e sottoposte all'azione dell'operatore di ricombinazione genetica (*crossover*). Ciascuna coppia viene incrociata con una determinata probabilità p_c. Per ciascuna delle coppie selezionate viene scelto un punto d'incrocio casuale attorno al quale avviene uno scambio reciproco di materiale genetico (Figura 1.2).

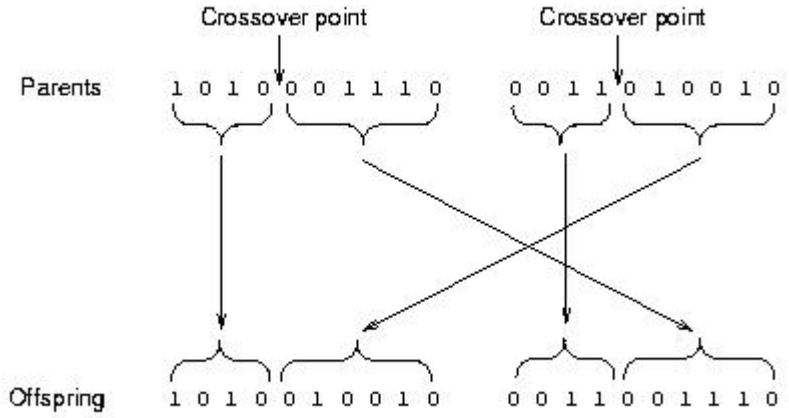

Figura 1.2: *Un esempio di ricombinazione genetica: il 'Single-Point Crossover'.*

Infine tutte le stringhe della popolazione subiscono un processo di *mutazione*: ciascun elemento del cromosoma cambia il proprio valore in base ad una probabilità p_m. Vi sono diversi tipi di operatori di mutazione a seconda del tipo di rappresentazione genetica impiegata: ad esempio, nel caso del codice binario, gli elementi selezionati per essere mutati modificano il loro stato da zero a uno o viceversa (Figura 1.3); nel caso di un codice composto da numeri reali, invece, gli elementi selezionati assumono un nuovo valore estratto in modo casuale da un certo intervallo.
La teoria tradizionale ritiene che il crossover sia più importane della mutazione per quanto riguarda la rapidità nell'esplorare lo spazio di ricerca; la mutazione porta un po' di 'casualità' in essa in modo che nessun punto nello spazio abbia probabilità nulla di essere esaminato.

Figura 1.3: *Un esempio di mutazione nella codifica binaria.*

La nuova popolazione di stringhe genetiche rimpiazza parzialmente o completamente le vecchie stringhe. Il processo di decodifica, valutazione, riproduzione selettiva, incrocio e mutazione riprende ciclicamente per parecchie generazioni fino a quando viene ottenuta una stringa che codifica una soluzione soddisfacente.
Se l'algoritmo genetico è correttamente implementato, la popolazione evolverà nel susseguirsi delle generazioni in modo che il fitness del miglior individuo e la media in ogni generazione cresca verso l'ottimo globale. La convergenza è la progressione verso la crescente uniformità. Si dice che un gene converge quando il 95% della popolazione condivide lo stesso valore. La popolazione converge quando tutti i geni convergono.

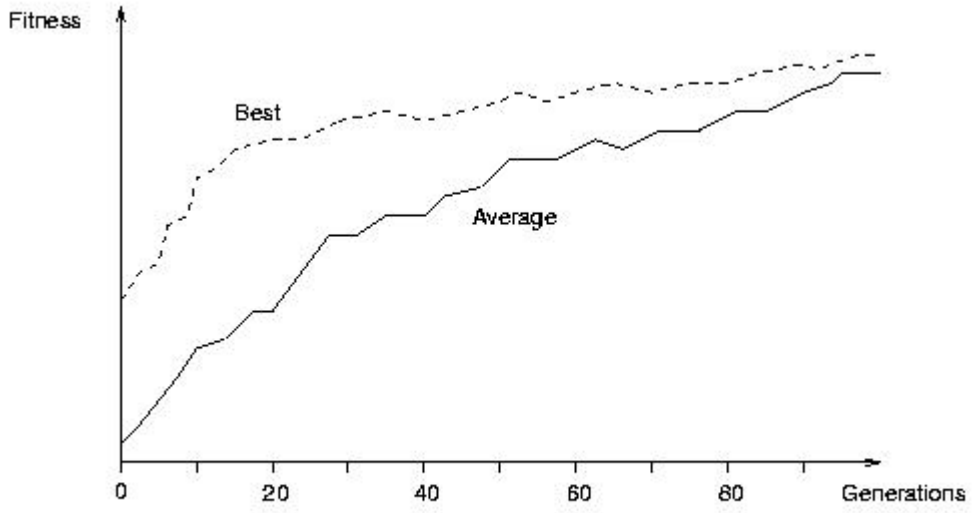

Figura 1.4: *Comportamento di un algoritmo genetico correttamente implementato (convergenza).*

1.2 Un primo esempio

In questa sezione osserveremo il funzionamento di un algoritmo genetico in una situazione molto semplice[2]: il problema consiste nel trovare i valori della variabile x che massimizzano la funzione potenza $\Phi(x) = x^2$, dove x può variare tra 0 e 31.

Il primo passo consiste nella scelta della rappresentazione genetica: utilizziamo un codice binario e stringhe di lunghezza 5 che ci permettono di codificare numeri (interi) da 0 ('00000') a 31 ('11111')[3].

Creiamo dunque una piccola popolazione composta solamente da quattro stringhe generate casualmente. Ciascuna stringa viene decodificata e il suo valore

[2]L'esempio è tratto da Goldberg, D.E. 'Genetic Algorithms in Search, Optimization and Machine Learning' (1989), Reading, MA, Addison-Wesley.

[3]Il processo di trasformazione da numero binario a numero decimale è dato da $x = \sum_{p=0}^{N-1} i \cdot 2^p$ dove p indica la posizione del numero intero binario a partire dall'ultima cifra (la numero zero), N è il numero di cifre e i è il valore binario dell'intero corrispondente (zero o uno).
Ad esempio '10100' = 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = 16 + 4 = 20.

di fitness viene calcolato applicando la funzione Φ al numero decimale ottenuto (Tabella 1.1).

Num. Stringa	**Popolaz. iniziale**	**Decod. x**	**Valutaz.** $F(x)=x^2$	**Prob. di riprod.** $P_i=\frac{f_i}{\sum_i^N f_i}$	**Num. figli**	**Num. estraz.**
1	01101	13	169	0,14	0,58	1
2	11000	24	576	0,49	1,97	2
3	01000	8	64	0,06	0,22	0
4	10011	19	361	0,31	1,23	1
somma			1170	1,00	4,00	4
media			293	0,25	1,00	1
massimo			576	0,49	1,97	2

Tabella 1.1: Popolazione iniziale

Alla fine del processo di valutazione ciascuna stringa riceve una probabilità di riproduzione proporzionale alla propria fitness e il generatore di numeri casuali assegna un numero di copie a ciascuna di esse in base a questa probabilità. Osserviamo nel frattempo due importanti indici evolutivi: il valore medio ed il valore massimo della fitness. Siccome una popolazione è sempre composta da individui che ottengono prestazioni diverse, il valore medio è sempre più basso del valore massimo.
Le nuove stringhe vengono poi accoppiate e per ciascuna di esse (qui la probabilità di crossover $p_c = 1$) viene scelto un punto d'incrocio casuale attorno al quale esse si scambiano parte del loro materiale.
Infine ciascun elemento viene mutato con probabilità $p_m = 0{,}001$: in questo caso nessun elemento subisce modifiche. La nuova popolazione di cromosomi così ottenuta viene nuovamente decodificata e valutata: sia il valore medio che il valore massimo di fitness sono superiori al valore della generazione iniziale (Tabella 1.2).

Se osserviamo il risultato dell'operatore di crossover, notiamo che lo scambio casuale non produce necessariamente stringhe ottimali: in questo caso per entrambe le coppie viene generata una stringa con prestazioni uguali o addirittura peggiori della media della generazione precedente (fitness 246 e 144), ma

Accoppiam. con crossover (!)	**Incrocio** p_c=1,0 (tutti)	**Mutazioni** p_m=0,001 (nessuna)	**Nuova popolaz.**	**Decod. x**	**Valutaz.** $\Phi(x) = x^2$
0110!1	01100	01100	01100	12	144
1100!0	11001	11001	11001	25	625
11!000	11011	11011	11011	27	729
10!011	10000	10000	10000	16	256
somma					1754
media					439
massimo					729

Tabella 1.2: Prima generazione

vengono generate anche stringhe molto migliori (fitness 625 e 729). Il processo di riproduzione selettiva fa sì che mediamente solo le stringhe migliori contribuiscano alla creazione della nuova popolazione.

Capitolo 2

La Teoria degli schemi

La teoria tradizionale degli Algoritmi Genetici, formulata da Holland a partire dal 1975[1], ipotizza che essi funzionino scoprendo, favorendo e ricombinando in modo altamente parallelo buoni 'blocchi costitutivi' di soluzioni, cioè combinazioni di bit che conferiscono maggiore capacità alle stringhe in cui si presentano. Holland introdusse la nozione di schema per formalizzare la nozione informale di blocco costitutivo e, quindi, per analizzare il funzionamento del processo evolutivo.
Uno 'schema' è una stringa i cui possibili elementi sono quelli dell'alfabeto genetico utilizzato per la codifica, con l'aggiunta del simbolo '*', che ha il ruolo di carattere jolly (ossia indica la presenza di un qualsiasi simbolo appartenente all'alfabeto genetico).
Uno schema rappresenta quindi un insieme di stringhe genetiche. Ad esempio, se si utilizza il codice binario come alfabeto genetico, lo schema '1111*' rappresenta la stringa '11111' e '11110'; lo schema '1***0' rappresenta invece le otto stringhe che possiedono un '1' iniziale ed uno '0' finale.
Gli schemi sono uno strumento utile per analizzare le informazioni contenute in una popolazione di stringhe e osservarne le somiglianze. Se consideriamo le due popolazioni dell'esempio riportato nel paragrafo precedente notiamo che le stringhe migliori appartengono allo schema '11***' (la stringa '11000' della prima generazione e le stringhe '11001' e '11011' della seconda generazione): la ricerca degli schemi ottimali ci permette di individuare efficientemente l'informazione importante presente nel codice genetico della popolazione di individui.

[1]Si legga al proposito 'Adaptation in Natural and Artificial Systems' di Holland J.H., University of Michigan Press, Ann Arbor, 1975.

Gli schemi sono anche uno strumento molto efficace per esplorare lo spazio delle possibili soluzioni del problema che vogliamo risolvere. Una stringa binaria composta da l elementi appartiene a 2^l schemi diversi (siccome ciascun elemento può assumere due diversi stati): quindi, una popolazione di N stringhe contiene un numero di schemi compreso tra 2^l (se tutte le stringhe sono identiche) e $N*2^l$ (se sono tutte diverse fra di loro). Questo significa che, a una data generazione, mentre l'algoritmo genetico valuta esplicitamente l'idoneità delle n stringhe della popolazione, in realtà sta stimando implicitamente l'idoneità media di un numero assai maggiore di schemi, dove l'idoneità media di uno schema è definita come la media delle idoneità di tutte le possibili istanze di quello schema.
Per esempio, in una popolazione di n stringhe generate casualmente, mediamente la metà delle stringhe sarà istanza di (1,*,*, ..., *,*) e metà sarà istanza di (0,*,*, ..., *,*). La valutazione delle circa n/2 stringhe che sono istanze di (1,*,*, ..., *,*) fornisce una stima dell'idoneità media di quello schema; si parla di stima perché le istanze valutate in una popolazione di dimensioni tipiche, sono solo un piccolo campione di tutte le possibili istanze. Così come gli schemi non sono rappresentati o valutati esplicitamente dall'algoritmo, anche le stime dell'idoneità media degli schemi non sono calcolate o memorizzate in modo esplicito. Tuttavia, il comportamento degli algoritmi genetici in termini di aumento o diminuzione del numero di istanze di un dato schema nella popolazione, può essere descritto come se queste medie fossero effettivamente calcolate e memorizzate.
Quindi da un lato gli schemi permettono di isolare i componenti importanti per la soluzione del problema e dall'altro lato permettono di esplorare un numero di potenziali soluzioni molto più grande del numero di stringhe della popolazione. Il processo di riproduzione selettiva fa sì che nel corso delle generazioni venga allocato un numero sempre maggiore di esemplari di schemi ad alta fitness: nell'esempio del paragrafo precedente notiamo infatti che lo schema '11***' ha duplicato la sua presenza alla seconda generazione.
La teoria degli schemi dimostra per l'appunto che schemi caratterizzati da una 'lunghezza significativa' corta e da un alto valore di fitness vengono propagati attraverso le generazioni per mezzo della creazione di un numero di esemplari crescente in modo esponenziale sulla base della semplice osservazione delle

stringhe migliori. Un ulteriore risultato della teoria[2] dimostra che gli algoritmi genetici vagliano ad ogni generazione l'equivalente di almeno N^3 schemi diversi, benché vengano valutate solamente N stringhe: questa caratteristica viene definita con il nome di **parallelismo implicito** degli algoritmi genetici.
Tuttavia non tutti gli schemi possibili vengono elaborati e trasmessi con egual efficienza dall'algoritmo genetico: a parità di fitness, gli schemi caratterizzati da elementi significativi molto distanti fra di loro (ad esempio '1***1') possiedono una maggior probabilità di essere spezzati dall'operatore di crossover, mentre gli schemi caratterizzati da un piccolo numero di elementi significativi raggruppati in posizioni contigue (ad esempio '*11**') tendono ad essere trasmessi e riprodotti con maggior frequenza.
A tal proposito introduciamo alcune definizioni legate alle proprietà degli schemi.

2.1 Il processo evolutivo in dettaglio

L'**ordine** di uno schema S, denotato con $o(S)$, è il numero di posizioni definite, ovvero (nel caso del codice binario) il numero di 0 e di 1 presenti.

$$o(S) = \text{(lunghezza totale dello schema) - (numero di caratteri jolly)}$$

L'ordine definisce la specificità di uno schema ed è utile per calcolare la probabilità di sopravvivenza di uno schema alla mutazione.

La **lunghezza di definizione** dello schema S, denotata con $\delta(S)$, è la distanza tra la prima e l'ultima posizione fissata nella stringa. Essa definisce la compattezza delle informazioni contenute in uno schema. Si osservi che uno schema con una sola posizione fissata ha lunghezza di definizione uguale a zero. A titolo di esempio si consideri la lunghezza di definizione e l'ordine dei seguenti schemi di dieci bits:

[2]Goldberg, D.E. 'Genetic Algorithms in Search, Optimization and Machine Learning' (1989), Reading, MA, Addison-Wesley.

i	$\mathbf{S}_i$	$\mathbf{o(S}_i\mathbf{)}$	$\delta(\mathbf{S}_i)$
1	***001*110	6	6
2	****00**0*	3	4
3	11101**001	8	9

Tabella 2.1: Ordine e Lunghezza di definizione di uno schema.

Consideriamo ora un certo tempo t (corrispondente alla *t-esima* generazione) e cerchiamo di analizzare più approfonditamente il meccanismo di passaggio da una generazione alla successiva.
Denotiamo con $\xi(S,t)$ il numero di stringhe in una popolazione al tempo t che siano rappresentate dallo schema S. Inoltre possiamo definire *eval(S,t)* come la fitness media di tutte le stringhe della popolazione che sono istanza dello schema S al tempo t.
Supponiamo che ci siano p stringhe $\{v_{s,1}, ..., v_{s,p}\}$ nella popolazione, istanze dello schema S al tempo t. Allora:

$$eval(S,t) \;=\; \sum_{j=1}^{p} \frac{eval(v_{s,j})}{p}$$

Durante il passaggio di selezione viene creata una popolazione intermedia: ogni stringa verrà copiata zero, una o più volte, in base alla propria idoneità. Ciascuna stringa ha probabilità $p_i = \frac{eval(v_i)}{F(t)}$ di essere selezionata, con *F(t)* fitness totale dell'intera popolazione al tempo t, cioé $F(t) = \sum_{i=1}^{n} eval(v_i)$, con n grandezza della popolazione. Notiamo anche che per ogni stringa istanza dello schema S la probabilità della sua selezione è (in media) $\frac{eval(S,t)}{F(t)}$.
Dopo la selezione ci aspettiamo di avere $\xi(S,t+1)$ stringhe istanza dello schema S. Il numero atteso di stringhe istanza di S al tempo *t+1*, è pari al numero di rappresentanti di S al tempo t, per il numero di stringhe da selezionare, per la probabilità media che sia selezionata una stringa istanza di S; ovvero:

$$\xi(S,t+1) \;=\; \xi(S,t)\cdot n \cdot \frac{eval(S,t)}{F(t)}$$

Possiamo riscrivere la formula precedente tenendo conto che l'idoneità media della popolazione è $\overline{F(t)} = \frac{F(t)}{n}$. Quindi

$$\xi(S,t+1) \;=\; \xi(S,t)\cdot \frac{eval(S,t)}{\overline{F(t)}} \tag{2.1}$$

La (2.1) rappresenta l'equazione della crescita riproduttiva dello schema. In altre parole, il numero di stringhe nella popolazione cresce come il rapporto tra l'idoneità dello schema e l'idoneità media della popolazione. Questo significa che uno schema 'sopra la media' avrà un maggior numero di rappresentanti nella generazione successiva, mentre un schema 'sotto la media' subirà un decremento del numero delle stringhe.
Gli effetti di questa legge sono chiari: se pensiamo ad esempio che lo schema S sia superiore alla media, in percentuale, di ϵ, cioè:

$$eval(S,t) \;=\; \overline{F(t)} \;+\; \epsilon \cdot \overline{F(t)}$$

allora, facendo le opportune sostituzioni, otteniamo:

$$\xi(S,t) \;=\; \xi(S,0) \cdot (1+\epsilon)^t$$

Questa equazione è una progressione geometrica; di conseguenza possiamo affermare non solo che uno schema sopra la media riceve un incremento di presenze nella generazione successiva, ma anche che questo incremento è esponenziale.

La semplice selezione però non introduce nessun nuovo punto dello spazio di ricerca: la seconda fase del processo evolutivo, la ricombinazione, ha in se la responsabilità di introdurre nuovi individui nella popolazione. Questo avviene mediante due operatori genetici: incrocio e mutazione.
È importante discutere gli effetti di questi due operatori rispetto al numero atteso di schemi nella popolazione.

Per quel che riguarda l'incrocio, possiamo facilmente notare come esso sia responsabile della distruzione o sopravvivenza degli schemi presenti in una stringa genetica. Prendiamo per esempio la stringa $v_1 = (100010)$: essa è istanza degli schemi $S_0 = (100{*}{*}{*})$ e $S_1 = (10{*}{*}10)$. Supponiamo che v sia selezionata per l'incrocio con la stringa $v_2 = (010100)$ con 'punto di taglio' uguale a tre. La situazione sarà dunque quella presentata nella tabella (2.2).

Notiamo che lo schema S_0 sopravvive all'incrocio, mentre lo schema S_1 viene distrutto: nessun discendente ne è istanza perché il punto di taglio divide

Genitori	Figli
$v_1 = (100\,010)$	$v_1' = (100\,100)$
$v_2 = (010\,100)$	$v_2' = (010\,010)$

Tabella 2.2: Riproduzione e incrocio.

in due le posizioni fissate.
Dovrebbe essere chiaro che la 'lunghezza di definizione' di uno schema gioca un ruolo considerevole nella probabilità di essere distrutto o di sopravvivere. La lunghezza di definizione dello schema S_0 è $\delta(S_0) = 2$ mentre la lunghezza di definizione dello schema S_1 è $\delta(S_1) = 5$.
In generale, in una stringa di lunghezza l, la posizione di incrocio è selezionata uniformemente tra le $l-1$ possibili posizioni. Questo implica che la probabilità di distruzione di uno schema è:

$$p_d(S) \;=\; \frac{\delta(S)}{l-1}$$

Di conseguenza la probabilità di sopravvivenza è:

$$p_s(S) \;=\; 1 - \frac{\delta(S)}{l-1}$$

Per l'esempio considerato le probabilità di distruzione sono $p_d(S_0) = 2/5 = 0.4$ e $p_d(S_1) = 5/5 = 1$, in accordo con quanto osservato.
È importante osservare che solo alcuni cromosomi subiscono il processo di incrocio, con probabilità selettiva di incrocio p_c. Questo si ripercuote sulle probabilità di distruzione e di sopravvivenza di uno schema:

$$p_d(S) \;=\; p_c \cdot \frac{\delta(S)}{l-1}$$

Di conseguenza la probabilità di sopravvivenza è:

$$p_s(S) \;=\; 1 - p_c \cdot \frac{\delta(S)}{l-1}$$

Bisogna inoltre tener conto che, anche se la posizione di incrocio è fissata in modo tale da rompere uno schema, c'è ancora una possibilità di sopravvivenza: se, per esempio, la stringa v_2 fosse terminata con '10' allora lo schema S_1 sarebbe comunque sopravvissuto all'incrocio (Tabella 2.3).

Genitori	Figli
$v_1 = (100\,010)$	$v_1' = (100\,110)$
$v_2 = (010\,110)$	$v_2' = (010\,010)$

Tabella 2.3: Riproduzione e incrocio: esempio modificato.

Quindi potremmo modificare nuovamente la formula per la probabilità di sopravvivenza di uno schema:

$$p_s(S) \;\geq\; 1 - p_c \cdot \frac{\delta(S)}{l-1}$$

Così gli effetti combinati della selezione e dell'incrocio ci danno una nuova forma dell'equazione di crescita dello schema riproduttivo.

$$\xi(S, t+1) \;\geq\; \xi(S,t) \;\cdot\; \frac{eval(S,t)}{\overline{F(t)}} \;\cdot\; \left[1 - p_c \cdot \frac{\delta(S)}{l-1}\right] \tag{2.2}$$

L'equazione (2.2) indica il numero atteso di stringhe istanza dello schema S nella generazione successiva, come funzione del numero attuale di stringhe istanza dello schema, dell'idoneità relativa allo schema e della sua lunghezza di definizione.

Il successivo operatore da considerare è la mutazione.
L'operatore di mutazione modifica casualmente una singola posizione in un cromosoma con probabilità p_m. È chiaro che se tutte le posizioni fissate di uno schema rimangono invariate lo schema sopravvive alla mutazione. Dunque il numero di bit 'importanti' è pari all'ordine dello schema, cioè al numero di posizioni fissate.
Poiché la probabilità di mutazione di un singolo bit è p_m, la probabilità di sopravvivenza di ogni bit è $(1-p_m)$; inoltre ogni singola mutazione è indipendente dalle altre mutazioni. Quindi la probabilità di uno schema di sopravvivere alla mutazione è:

$$p_s(S) \;=\; (1-p_m)^{o(S)}$$

Gli effetti combinati di selezione, incrocio e mutazione ci danno una nuova forma dell'equazione di crescita dello schema riproduttivo:

$$\xi(S, t+1) \;\geq\; \xi(S,t) \;\cdot\; \frac{eval(S,t)}{\overline{F(t)}} \;\cdot\; \left[1 - p_c \cdot \frac{\delta(S)}{l-1}\right] \;\cdot\; (1-p_m)^{o(S)} \tag{2.3}$$

Questa equazione ci fornisce il numero atteso di stringhe istanze dello schema S nella generazione $t+1$ come funzione del numero di stringhe istanze dello schema nella t-esima generazione, dell'idoneità relativa dello schema, della sua lunghezza di definizione e dell'ordine.
È chiaro che schemi con lunghezza di definizione breve e di basso ordine avranno un ritmo di crescita esponenziale.

È bene osservare che l'equazione (2.3) è basata sull'ipotesi che la funzione di idoneità restituisca solo valori positivi; quando si applica un algoritmo genetico ad un problema di ottimizzazione dove la funzione ottimizzatrice può restituire valori negativi, bisogna fare qualche considerazione in più.
Dunque l'equazione di crescita (2.3) mostra che la selezione incrementa il numero delle istanze di uno schema sopra la media in modo esponenziale, ma questo primo passaggio non introduce nessun nuovo schema, fatto che può avvenire solo grazie agli operatori di mutazione e incrocio. In particolare l'operatore di mutazione introduce grande variabilità nella popolazione. Gli effetti combinati di questi tre operatori per uno schema non sono distruttivi se lo schema è corto e di basso ordine.

Il risultato finale dell'equazione di crescita (2.3) può essere enunciato come Teorema degli Schemi.

2.2 Teorema degli Schemi

'Gli schemi piccoli, di basso ordine e con fitness superiore alla media subiscono una crescita esponenziale nelle generazioni successive di un algoritmo genetico.'

Il Teorema degli Schemi espresso nella forma dell'equazione (2.3) è un limite inferiore, poiché contempla esclusivamente gli effetti distruttivi di incrocio e mutazione. Tuttavia, l'incrocio è considerato una delle principali fonti della potenza di un algoritmo genetico, per la sua capacità di ricombinare istanze di buoni schemi per formare istanze di schemi di ordine superiore altrettanto

buoni o migliori.

La supposizione che questo sia il processo mediante il quale gli algoritmi genetici operano è nota con il nome di 'Ipotesi dei Blocchi Costitutivi' o 'Building Block Hypothesis'[3].

2.3 Ipotesi dei blocchi costitutivi

'Un algoritmo genetico ottiene delle prestazioni vicine all'ottimo attraverso la contrapposizione di schemi piccoli, di basso ordine e con alte prestazioni, chiamati blocchi costitutivi.'

Sebbene siano state condotte alcune ricerche per provare queste ipotesi[4], per alcune applicazioni non banali si fa ancora affidamento principalmente su risultati empirici.
Durante gli ultimi vent'anni sono state sviluppate molte applicazioni degli algoritmi genetici che supportavano le ipotesi dei blocchi costitutivi in problemi riguardanti campi molto diversi. Tuttavia queste ipotesi mostrano come il problema della codifica per un algoritmo genetico sia cruciale per la sua riuscita finale, e che tale codifica dovrebbe soddisfare l'idea dei blocchi costitutivi corti.
L'elaborazione di schemi di lunghezza significativa ridotta e ad alta fitness riduce notevolmente la complessità della ricerca di una soluzione: invece di cercare di costruire stringhe che possiedano un'alta fitness provando tutte le possibili combinazioni degli elementi disponibili, gli algoritmi genetici procedono gradualmente combinando le migliori soluzioni parziali degli individui precedenti.
Gli schemi rappresentano quindi i mattoni elementari usati dall'evoluzione per la costruzione di complesse strutture. Affinché la procedura di isolamento e combinazione degli schemi risulti in un graduale miglioramento delle soluzioni

[3]Si legga a tal proposito 'Genetic Algorithms in Search, Optimization and Machine Learning' di D.E. Goldberg, Addison-Wesley, 1989 oppure 'Crossover or Mutation ?' di W.M. Spears in 'Whitley, L. D. - Foundations of Genetic Algorithms 2', Morgan Kaufmann Pubblishers, 1993.

[4]Si legga 'Genetic Algorithms as Function Optimizers' di A.D. Bethke, Doctoral Dissertation, University of Michigan, 1980.

offerte è importante che la rappresentazione genetica sia appropriata al problema da risolvere, perché rappresentazioni inadeguate aumentano la complessità e la lunghezza significativa degli schemi necessari a codificare una soluzione accettabile. Vi sono due regole di base per scegliere una rappresentazione adeguata: la rappresentazione dovrebbe permettere la presenza di schemi rilevanti e corti, e dovrebbe basarsi su di un alfabeto ristretto che permetta un'espressione naturale del problema.
Inoltre, uno schema di codifica ben riuscita è uno schema che incoraggia la formazione di building block, assicurandosi che:

- i geni correlati siano vicini all'interno del cromosoma;

- ci sia poca interazione tra i geni.

Per 'interazione tra geni', o *epistasis*, si intende che il contributo di un gene al fitness dipende dal valore degli altri geni nel cromosoma.
Se queste regole sono rispettate, l'algoritmo genetico sarà efficiente come predetto dal teorema degli schemi.
Sfortunatamente le due condizioni sono difficili da incontrare e spesso i geni possono essere correlati in modo che non sia possibile metterli vicino in una stringa monodimensionale (per esempio se sono collegati gerarchicamente). In molti casi, l'esatta natura del legame tra i geni può non essere conosciuto dal programmatore, così anche se sono relazioni semplici, può essere impossibile arrangiare la codifica per mostrarle.
Da qui nascono due domande: se non è possibile rispettare le due regole precedenti, può un algoritmo genetico essere modificato in modo da migliorare il suo funzionamento ? Se sì, come ? Queste domande sono entrambe argomento di ricerca.

Il Teorema degli Schemi e l'Ipotesi dei Blocchi Costitutivi si occupano principalmente del ruolo della selezione e dell'incrocio negli algoritmi genetici. Qual è il ruolo della mutazione? Secondo Holland[5] la mutazione è ciò che impedisce la perdita di diversità in una posizione data. Per esempio, senza la mutazione,

[5] 'Adaptation in Natural and Artificial Systems' di Holland J.H., University of Michigan Press, Ann Arbor, 1975.

potrebbe capitare che ogni stringa di una certa popolazione con codifica binaria abbia un '1' nella prima posizione. Di conseguenza non ci sarebbe modo di ottenere una stringa che inizi con uno '0'. La mutazione fornisce una 'polizza di assicurazione' nei confronti di un tale irrigidimento.

Capitolo 3

Aspetti pratici

La versione di base degli operatori genetici descritta nei paragrafi precedenti è sufficiente per produrre buoni risultati nella maggior parte dei casi. Tuttavia decenni di teoria e sperimentazione nel campo dell'evoluzione artificiale hanno prodotto molteplici variazioni sui diversi componenti ed hanno aperto interessanti dibattiti. In questo capitolo prenderemo in esame gli aspetti principali del problema.

Un punto molto importante è che gli algoritmi genetici operano su elementi casuali e utilizzano processi di transizione probabilistici: è quindi buona norma ripetere ogni esperimento più volte iniziando da popolazioni casuali diverse per assicurarsi la stabilità della soluzione raggiunta. Quanto minore è la dimensione della popolazione, tanto maggiore è il rischio di non riuscire a replicare i dati oppure di non riuscire ad ottenere una soluzione soddisfacente.

Il corretto funzionamento degli algoritmi genetici si basa sulla diversità dei cromosomi della popolazione. La mancanza di diversità corrisponde ad una stagnazione che non è desiderabile né durante il processo evolutivo, né quando le fitness media e massima della popolazione si sono stabilizzate: anche in questo ultimo caso potrebbero sempre verificarsi delle situazioni per cui potrebbe tornare utile continuare il processo evolutivo al fine di individuare nuove soluzioni.

Inizialmente la diversità è assicurata dalla generazione casuale delle stringhe. Nel corso delle generazioni il processo di riproduzione selettiva tende a ridurre la diversità della popolazione e, anche se il processo di selezione fosse completamente casuale, dopo qualche generazione tutti i cromosomi si assomiglierebbero

(a causa del fenomeno della 'deriva genetica', che verrà spiegato successivamente). L'operatore crossover tende a contrastare la riduzione della diversità creando nuove strutture dalla combinazione di parti delle stringhe esistenti, ma esso non è sufficiente in tutti i casi.
Immaginiamo ad esempio che un determinato 'allele' (il valore specifico di una determinata caratteristica genetica, ad esempio il valore '1' in terza posizione nelle stringhe di cinque elementi considerate nell'esempio del paragrafo 1.2) scompaia dalla popolazione perché nessuno degli individui che lo possedevano è stato in grado di riprodursi: in questo caso il meccanismo di crossover non è in grado di ricreare l'allele mancante, ma potrebbe riuscirci l'operatore di mutazione. Nell'interpretazione tradizionale degli algoritmi genetici le mutazioni sono essenzialmente un modo per mantenere un certo grado di diversità nella popolazione. Tuttavia, siccome l'operatore di mutazione opera ciecamente, senza rispettare l'ingegnoso processo di elaborazione degli schemi, la maggior parte dei ricercatori tende ad utilizzare una probabilità di mutazione molto bassa (valori inferiori all'1%). Un altro metodo diffuso per mantenere la diversità consiste nell'introdurre ad ogni generazione nuove stringhe casuali nella popolazione, sostituendole a quelle più scadenti.
Cercheremo ora di analizzare le principali variazioni legati alle differenti componenti degli algoritmi genetici.

3.1 Scelta della codifica

Un cromosoma è una sequenza di simboli che identifica una possibile soluzione al problema affrontato. Generalmente l'alfabeto scelto per la codifica è il codice binario, ma questa non è l'unica possibilità.
Alfabeti con alte cardinalità sono stati usati in varie ricerche e alcuni ritengono che siano vantaggiosi. Secondo Goldberg la rappresentazione binaria dà il più grande numero di schemi, e fornisce il più alto grado di parallelismo implicito, mentre Antonisse interpreta gli schemi differentemente e conclude che gli alfabeti con alte cardinalità contengono più schemi di quelli binari (la questione non è stata ancora risolta).
Studi empirici sugli alfabeti con alte cardinalità hanno usato cromosomi dove

ciascun simbolo rappresenta un intero o un floating point: infatti, poiché i parametri del problema sono spesso numerici, rappresentare i geni direttamente come numeri, anziché come cifre binarie, può essere un vantaggio.
Di conseguenza possiamo definire in modo più naturale anche gli operatori di crossover e mutazione.
Alcuni esempi maggiormente utilizzati di operatori di combinazione sono:

- la media (si prende la media aritmetica dei geni dei due genitori);
- la media geometrica (la radice quadrata del prodotto dei due valori);
- l'extension (si prende la differenza tra due valori e la si aggiunge al più alto o la si sottrae al più basso).

Per quel che riguarda gli operatori di mutazione, gli esempi più importanti sono:

- il rimpiazzamento casuale (si rimpiazza un valore con uno casuale);
- il 'creep' (si aggiunge o sottrae un piccolo numero generato casualmente al gene);
- il 'geometric creep' (si moltiplica il gene per un valore prossimo a 1).

Per entrambi gli operatori creep il numero generato casualmente può avere diverse distribuzioni: uniforme dentro un dato range, esponenziale, gaussiana, binomiale o altro.

3.1.1 Codifiche ad hoc

Un esempio di codifica caratteristica per un determinato problema è la 'Codifica per permutazione'. Essa può essere usata in problemi di ordinamento, come il problema del commesso viaggiatore (a cui è dedicato il capitolo 6).
Con questa codifica, tutti i cromosomi sono una stringa di numeri: ogni cromosoma è una particolare permutazione della stringa originale.

L'idea principale è che la codifica (e di conseguenza gli operatori ad essa legati) debba rappresentare in maniera naturale il problema e le sue possibili soluzioni. Una conseguenza di questa convinzione è la 'codifica per valore'.
Una codifica col valore diretto può essere utilizzata in problemi dove sono pre-

Stringa Originale	(1,2,3,4,5,6,7,8,9)
Cromosoma A	(1,5,3,2,6,4,7,9,8)
Cromosoma B	(8,5,6,7,2,3,1,4,9)

Tabella 3.1: Esempio di codifica per permutazione.

senti valori complicati come i numeri reali. L'uso di un alfabeto binario per questo tipo di problemi sarebbe molto difficile.
In questa codifica, ogni cromosoma è una stringa di alcuni valori. I valori possono essere qualsiasi cosa correlata al problema: interi, numeri reali, caratteri o oggetti complicati.

Cromosoma A	(1.2324; 5.3243; 0.4556; 2.3293; 2.4545)
Cromosoma B	ABDJEIFJDHDIERJFDLDFLFEGT
Cromosoma C	*(back), (back), (right), (forward), (left)*

Tabella 3.2: Esempi di codifica per valore.

La codifica per valore è particolarmente indicata per alcuni problemi speciali. D'altra parte, per questa codifica è spesso necessario costruire nuovi operatori crossover e mutazione specifici per il problema.

3.1.2 Mapping dei valori ridondanti

Quando un gene può assumere un numero finito di valori validi discreti, possono verificarsi dei problemi: se per esempio si usa una rappresentazione binaria e il numero dei valori non è una potenza di due, allora alcuni dei codici binari sono ridondanti (ovvero non corrispondono a nessun valore valido del gene). Per esempio se un gene rappresenta un oggetto selezionato tra un gruppo di dieci oggetti, allora quattro bit saranno necessari per codificare il gene ('1001' in codice binario è equivalente a '9' in codice decimale).
Durante il crossover e la mutazione non possiamo garantire che i codici ridondanti non si manifestino (nel nostro caso, per esempio '1111' equivale a '15', valore decimale che non rappresenterebbe alcun oggetto). Il problema è cosa farne. Questo problema non è stato studiato in maniera molto approfondita perché gli studiosi si sono maggiormente concentrati su funzioni continue dove

questo problema non si manifesta.
Comunque le principali soluzioni proposte, sono:

• scartare il cromosoma come illegale (e quindi generarne uno nuovo);
• assegnare al cromosoma un basso fitness;
• mappare un codice invalido verso uno valido (cioè assegnare a un cromosoma non valido il valore di quello valido che gli sta più vicino).

Le prime due soluzioni daranno molto probabilmente risultati insoddisfacenti perché rischiamo di scartare cromosomi con geni interessanti. Ci sono buoni metodi per sviluppare la terza, come il rimappamento fisso e il rimappamento casuale.
Nel primo un particolare valore ridondante è rimappato verso un valore valido specifico. È una soluzione molto semplice, ma ha lo svantaggio che alcuni valori sono rappresentati da due modelli con bit differenti, mentre gli altri sono rappresentati da uno solo (nell'esempio precedente, i codici per i numeri da 10 a 15 possono essere rimappati per i valori da 0 a 5, cosicché questi valori sono rappresentati due volte, mentre quelli da 6 a 9 hanno una rappresentazione singola). Di conseguenza si avrà uno sbilanciamento delle probabilità in favore dei valori rappresentati da due cromosomi differenti.
Nella 'random rimapping' un valore ridondante è rimappato verso un valore valido casualmente. Questo evita il problema rappresentativo, ma passa meno informazioni ai figli.
Il rimappamento probabilistico è un ibrido tra queste due tecniche. Tutti i valori dei geni (non solo quelli in eccesso) sono rimappati verso uno dei valori validi in maniera probabilistica, come se ciascun valore valido avesse uguale probabilità di essere rappresentato.
Vi sono inoltre numerosi altri approcci che portano a risultati validi: un esempio è il suggerimento di Cramer riguardo ai problemi del 'tutto o niente' (ossia di problemi di ottimizzazione combinatoria con molti vincoli in cui si cerca una soluzione accettabile utilizzando soluzioni non valide) che verrà descritto più avanti.

3.2 Funzione Fitness e Selezione

L'operatore di riproduzione selettiva svolge il ruolo di guida del processo evolutivo: esso assegna ad ogni stringa genetica un numero di copie grosso modo proporzionale alla sua fitness. Il meccanismo di riproduzione interagisce dunque strettamente con la funzione fitness.
Ora descriveremo alcuni metodi per selezionare due individui per l'accoppiamento; per capire i motivi che portano allo sviluppo di queste tecniche, descriveremo prima i problemi che esse cercano di superare: questi problemi sono legati alla funzione fitness, di cui ora parleremo più in dettaglio.

3.2.1 Funzione Fitness

Insieme allo schema di codice usato, la funzione fitness è l'aspetto cruciale di ogni algoritmo genetico. Molta ricerca si è concentrata sull'ottimizzazione di tutte le parti degli algoritmi genetici, poiché i miglioramenti possono essere applicati a una varietà di problemi. Frequentemente, comunque, si è trovato che può essere ottenuto solo un piccolo miglioramento del comportamento. Grefenstette ha indicato un insieme di parametri ottimali (crossover, mutazione, popolazione e altri), ma ha concluso che i meccanismi di base di un algoritmo genetico sono così robusti che, entro margini abbastanza grandi, i parametri non sono critici. Quello che è critico è invece la funzione fitness e lo schema di codifica usato.
Idealmente, vogliamo che la funzione fitness sia piatta e regolare, cosicché i cromosomi con fitness ragionevole siano vicini (nello spazio dei parametri) ai cromosomi con fitness leggermente migliore. Per molti problemi di interesse, purtroppo, non è possibile costruirla con questo andamento (se fosse possibile converrebbe forse usare il metodo 'hillclimb'). Inoltre, perché gli algoritmi genetici funzionino bene, dovremmo trovare il modo per costruire funzioni fitness che non abbiano troppi massimi locali o massimi locali troppo isolati.
La regola generale per costruire la funzione fitness, è che essa dovrebbe riflettere il valore reale del cromosoma in una qualche maniera. Come detto sopra, per molti problemi la costruzione può essere un passo ovvio, ma il valore reale di un cromosoma non sempre è una quantità utile per guidarci nella ricerca

genetica.
In problemi di ottimizzazione combinatoria, ci sono molti vincoli e molti punti nello spazio rappresentano cromosomi non validi, e perciò hanno valore 'reale' nullo. Un esempio è la stesura di un orario scolastico. A un certo numero di classi deve venire assegnato un certo numero di lezioni, avendo un numero finito di aule e lettori. Molte assegnazioni delle classi e dei lettori alle aule saranno violate, ad esempio se un lettore è stato messo in due posti contemporaneamente o se ad una classe non sono state assegnate tutte le lezioni che dovrebbe svolgere.
Perché un algoritmo genetico sia efficace in questo caso, dobbiamo inventare una funzione dove il fitness di un cromosoma è visto in funzione di quanto è in grado di portarci vicino ad un cromosoma valido. Dobbiamo sapere dove si trovano i cromosomi validi per assicurarci che ai punti vicini può essere assegnato un buon valore di fitness, ma se non sappiamo dove sono i cromosomi validi, ciò non può essere fatto.
Come precedentemente accennato, Cramer ha suggerito che, se il naturale obbiettivo di un problema è 'tutto o niente', risultati migliori possono essere ottenuti se inventiamo sotto-obiettivi significativi, e li ricompensiamo. Nel problema dell'orario, per esempio, potremmo dare un compenso ad ogni classe a cui sono state regolarmente assegnate tutte le lezioni.
Un altro approccio è quello di considerare una funzione penalità, che rappresenta quanto i cromosomi sono inadeguati e costruisce la funzione fitness come: '(costante - penalità)'.
Secondo alcuni è più utile considerare quanti vincoli sono violati piuttosto che quanti sono soddisfatti. Una buona funzione di penalità può essere costruita a partire dal costo di completamento stimato, cioè il costo necessario (stimato) per far diventare valido un cromosoma che non lo è.
La stima di funzioni approssimate è una tecnica che può qualche volta essere usata se la funzione fitness è troppo complessa o lenta da valutare. Se può essere creata una funzione più veloce che approssimativamente dà il valore della funzione fitness 'corretta', si può trovare, in un dato tempo, un cromosoma migliore di quello che si sarebbe trovato usando la vera funzione fitness. Se per esempio la funzione semplificata è dieci volte più veloce, possono essere stimati dieci punti approssimati invece di valutare un solo punto esattamente,

e questo è generalmente meglio: infatti un algoritmo genetico è abbastanza robusto da convergere anche in presenza del 'rumore' (noise) introdotto dall'approssimazione. Questa tecnica è stata usata per la registrazione di immagini mediche, dove, volendo allineare due immagini, i risultati migliori sono stati ottenuti testando solo un millesimo dei pixel. Questo approccio può essere usato proficuamente se la funzione fitness è stocastica.

3.2.2 Fitness e Selezione

Talvolta la funzione di fitness possiede delle discontinuità tali per cui alcuni individui migliori ottengono un valore di fitness molto elevato (anche se non ottimale), mentre tutti gli altri membri ottengono delle valutazioni molto basse. In questo caso la generazione successiva tenderà ad essere dominata da copie genetiche del miglior individuo, le quali non avranno la possibilità di combinarsi con stringhe diverse: il processo evolutivo finirà presto in uno stato di convergenza prematura in un minimo locale.
Una volta che la popolazione converge, l'abilità dell'algoritmo di continuare la ricerca per trovare una soluzione migliore è praticamente eliminata: il crossover di individui quasi identici può portare ben pochi miglioramenti. Solo la mutazione rimane per poter esplorare nuove zone nello spazio dei parametri, e questo semplicemente porta ad una ricerca lenta e casuale.
Il teorema dello schema dice che dovremmo allocare prove riproduttive (o opportunità) in proporzione al loro fitness relativo, ma si può verificare in alcuni casi una convergenza prematura a causa del fatto che la popolazione non è infinita. Per far lavorare bene gli algoritmi genetici su di una popolazione finita, dobbiamo modificare la maniera con cui vengono scelti gli individui per la riproduzione.
L'idea base è che dobbiamo controllare il numero di opportunità che ogni individuo riceve in modo che non sia né troppo basso né troppo alto. L'effetto voluto è quello di restringere il range del fitness in modo da impedire che un qualsiasi individuo 'superfit' possa improvvisamente prevalere.

In altri casi potrebbe accadere la situazione opposta: tutte le stringhe riportano valori di fitness molto simili, per cui quasi tutti gli individui ottengono un

numero simile di copie. In questo caso il processo evolutivo tenderà a stagnare diventando simile ad un processo di ricerca casuale.
Nei casi limite l'individuo migliore potrebbe addirittura fallire nel riprodursi e qualsiasi soluzione, anche raggiunta in modo più o meno casuale, sarà persa nel ricambio generazionale. Esistono alcuni metodi (dei quali parleremo più avanti) per far fronte a questi problemi[1].

3.2.3 Tecniche di selezione dei genitori

La selezione dei genitori è il compito di allocare opportunità riproduttive a ciascun individuo. In principio, gli individui sono copiati dalla popolazione in una vasca di accoppiamento (*mating pool*) seguendo uno schema probabilistico. Sotto un severo schema di riproduzione, la dimensione della vasca è uguale a quella della popolazione (ma questa non è una regola obbligatoria). Terminata questa prima fase, tutti gli individui presenti nella vasca vengono estratti a coppie e fatti 'riprodurre'.
Il comportamento di un algoritmo genetico dipende in larga misura da come gli individui vengono scelti per andare nella *mating pool*. Il metodo più utilizzato (come già descritto nel primo capitolo) è quello della 'ruota della fortuna truccata'. Esistono numerosi altri sistemi per selezionare in maniera probabilistica i genitori: i più rappresentativi sono i metodi ottenuti tramite una rimappatura del fitness.

Si parla di **fitness rimappato esplicitamente** se si effettua una qualche operazione di normalizzazione sui valori di fitness grezzo al fine di migliorare l'evoluzione dell'algoritmo. I principali metodi di rimappamento esplicito, sono:

- il metodo di *Fitness Scaling*: vengono normalizzati i valori di fitness decimali a valori interi rapportando il fitness massimo con quello medio e

[1] J.E. Baker, 'Reducing bias and inefficiency in the selection algorithm', in 'Proceedings of the second international conference on genetic algorithms', (1987) a cura di J.J. Grefenstette, Hillsdale, NJ, Lawrence-Erlbaum Associates, pp. 14-21;
D.E. Goldberg e K. Deb, 'A comparative analysis of selection schemes used in genetic algorithms', in 'Foundations of genetic algorithms', (1991) a cura di G.J.E. Rawlins, San Mateo, CA, Morgan Kaufmann.

stabilendo a priori un massimo numero di prove riproduttive per ogni individuo (tipicamente due);

- il metodo di *Fitness Windowing* (o *metodo della finestra*[2]): simile al *Fitness Scaling*, ma viene considerato il fitness minimo nella normalizzazioni dei valori. In pratica questa tecnica consiste nel trovare il valore di fitness minimo della popolazione e assegnare a ciascuna stringa un nuovo valore di fitness uguale alla differenza tra la fitness della stringa e il valore minimo;

- il metodo di *Fitness Ranking*:è un altro metodo comune che evita di appoggiarsi su un individuo estremo. Gli individui sono ordinati a seconda del fitness grezzo, cioè il peggiore individuo avrà fitness 1, il secondo peggiore 2 e il migliore N (numero di cromosomi nella popolazione); solo allora i valori di fitness riproduttivo saranno assegnati in accordo al rango. Questo può essere fatto linearmente o esponenzialmente e dà un risultato simile al *fitness scaling*: in questo il rapporto massimo/media del fitness è normalizzato a un particolare valore. Inoltre ci assicura che il fitness rimappato di un individuo intermedio sarà regolarmente propagato. A causa di questo l'effetto di uno o due individui estremi sarà trascurabile, indipendentemente da quanto piccolo o grande sia il valore del loro fitness in rapporto al resto della popolazione, in questo modo i migliori cromosomi non differiscono molto dagli altri e questo può rallentare molto la convergenza. Il numero di prove riproduttive allocate, per esempio, al quinto miglior individuo sarà sempre lo stesso, qualunque sia il valore di fitness.
  Alcuni esperimenti dimostrano che il ranking è superiore al fitness scaling.

Si parla di **rimappamento implicito del fitness** se si selezionano i cromosomi da riprodurre (i genitori) senza passare attraverso livelli intermedi di rimappamento del fitness. La tecnica più rappresentativa di rimappamento implicito è la 'Tournament Selection': nella sua variante più semplice (la

[2]J.J.Grefenstette, 'Optimization of control parameters for genetic algorithms', in 'IEEE Transactions on systems, man and cybernetics', vol.16, pp.122-128 (1986).

cosiddetta 'selezione binaria di torneo'), coppie di individui sono prese a caso tra la popolazione e confrontate. Il vincitore del torneo viene copiato nella mating pool; questo è ripetuto finché la piscina non è piena. Tornei più grandi possono essere usati, dove il migliore di n individui scelti a caso è copiato nella mating pool, e questo ha l'effetto di aumentare la pressione di selezione, perché gli individui sotto la media difficilmente vinceranno i tornei, mentre i migliori avranno ottime probabilità.
Un ulteriore generalizzazione è la 'selezione con torneo binario statistico', dove i migliori individui vincono i tornei con probabilità p dove $0.5 < p < 1$. Usando valori più bassi di p si diminuisce la pressione di selezione, perché gli individui sotto la media sono in proporzione più avvantaggiati nel vincere un torneo, mentre quelli sopra la media perdono probabilità.
Aggiustando la probabilità di vincere o la dimensione del torneo, la pressione di selezione può essere resa grande o piccola a piacere.

Goldberg e Deb hanno confrontato quattro differenti schemi: selezione proporzionale, fitness ranking, tournment selection e steady-state selection (che vedremo nel prossimo paragrafo) ed hanno concluso che, tramite opportuni aggiustamenti dei parametri, tutti gli schemi (tranne la selezione proporzionale) hanno performance simili, quindi non esiste una tecnica migliore in assoluto.

3.2.4 Elitismo e Steady-State Reproduction

Le tecniche di riproduzione viste fin'ora prevedevano che la popolazione fosse sostituita completamente ad ogni generazione. Questa impostazione porta però qualche problema: se per esempio il migliore individuo non riesce a riprodursi, avremo nel ricambio generazionale una perdita più o meno importante di informazione.
Sono dunque state studiate tecniche che permettono di ridurre perdite consistenti di informazione. Una di esse prevede che la migliore stringa di ciascuna generazione venga ricopiata immutata nella generazione successiva. Questa tecnica, definita 'strategia elitista', svolge due importanti funzioni: assicura che il cromosoma migliore sia presente nella generazione successiva anche se il processo di riproduzione probabilistica fallisce nel riprodurlo e preser-

va la struttura genetica nel caso in cui l'operatore di crossover distrugga lo schema ottimale sottostante. Siccome solitamente la strategia elitista produce un miglioramento del processo evolutivo, essa viene impiegata nella maggior parte degli algoritmi genetici assieme ad un processo di normalizzazione della fitness per evitare che l'individuo migliore domini la popolazione.
Una seconda tecnica consiste nel modificare l'operatore di riproduzione in modo da rimpiazzare solamente pochi individui alla volta: questa tecnica viene definita con il nome di 'riproduzione a regime stazionario' (*steady-state reproduction*[3]). Il processo di creazione dei nuovi individui è identico a quello della ruota della fortuna, ma in questo caso vengono create solo le prime k copie (con k minore della grandezza della popolazione) che vengono incrociate, mutate e sostituite ai k individui peggiori della popolazione. Una versione più efficiente della riproduzione a regime costante consiste nel gettar via le stringhe che sono uguali a quelle già presenti nella popolazione ('riproduzione a regime stazionario senza duplicazione'): questa soluzione assicura costantemente la diversità della popolazione allontanando così il pericolo di una convergenza prematura.

3.3 Crossover

Sono stati proposti diversi tipi di *crossover*: quello descritto nel primo capitolo viene anche definito 'crossover a taglio singolo' (*single-point crossover*) perché viene selezionato un unico punto casuale attorno al quale viene scambiato parte del materiale genetico.
Esistono però anche diverse versioni di 'crossover a taglio multiplo' (*multi-point crossover*) dove vengono selezionati diversi punti casuali di scambio genetico. Questa soluzione permette un maggior numero di gradi di libertà nella ricombinazione di soluzioni parziali in strutture di complessità maggiore e potrebbe accelerare il processo evolutivo. Questo operatore diventa interessante anche

[3]G. Syswerda, 'Uniform Crossover in Genetic Algorithms', in 'Proceedings of the Third International Conference on Genetic Algorithms' (1989), a cura di J.D. Schaffer, San Mateo, CA, Morgan Kaufmann;
D. Whitley, 'GENITOR: A Different Genetic Algorithm' in 'Proceedings of the Rocky Mountain Conference on Artificial Intelligence' (1988), Denver, CO.

quando la stringa genetica di un individuo codifica diversi parametri non strettamente collegati fra di loro: in questo caso la mescolanza dei geni potrebbe risultare in un'evoluzione meno efficiente o più lunga per cui conviene piuttosto effettuare tagli ed incroci separati all'interno delle rispettive aree genetiche.
Il crossover svolge un ruolo molto importante nella teoria degli schemi ed ha tradizionalmente occupato una posizione centrale negli algoritmi genetici; recentemente però questa tradizione è stata messa in discussione. Innanzi tutto la teoria degli schemi è valida solamente in un contesto limitato, in quanto assume un alfabeto genetico discreto e finito: questo non significa che gli algoritmi genetici falliscano in tutti gli altri contesti (al contrario !), ma in questi casi il processo di elaborazione degli schemi non è stato dimostrato. In secondo luogo, esiste un vasto corpus di esperimenti recenti su codici genetici composti da numeri reali in cui non viene utilizzato il crossover, ma solamente la riproduzione selettiva e una probabilità di mutazione maggiore. Infine, come vedremo più avanti, esiste una versione 'europea' di evoluzione artificiale che si rivela altrettanto efficace, ma che non incorpora l'operatore di crossover.

3.3.1 2 - point Crossover

In questa tecnica (e in generale nel multi-point) piuttosto che stringhe lineari i cromosomi possono essere considerati come circoli formati unendo gli estremi del cromosoma (vedi Figura 3.1). Per cambiare un segmento da un circolo con un altro proveniente da un altro ciclo, si richiede la selezione di due punti di taglio, come mostrato in figura. In questo modo il one point crossover può essere visto come un 2-point crossover, con uno dei punti di taglio fissato all'inizio della stringa. Quindi il 2-point opera come il one-point (cioè cambiando un solo segmento), ma è più generale. Un cromosoma, considerato come un circolo, può contenere più building blocks, poiché sono in grado di avvolgersi alla fine della stringa. I ricercatori sono d'accordo nell'affermare che il 2-point crossover è generalmente migliore del one-point.

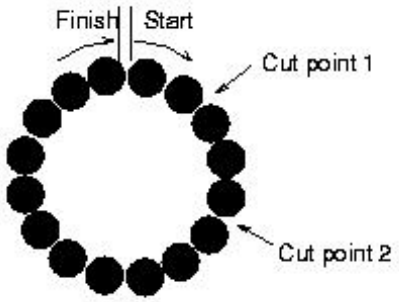

Figura 3.1: *2 - point Crossover.*

3.3.2 Crossover Uniforme

Questa tecnica è completamente differente dal one-point crossover: ciascun gene nei figli è creato tramite una copia del corrispondente gene da uno dei due genitori, scelto in accordo a una 'maschera di crossover' creata in maniera casuale (vedi Figura 3.2).

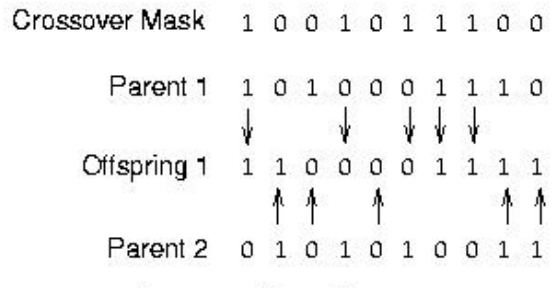

Figura 3.2: *Crossover Uniforme.*

Dove c'è un '1' nella maschera, il gene è copiato dal primo genitore; dove c'è uno '0', il gene è copiato dal secondo genitore. Il processo è ripetuto con i genitori scambiati per produrre un secondo figlio. Una nuova maschera di crossover è generata casualmente per ciascuna coppia di genitori. Il figlio quindi contiene una mistura di geni provenienti da ciascun genitore. La tecnica del

crossover uniforme potrebbe essere paragonata ad un multi-point crossover, in cui il numero degli effettivi punti di taglio non è fissato a priori.

3.3.3 Quale tecnica è la migliore?

Il dibattito su quale sia la migliore tecnica di crossover è tutt'altro che risolto. Riguardo il crossover uniforme gli schemi che hanno un particolare ordine (ricordiamo che l'ordine di uno schema è il numero di posizioni 'fisse') hanno la stessa probabilità di essere distrutti, a prescindere dalla lunghezza di definizione (la distanza tra la prima e l'ultima posizione fissata nella stringa). Con il 2-point crossover è la lunghezza di definizione dello schema che determina la sua predisposizione alla distruzione, non il suo ordine. Questo significa che rispetto al 2-point, nel crossover uniforme gli schemi con lunghezze di definizione corte hanno maggiori probabilità di essere distrutti, mentre, per quel che riguarda schemi 'più lunghi', vale il contrario.
Il crossover uniforme ha il vantaggio che l'ordinamento dei geni è del tutto irrilevante e questo significa che gli operatori di riordinamento come l'inversione (che sarà spiegato più avanti) non sono necessari, e noi non dobbiamo preoccuparci di posizionare i geni per migliorare i buiding blocks. L'efficienza dell'algoritmo genetico che usa il 2-point cade drammaticamente se non sono rispettate le raccomandazioni della 'building block hypothesis'. Il crossover uniforme, inoltre, continua a lavorare bene almeno tanto quanto un 2-point usato con un cromosoma ordinato correttamente.
DeJong ha studiato l'efficienza del crossover multipoint ed è arrivato alla conclusione che il 2-point crossover dà un miglioramento, ma che aggiungere più punti crossover riduce le prestazioni dell'algoritmo: in quest'ultimo caso, inoltre, i building blocks sono più facili da spezzare. Al contrario, un vantaggio di avere molti punti crossover è che nello spazio del problema si può fare una ricerca più accurata.

Non è chiaro, quindi, se una tecnica sia migliore dell'altra: molti sono comunque gli studi che analizzano le differenze e le presunte superiorità di uno o dell'altro metodo, senza però raggiungere un consenso unanime.

3.3.4 Altre tecniche di Crossover

Nel corso degli anni sono state ideate molte altre tecniche.
Un esempio è l'idea che il crossover debba essere più forte in certe posizioni sulla stringa piuttosto che in altre (modello che ha qualche fondamento in natura). In questo caso, il principio generale è che l'algoritmo genetico impara adattativamente quali siti dovrebbero essere favoriti per il crossover. Questa informazione è registrata in una 'stringa punteggiatura', che è essa stessa parte del cromosoma, e quindi viene incrociata e passata ai discendenti. In questo modo le stringhe punteggiatura che vanno in direzione della migliore discendenza saranno esse stesse propagate attraverso la popolazione.
Goldberg, invece, descrive un operatore crossover abbastanza diverso che si chiama 'Partially Matched Crossover' (PMX), per l'uso in problemi basati sull'ordine (in un problema basato sull'ordine, come il problema del commesso viaggiatore, i valori dei geni sono discreti e limitati e il fitness dipende dall'ordine col quale loro appaiono). Nel *Partially Matched Crossover* non sono incrociati i valori dei geni, ma l'ordine con cui appaiono. I figli hanno geni che ereditano ordinando informazioni da ciascun genitore. Questo elimina la generazione di figli che violano i vincoli del problema.
Anche Syswerda e Davis descrivono altri operatori meno importanti basati sull'ordine.

3.4 Mutazione ed Evoluzione Naive

La mutazione è tradizionalmente vista come un operatore secondario, responsabile di una reintroduzione inaspettata di valori di geni perduti (ad esempio alleli recessivi) che prevengono la deriva genetica, fornendo un piccolo elemento di ricerca casuale nella vicinanza della popolazione dove esso è largamente convergente. Generalmente, infatti, si ritiene che il crossover sia la principale forza che guida la ricerca dello spazio del problema.
Comunque, gli esempi in natura, mostrano che la riproduzione asessuata può produrre creature sofisticate senza il crossover. I biologi considerano infatti la mutazione come la principale fonte di materiale per cambiamenti evolutivi.
Gli esperimenti che sono stati fatti sull'ottimizzazione dei parametri per gli

algoritmi genetici hanno mostrato che il crossover produce molti meno effetti di quanto si pensava mentre l'evoluzione *naive* (solo selezione e mutazione) agisce in modo simile all'*hillclimb* e può essere potente anche senza il crossover. Il crossover infatti produce evoluzioni più veloci rispetto a una popolazione che subisce solo mutazione, ma la mutazione, alla fine del processo evolutivo, fornisce soluzioni migliori. Davis puntualizza che mentre la popolazione si avvicina alla convergenza la mutazione diventa più produttiva del crossover.
Nonostante la sua bassa probabilità di uso, la mutazione è dunque un operatore molto importante. Spears ritiene che l'operatore di mutazione opportunamente modificato può fare tutto quello che fa il crossover ed afferma anche che questi due operatori sono in realtà forme di un più generale operatore di esplorazione.

Capitolo 4

Aspetti avanzati

Numerosi sono gli aspetti ed i problemi che sono stati affrontati nei capitoli precedenti,ma anni di teoria e utilizzo pratico degli algoritmi genetici hanno portato molti studiosi a confrontarsi con aspetti caratteristici di problemi specifici. Nelle prossime sezioni ne vedremo alcuni tra i più frequenti e interessanti.

4.1 Inversione e Ordinamento

L'ordine dei geni in un cromosoma è critico perché la building block hypothesis funzioni efficientemente; per questo sono state suggerite tecniche per riordinare le posizioni dei geni durante l'esecuzione.
Una di queste tecniche è l'**inversione**: essa lavora invertendo l'ordine dei geni tra due posizioni scelte casualmente all'interno del cromosoma (quando sono usate queste tecniche, i geni devono essere associati ad etichette per poterne identificare la posizione oltre al valore numerico). Il fine di questo riordinamento è di cercare di trovare sequenze ordinate di geni che anno potenziale evolutivo migliore.
Il riordinamento non fa niente per una bassa *epistasis* (si veda la prossima sezione), quindi non può aiutare per quanto riguarda le altre richieste della 'building block hypothesis' e chiaramente non aiuta se la relazione tra i geni non consente un semplice ordinamento lineare. Inoltre, se si usa il crossover uniforme, l'ordine del gene è irrilevante e quindi non è necessario il riordinamento.

L'algoritmo che utilizza l'inversione non solo cerca delle buone sequenze di geni, ma simultaneamente cerca di ordinarli in maniera opportuna. Il riordinamento espande lo spazio di ricerca in modo molto grande e rende il problema molto più difficile da risolvere; inoltre il tempo speso cercando ordinamenti di geni migliori è tempo portato via alla ricerca di migliori valori di geni.

In natura ci sono molti meccanismi tramite i quali la disposizione dei cromosomi si può evolvere ('evoluzione karyotypic'), l'inversione è solo una di questi. Le specie hanno maggiore probabilità di sopravvivere se la loro *evoluzione cariotipica* li porterà a essere più facilmente adattati a nuove condizioni, come i cambiamenti ambientali. La valutazione del genotipo prende posto velocemente in ciascuna generazione, ma quella del cariotipo prende spazio molto lentamente, forse dopo migliaia di generazioni.
Per la grande maggioranza delle applicazioni degli algoritmi genetici, l'ambiente espresso tramite la funzione fitness è statico; prendendo esempio dalla natura sembrerebbe che l'*evoluzione karyotype* sia di poca importanza in questi casi. Comunque in applicazioni dove la funzione fitness varia dopo tempo, e l'algoritmo deve fornire una soluzione che si adatti all'ambiente che cambia, l'evoluzione del cariotipo può valere la pena di essere rappresentata.

4.2 Epistasis

Il termine 'epistasi' è stato definito dai genetisti per indicare il fatto che l'influenza di un gene nel fitness di un individuo dipende da valori di geni presenti in altri punti della sequenza cromosomica. Più specificatamente i genetisti usano questo termine nel senso di un effetto di mascheramento o di cambiamento.
Un gene è detto **epistatico** quando la sua presenza inibisce l'effetto di un altro gene in un altro locus (per locus si intende la posizione all'interno del cromosoma). I geni epistatici sono alcune volte chiamati geni inibitori a causa dei loro effetti sugli altri geni.
Generalmente ci sono interazioni molto intricate e complesse tra un gran numero di geni che si sovrappongono; in particolare ci sono 'catene di influenza':

in natura, per esempio, la produzione di una proteina è codificata da un gene che è coinvolto con una proteina prodotta da un altro gene per produrre un terzo prodotto, che a sua volta reagisce con altri enzimi prodotti da qualche altra parte. Quindi c'è un considerevole numero di interazioni tra geni nel corso della produzione del fenotipo, benché i genetisti non possano riferirsi a questo come epistasi.
Quando i ricercatori degli algoritmi genetici usano il termine epistasi, si stanno riferendo generalmente a una forte interazione tra geni, ma solitamente evitano di dare una precisa definizione. Comunque possiamo dire che l'epistasi è l'interazione tra due differenti geni in un cromosoma. Questo implica che il valore di un gene (in termini di fitness) dipende dai valori di altri geni.
Il grado di interazione sarà, in generale, differente per ciascun gene in un cromosoma: se effettuiamo una modifica di un gene, ci aspettiamo un cambiamento risultante nel fitness nel cromosoma. Questo cambiamento può variare in accordo con i valori degli altri geni.
Per fare una breve classificazione distinguiamo tre livelli di interazione tra i geni a seconda dell'effetto che hanno nel fitness dei cromosomi:

• Livello 0: nessuna interazione. Un particolare cambiamento in un gene produce lo stesso cambiamento nel fitness.
• Livello 1: leggera interazione. Un particolare cambiamento in un gene spesso produce un cambiamento nel fitness dello stesso segno o nullo.
• Livello 2: epistasi. Un particolare cambiamento in un gene produce un cambiamento nel fitness che varia nel segno e magnitudine(intensità), in base al valore di altri geni.

Un esempio di problema a 'livello 0' è il semplice problema del 'contare i bit 1', dove il fitness è proporzionale al numero di '1' in una stringa binaria. Un esempio del 'livello 1' è la funzione 'plateau', dove il fitness vale 1 se tutti i bit sono 1 e 0 se sono tutti 0.
Quando i ricercatori di algoritmi genetici usano il termine epistasi si vogliono generalmente riferire al livello 2.
Gli algoritmi genetici possono migliorare tecniche semplici in problemi complessi di livello 2 esibendo molte interazioni tra i parametri, cioè con epistasi significante. Sfortunatamente, in accordo con la building block hypothesis,

una delle richieste di base per il successo di un algoritmo genetico è che ci deve essere una bassa epistasi; questo suggerisce che essi non saranno efficienti nei problemi dove sono necessari, quindi ci serve sapere come evitare l'epistasi oppure come costruire un algoritmo che lavorerà bene con alta epistasi.

4.2.1 Deception (inganno)

Uno dei fondamentali principi degli algoritmi genetici è che i cromosomi in cui sono presenti schemi che sono contenuti nell'ottimo globale cresceranno in frequenza (dalla teoria degli schemi sappiamo che questo è vero soprattutto per schemi piccoli e di ordine basso, noti come building block). Nel corso delle generazioni, per via del processo del crossover, questi schemi si consolideranno verso l'ottimo globale; ma se gli schemi che non sono contenuti nell'ottimo globale crescono in frequenza più rapidamente di quelli che vi sono contenuti, l'algoritmo si allontanerà dall'ottimo invece di avvicinarsi. Questo effetto è noto come **deception** ed è uno speciale caso di epistasi (più precisamente, l'epistasi è una condizione necessaria ma non sufficiente per la deception).
Statisticamente uno schema crescerà in frequenza su una popolazione se il suo fitness è più alto della media di tutti gli altri (per fitness di uno schema, come già precedentemente definito, si intende la media dei fitness dei cromosomi che contengono quello schema). Un problema è detto 'deceptive' (ingannevole) se il fitness medio degli schemi che non sono contenuti nell'ottimo globale è più alto della media di quelli che vi sono contenuti. Inoltre un problema è definito 'completamente ingannevole' se tutti gli schemi di basso ordine contenuti in una soluzione ottima hanno un valore di fitness inferiore agli altri schemi presenti nella popolazione.

4.2.2 Affrontare l'epistasi

Il problema dell'epistasi può essere affrontato in due maniere: come un problema di codifica o come un problema teorico. Se trattato come problema di codifica la soluzione è trovare una differente codifica (rappresentazione) e un metodo di decodifica che non esibisce epistasi: questo permetterà di usare

un algoritmo genetico convenzionale. Alcune codifiche possono essere fatte semplicemente usando crossover e mutazione appropriatamente progettati, in modo che sia possibile rappresentare il problemi con epistasi assente o piccola. Comunque, per problemi difficili, lo sforzo coinvolto nell'inventare questa codifica sarà considerevole e costituirà effettivamente la soluzione del problema iniziale. Se questo non può essere fatto si usa il secondo approccio.
La teoria tradizionale derivata dal teorema degli schemi si basa sulla bassa epistasi. Se i geni in un cromosoma hanno alta epistasi, deve essere sviluppata una nuova teoria e inventati nuovi algoritmi per agire in queste condizioni. L'ispirazione deve venire ancora una volta dalla genetica della natura dove l'epistasi è molto comune. Per esempio, per ridurre l'epistasi nel cromosoma si è trovato che è utile convertire il problema in un altro basato sull'ordine. Per operare conversioni di questo tipo bisogna modificare la teoria degli algoritmi genetici nel suo insieme, ad esempio utilizzando il PMX crossover anzichè il crossover tradizionale.

4.3 Operatori Dinamici

Durante l'esecuzione dell'algoritmo le impostazioni ottime dei valori della probabilità degli operatori (mutazione e crossover) possono variare. Davis ha provato una variazione lineare della probabilità della mutazione e del crossover: mentre il secondo decresce durante l'esecuzione, il primo cresce. Booker invece utilizza un valore di crossover variabile dinamicamente, che dipende dalla propagazione del fitness: quando la popolazione converge, il valore del crossover si riduce per dare più opportunità alla mutazione di trovare nuove variazioni, quindi questo ha un effetto simile alla tecnica lineare di Davis, ma con il vantaggio di essere adattativo. Davis descrive una nuova tecnica adattativa che si basa sul successo degli operatori nel trovare buoni figli. Si dà un credito a ciascun operatore quando produce un buon individuo nella popolazione in base agli ultimi cinquanta accoppiamenti e in seguito, per ogni evento riproduttivo, un operatore è selezionato a caso secondo il suo credito. Il valore del credito varia in maniera adattativa dipendente dal problema; se durante il corso dell'esecuzione un operatore perde improvvisamente molto credito,

probabilmente è meno efficace degli altri. In questo modo si può costruire un algoritmo efficiente senza preoccuparci troppo di trovare parametri ottimi degli operatori, dato che questi si adattano automaticamente durante l'esecuzione.

4.4 Nicchia e Specificazione

Negli ecosistemi naturali, ci sono molti modi differenti con i quali gli animali possono sopravvivere: cacciando, pascolando, sottoterra, sugli alberi o in innumerevoli altri modi, e le differenti specie evolvono per riempire ciascuno la sua 'nicchia ecologica'. La specificazione è il processo nel quale una singola specie si differenzia in due o più specie che occupano differenti nicchie.
Negli algoritmi genetici le nicchie sono il massimo della funzione fitness. Alcune volte abbiamo funzioni fitness multimodali di cui vogliamo localizzare tutti i picchi; sfortunatamente gli algoritmi genetici tradizionali non riescono a fare questo: l'intera popolazione converge in un singolo picco.
Alcune modifiche agli algoritmi genetici tradizionali, basate sugli ecosistemi naturali, sono state proposte per risolvere questo problema. Le due tecniche base sono state pensate per mantenere la diversità e per raggiungere il 'costo' associato ad una nicchia.
Cavicchio ha ideato un meccanismo che introduce 'preselezione': ovvero un figlio viene confrontato con il genitore con fitness più basso e dei due sopravvive quello con fitness maggiore. Innanzitutto notiamo una dura competizione tra genitori e figli; inoltre il guadagno non viene condiviso, ma il vincitore prende tutto. Questo metodo aiuta a mantenere la diversità in quanto le stringhe tendono a rimpiazzarne altre che sono simili a loro, cercando di prevenire la convergenza in un singolo massimo. Il problema spesso non viene superato se però ci sono molti massimi locali con fitness vicino al massimo globale. Inoltre una tecnica che distribuisce i membri della popolazione nei picchi in proporzione al fitness di questi ultimi, non riesce a trovare facilmente il massimo globale se ci sono più picchi di quanti sono i membri della popolazione.
Un differente approccio è quello delle 'nicchie sequenziali' che si basa su esecuzioni multiple dell'algoritmo genetico: ciascuna di esse localizza un picco e

la sua funzione fitness viene modificata in modo che il picco venga 'cancellato'. Questo ci assicura che in una esecuzione successiva non venga ritrovato. L'algoritmo quindi riparte con una nuova popolazione e in questo modo viene localizzato un nuovo picco per ogni esecuzione.

4.4.1 Restricted Mating

Uno schema di 'mating restriction' permette a un individuo di accoppiarsi con un altro solo se quest'ultimo appartiene alla stessa nicchia oppure, se non ci sono altri nella nicchia, con un individuo scelto a caso. Lo scopo è quello di incoraggiare la specificazione e ridurre la popolazione di 'letali', cioè di individui figli di genitori di nicchie differenti: in questo caso, infatti, sebbene ciascun genitore possa avere alto fitness, la combinazione dei loro cromosomi può essere molto scadente se cade in una valle tra due massimi. La natura evita la formazione di letali evitando l'accoppiamento tra specie differenti.
La filosofia del 'restricted mating' assume come ipotesi che se due genitori simili (cioè della stessa nicchia) sono accoppiati, allora i figli saranno simili ad essi. Molto dipende anche dallo schema di codifica, in particolare dall'esistenza di building block e bassa epistasi: con queste ipotesi, usando gli operatori convenzionali di mutazione e crossover, due genitori con genotipo simile produrranno sempre figli con genotipo simile. Se, al contrario, il cromosoma è altamente epistatico, non c'è garanzia che questi figli non abbiamo basso fitness (cioè siano letali): somiglianze nel genotipo non implicano infatti somiglianze nel fenotipo. Questi effetti limitano quindi l'uso del restricted mating.

4.5 Duplicità e Dominio

Nelle forme viventi superiori i cromosomi contengono due insiemi di geni piuttosto che uno solo; questo fenomeno è conosciuto come diploidia (mentre un cromosoma con un solo insieme è detto aploide). Molti libri di genetica tendono a concentrarsi sui cromosomi diploidi, mentre virtualmente tutto il lavoro degli algoritmi genetici si concentra sugli aploidi (per semplicità). I cromosomi diploidi danno benefici agli individui quando l'ambiente può cambiare dopo

un certo tempo: avere due geni significa essere in possesso di due differenti soluzioni da utilizzare e passare ai figli; una di queste sarà dominante (e sarà espressa nel fenotipo), mentre l'altra sarà recessiva. Se le condizioni esterne cambiano la dominante può dar spazio al gene recessivo: questo passaggio può avvenire molto più rapidamente di quanto sia possibile fare mutando il gene con meccanismi evolutivi. Il meccanismo evidenziato è particolarmente adatto se l'ambiente può assumere due stati (ad esempio era glaciale - non glaciale).
Il principale vantaggio della diploidia è che mostra una diversità maggiore degli alleli comparato con l'aploidia. Un allele correntemente dannoso, ma potenzialmente utile, può essere mantenuto in una posizione recessiva. Altri meccanismi genetici possono realizzare lo stesso effetto: per esempio un cromosoma può contenere alcune varianti di un gene, assicurandosi che soltanto una delle varianti venga espressa in un particolare individuo. Una situazione come questa accade in natura con la produzione dell'emoglobina: differenti geni codificano la sua produzione durante differenti fasi dello sviluppo, uno nella fase fetale, l'altro nella fase adulta.

4.6 Tecniche basate sulla conoscenza

Molti ricercatori al posto dei tradizionali operatori di crossover e mutazione hanno ideato per particolari problemi dei nuovi operatori che usano la conoscenza del dominio. Questo rende ciascun operatore più adatto al compito specifico: esso risulta quindi meno robusto, ma può migliorare significativamente la performance.
La conoscenza del dominio può essere usata per scartare cromosomi poco adatti o quelli che possono violare i vincoli del problema: questo evita di perdere tempo a valutare individui non significativi e di introdurre individui scadenti nella popolazione.
La conoscenza può essere utilizzata inoltre per introdurre operatori di miglioramento locale che compino esplorazioni più efficienti nello spazio della ricerca oppure per compiere una 'inizializzazione euristica' della popolazione, cosicché la ricerca inizi da punti ragionevolmente buoni rispetto a un insieme scelto casualmente.

4.7 Diversi algoritmi genetici

Come è stato accennato all'inizio del secondo capitolo, si è soliti attribuire l'invenzione degli algoritmi genetici all'americano John Holland; in realtà essi erano già stati inventati due anni prima dal tedesco Rechenberg[1]. Gli algoritmi evolutivi tedeschi, chiamati con il nome di 'strategie evoutive' (*evolutionstrategie*), sono leggermente diversi dagli algoritmi genetici in quanto sono basati soprattutto sull'operatore di mutazione ed operano su stringhe di numeri reali. Successivamente Schwefel[2], apparentemente all'oscuro di quanto stava avvenendo negli Stati Uniti, sviluppò ulteriormente le strategie evolutive aggiungendovi un operatore di crossover (che comunque continuava a svolgere un ruolo secondario) e lasciando evolvere alcuni parametri di mutazione assieme agli individui della popolazione. In un confronto fra gli algoritmi genetici e le strategie evolutive su una serie di problemi standard, queste ultime sono state in grado di convergere verso una soluzione ottimale in un numero più grande di casi, probabilmente grazie all'adattamento autonomo dei parametri di mutazione durante il processo evolutivo[3].

4.8 Confronto con altre tecniche

Nell'ultimo decennio il ruolo dell'ottimizzazione è diventato sempre più grande, in quanto molti problemi possono essere risolti con i computer moderni, ma solo attraverso processi di ottimizzazione. Gli algoritmi genetici sono rivolti a tali problemi complessi. Essi appartengono alla classe degli algoritmi probabilistici, ma sono differenti dagli algoritmi casuali poiché combinano elementi di ricerca diretta e stocastica; inoltre, mentre gli altri metodi processano un singolo punto dello spazio di ricerca per volta, gli algoritmi genetici lavorano

[1] I. Rechenberg, 'Evolutionstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution', (1973), Stuttgart, Friedrich Fromann Verlag.

[2] H.P. Schwefel, 'Numerical optimization of computer models', (1981), Chichester, Wiley.

[3] F. Hoffmeister e T. Bäck, 'Genetic Algorithms and Evolution Strategies: Similarities and Differences' in 'Parallel Problem Solving from Nature', (1991), a cura di H.P. Schwefel e R. Maenner, Berlin, Springer-Verlag, pp. 455-469.

su di una popolazione di soluzioni.

4.8.1 Esplorazione e sfruttamento

Un qualsiasi algoritmo di ottimizzazione efficiente, deve usare due tecniche per trovare il massimo globale: esplorazione, per esaminare nuove e sconosciute aree dello spazio di ricerca, e sfruttamento, ovvero utilizzare i punti precedentemente visitati per cercare punti migliori. Queste richieste sono contraddittorie, e un buon algoritmo di ricerca deve trovare un buon compromesso tra le due.
Una ricerca puramente casuale è buona per l'esplorazione, ma non fa nessuno 'sfruttamento', mentre un metodo puramente di scalata (*hillclimb*) è buono per lo sfruttamento, ma fa poca esplorazione. La combinazione di queste due tecniche può essere abbastanza efficace, ma è difficile sapere dove si trova l'equilibrio migliore (cioè quanto sfruttamento bisogna fare prima di arrendersi e esplorare di più).
Holland ha dimostrato che un algoritmo genetico combina insieme esplorazione e sfruttamento allo stesso tempo e in un modo ottimale.
Sebbene questo sia teoricamente vero, in pratica ci sono problemi inevitabili. Holland infatti è partito da alcune importanti semplificazioni:

- la popolazione è infinita;
- la funzione fitness riflette accuratamente l'utilità della soluzione;
- i geni in un cromosoma non interagiscono significativamente.

La prima condizione non può essere mai verificata in pratica e a causa di ciò il funzionamento dell'algoritmo sarà soggetto ad errori stocastici. Un problema del genere, che si trova anche in natura è quello dello deriva genetica (*genetic drift*): anche in assenza di qualsiasi pressione di selezione (cioè se la funzione fitness è costante), membri della popolazione continueranno a convergere verso qualche punto nello spazio di ricerca; questo succede semplicemente a causa dell'accumulazione di errori stocastici. Se un gene diventa predominante nella popolazione, allora ha la stessa probabilità sia di diventare più dominante nella generazione successiva, che meno dominante. Se avviene un aumento della predominanza in alcune generazioni successive, e la popolazione è finita,

allora un gene si può propagare a tutti i membri della popolazione. Una volta che un gene converge in questa maniera, il crossover non può introdurre nuovi valori di geni: ciò produce un effetto a catena, tale da rendere 'fissi' più geni nell'evolversi delle generazioni.
Il tasso di deriva genetica produce allora un limite inferiore alla velocità con la quale un algoritmo genetico può convergere verso la soluzione corretta: può essere ridotto introducendo la mutazione, ma, se la mutazione è troppo elevata, può portare ad una eccessiva casualità con risultati poco interessanti.
La seconda e la terza condizione possono essere verificate su funzioni test che si comportano bene in laboratorio, ma sono difficili da soddisfare nel mondo reale: abbiamo infatti già visto come siano spesso presenti problemi legati alla funzione fitness e all'epistasis.

4.8.2 Vantaggi offerti dagli algoritmi genetici

Come è stato evidenziato più volte, gli algoritmi genetici possono essere considerati una tecnica di ricerca dei massimi di funzioni. È quindi lecito chiedersi quali vantaggi essi possano offrire rispetto agli altri metodi di ricerca dell'ottimo di una funzione. Golberg[4] ha messo a confronto gli algoritmi genetici con i tre metodi principali: metodi di ottimizzazione basati sul calcolo infinitesimale, metodi enumerativi e metodi di ricerca casuale.
I metodi basati sul calcolo infinitesimale si dividono a loro volta in due classi: i metodi di ricerca diretta ed i metodi di ricerca indiretta. I primi si basano sulla risoluzione analitica di un sistema di equazioni (solitamente non lineari) ottenuto eguagliando il gradiente della funzione a zero. I metodi di ricerca indiretta consistono invece nel partire da un punto qualsiasi dello spazio dei parametri della funzione spostandosi nella direzione del gradiente della funzione. Entrambi i metodi spesso non sono applicabili nella soluzione di problemi tipici del mondo reale, i quali raramente rispettano l'assunzione matematica di continuità della funzione necessaria per il calcolo del gradiente. Inoltre essi operano partendo da un singolo punto nello spazio e finiscono coll'individuare il massimo più vicino (locale) della funzione. Il risultato della ricerca può

[4]D.E. Goldberg, 'Genetic Algorithms in Search, Optimization and Machine Learning', (1989), Reading, MA, Addison-Wesley.

essere scadente quando la funzione da ottimizzare possiede diversi massimi locali, come è il caso della maggior parte dei problemi del mondo reale: una volta che il metodo ha raggiunto un picco locale, è ben difficile scenderne per raggiungerne uno più alto.

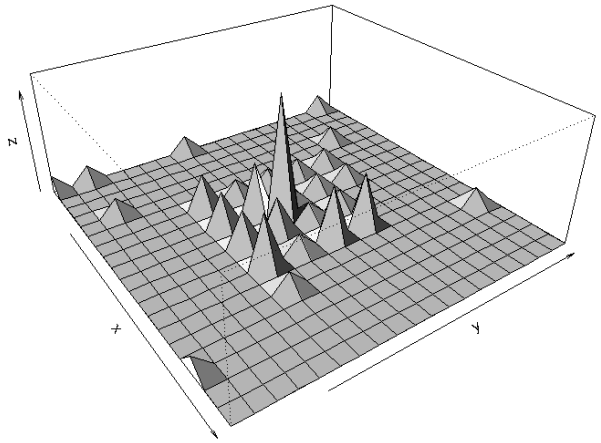

Figura 4.1: *Esempio di funzione con diversi massimi locali e discontinuità delle derivate.*

I metodi di ricerca enumerativi consistono nell'esame sistematico di uno spazio finito opportunamente discretizzato. Ciascun punto dello spazio della funzione viene valutato fino a quando viene trovato il punto di massimo. Ovviamente questi metodi tendono a comportare tempi di ricerca estremamente lunghi non appena il numero di punti dello spazio di ricerca comincia a crescere e possono quindi operare solamente su spazi estremamente ridotti rispetto a gran parte dei problemi di reale interesse.
I metodi di ricerca casuale consistono nel valutare in successione diversi punti dello spazio scelti in modo aleatorio, registrando di volta in volta i punti migliori. Per certi versi essi sono simili ai metodi di ricerca enumerativa, in particolare per quanto riguarda i problemi di efficienza descritti sopra. Tuttavia, i metodi di ricerca casuale sono stati resi molto più efficienti con l'impiego delle tecniche di ricottura simulata[5].
Questa tecnica è stata inventata da Kirkpatrick nel 1982 ed è sostanzialmente

[5]Si legga di L. Davis, 'Genetic Algorithms and Simulated Annealing', (1987), London, Pitman, per un confronto tra ricottura simulata ed algoritmi genetici.

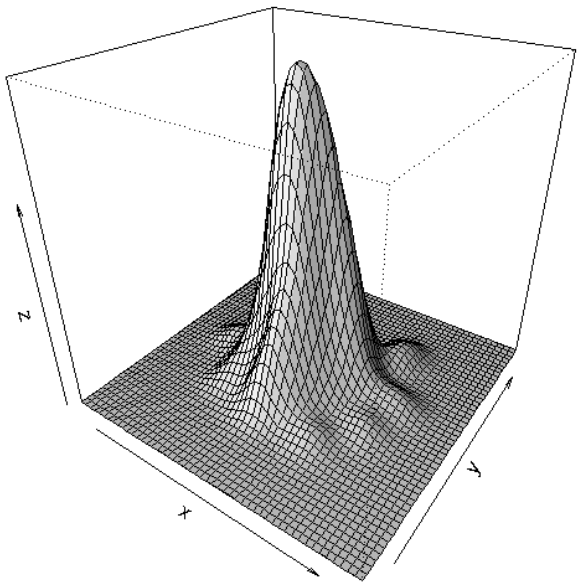

Figura 4.2: *Esempio di funzione 'liscia' e con pochi massimi locali.*

una versione modificata dell'hillclimbing. Iniziando da un punto scelto a caso nel dominio, viene fatto un movimento casuale: se il movimento porta a un punto con valore di fitness maggiore allora è accettato, se ci porta ad un valore con una valutazione inferiore è accettato con probabilità $p(t)$, dove t rappresenta il tempo.
All'inizio $p(t)$ vale quasi uno, ma gradualmente tende a zero: quindi, durante le prime iterazioni, ogni movimento viene accettato con maggiore probabilità, ma, proseguendo con la ricerca, la 'temperatura' si riduce e la probabilità di accettare un movimento negativo diminuisce. È importante notare come i movimenti negativi siano necessari per tentare di evitare massimi locali, ma se sono troppi possono allontanarci dal massimo. Come la tecnica della ricerca casuale, anche la ricottura simulata lavora con solo una soluzione candidata per volta e perciò non costruisce una figura complessiva dello spazio di ricerca e non vengono salvate informazioni dai precedenti movimenti per guidarci verso la soluzione.
Anche gli algoritmi genetici fanno uso di strumenti di ricerca casuale, ma l'intero processo è guidato dalla riproduzione selettiva e si basa su una codifica dei parametri da ottimizzare piuttosto che sui parametri stessi. Queste caratteri-

stiche danno agli algoritmi genetici quelle doti di robustezza che permettono loro di risolvere problemi del mondo reale che risultano invece difficili per le altre tecniche. Essi sono sufficientemente generali per poter essere applicati ad una vasta gamma di problemi (è sufficiente trovare una rappresentazione genetica dei parametri del problema), non sono soggetti ai reqisiti di continuità della funzione, non richiedono la visita dell'intero spazio del problema e operano in parallelo su di una popolazione di soluzioni variamente distribuite sulla superficie.

Capitolo 5

Il Master Mind

Il Master Mind è un gioco di 'code-breaking' (decifrazione di codice) inventato nel 1970 da Mordecai Meirowitz, un esperto di telecomunicazioni israeliano. I giocatori sono due: un 'Code Maker' (o 'Mastermind') che definisce una sequenza di colori e la tiene nascosta[1] ed un 'Code Breaker' (o 'Seeker') che deve indovinare la giusta combinazione. Ogni tentativo del 'Cercatore' è valutato dal 'Mastermind' per mezzo di picchetti neri (ognuno dei quali corrisponde ad un colore indovinato nella giusta posizione) e bianchi (per i colori indovinati, ma mal posizionati)[2].

5.1 Giocare a Master Mind

La risoluzione del gioco del Master Mind è un problema molto impegnativo: se da una parte bisogna analizzare un numero di prove piuttosto grande[3], dall'altra si può solo cercare di prevedere il valore della funzione fitness associata alle singole combinazioni: infatti lo scopo principale è quello di minimizzare il numero di tentativi valutati per trovare la soluzione.
Esistono diversi metodi pensati per ottimizzare questo fattore: essi differiscono

[1] Sia i colori possibili che la lunghezza della sequenza sono stabiliti a priori, e, tipicamente, il numero di colori varia da sei a otto, e la lunghezza della sequenza è pari a quattro o cinque.

[2] Nel seguito, per esempio, indicheremo con [2,1] una sequenza che ha due colori nella giusta posizione ed uno mal posizionato.

[3] Nella versione classica del gioco, ossia quella con quattro case e sei possibili colori, il numero di possibili combinazioni è pari a 1296 (6^4); ma già nella versione 'avanzata', cioé con cinque caselle e otto colori possibili, il numero di combinazioni sale a 32768 (8^5).

per molti aspetti, tra i quali l'analisi del problema, il metodo di scelta della sequenza da provare ad ogni turno e la propagazione della conoscenza dovuta alla sua valutazione.
Gli algoritmi genetici sembrano essere una buona soluzione al nostro problema, soprattutto grazie alla capacità di propagare velocemente (e implicitamente) l'informazione di ogni singola prova. L'idea è considerare i colori come i possibili geni e le sequenze di colori come cromosomi: poichè il problema è particolare rispetto ai comuni algoritmi genetici (non possiamo considerare in parallelo un insieme di soluzioni e calcolare il valore di fitness di ognuna, ma possiamo provare solo i cromosomi più promettenti) gli operatori (selezione, crossover e mutazione) saranno a loro volta caratteristici dell'applicazione.

5.2 Strategia

La strategia di ricerca dell'ottimo (cioè della soluzione segreta) avrà come iter la sottomissione al Code Maker di una singola prova alla volta. Di conseguenza, ad ogni passo, l'operatore di selezione dovrà avere come punto di partenza un unico cromosoma, dal quale deriverà la prova successiva. Una volta scelto il cromosoma più promettente, si effettuerà (sempre operando sulla singola prova) un particolare crossover che permetterà di ottenere una nuova sequenza. L'ultimo operatore ad intervenire sarà la mutazione, che modificherà alcuni geni in favore di altri ritenuti più promettenti.
Il passo conclusivo del procedimento sarà il confronto fra la combinazione di geni ottenuta e le prove precedenti: poichè non possiamo permetterci di sprecare un solo tentativo, scarteremo la prova se risulterà essere non consistente (ovvero se sappiamo con certezza che non porterà alla soluzione).

5.2.1 Scelta della Miglior Prova

Il punto di partenza dell'algoritmo è dunque rappresentato dalla scelta del cromosoma più promettente fra quelli valutati dal Code Maker.
Nel Master Mind la funzione di valutazione (che abbiamo utilizzato come funzione fitness) è discreta e non continua. In particolare potremmo descriverla

come una funzione 'a gradino' multidimensionale, che ha per dominio l'insieme di tutte le possibili combinazioni dei colori di lunghezza fissata.
Il valore della funzione in un punto (cioé per una particolare combinazione) è espresso però come una coppia di numeri (i picchetti neri e bianchi descritti all'inizio del capitolo); questo provoca un problema nel momento in cui si devono confrontare due prove per scegliere la migliore: non esiste un unico metodo di confronto oggettivamente migliore, ma dobbiamo imporre un criterio di valutazione che abbia comunque un certo fondamento.
Seguendo l'idea di Alexandre Temporel e Tim Kovacs[4] consideriamo a una prova migliore di b se:

• il numero di colori di a presenti nella sequenza segreta è maggiore del numero di colori di b presenti nella sequenza segreta;
• a parità di numero di colori indovinati, il numero di picchetti neri ottenuti come valutazione di a è maggiore di quello ottenuto come valutazione di b;
• a parità di numero di colori indovinati e di picchetti neri, a è stata giocata successivamente a b.

Mentre le prime due condizioni seguono la logica del favorire una maggior conoscenza, anche a discapito della precisione della soluzione, la terza condizione è un atto puramente formale per stabilire un'unica sequenza in caso di ambiguità.
Chiameremo la sequenza così trovata 'Prova Corrente Favorita' (**CFG**, dall'inglese 'Current Favourite Guess').
È importante notare come la funzione di valutazione che consideriamo ha il potere di determinare sia l'individuo per la riproduzione (non in maniera probabilistica, ma deterministica), che (come vedremo in seguito) il numero di geni da conservare per la generazione successiva (quindi sapremo esattamente il numero di geni che verranno mutati).

5.2.2 L'operatore di crossover

Il secondo passo che dobbiamo compiere è utilizzare il crossover per modificare la Prova Corrente Favorita.

[4] Alexandre Temporel e Tim Kovacs, 'A heuristic hill climbing algorithm for Mastermind'.

L'operatore di crossover di cui ci serviamo utilizza il CFG e la sua valutazione per creare una nuova combinazione di colori: la valutazione di una sequenza ci fornisce l'informazione di quanti geni abbiamo indovinato nella giusta posizione e di quanti ne abbiamo indovinati in una posizione sbagliata.
L'operatore si basa su questi dati per creare il 'figlio':

• si estraggono in maniera casuale dalla sequenza tanti colori quanti sono i picchetti neri e li si inseriscono in una nuova sequenza nella stessa posizione in cui erano nel CFG;
• si estraggono in maniera casuale dall'insieme dei colori della sequenza non ancora considerati, tanti colori quanti sono i picchetti bianchi e li si inseriscono nella nuova sequenza appena creata in una posizione differente rispetto a quella del CFG;

Chiameremo la nuova sequenza appena creata (che potrebbe essere ancora incompleta) 'Nuovo Codice Potenziale' (**NCP**).
Con questo metodo cerchiamo di sfruttare al meglio il risultato ottenuto con il CFG: si suppone in pratica che i colori scelti per il Nuovo Codice Potenziale siano quelli relativi ai picchetti neri e bianchi della valutazione; il nome esprime quindi il fatto che potenzialmente la nuova sequenza potrebbe essere il codice segreto.

5.2.3 L'operatore di mutazione

Il crossover si occupa di sfruttare l'informazione ottenuta tramite la valutazione del CFG creando una nuova combinazione di colori. La sequenza generata può però risultare incompleta. L'operatore di mutazione si occupa di 'riempire' questi vuoti con nuovi geni, secondo un criterio probabilistico di estrazione che si basa sulle precedenti presenze del gene nelle prove valutate: l'idea è che migliori sono state le valutazioni dove il gene era presente, più alta sarà la probabilità di estrarre quel gene.
Un compito importante di questo operatore è anche quello di introdurre nuovi geni per effettuare una migliore esplorazione dello spazio di ricerca, in accordo con la teoria.

5.2.4 Consistenza

Una importante osservazione fatta da Darby[5] è che, nel Mastermind, la valutazione è commutativa: se noi invertiamo il ruolo tra la combinazione segreta e una qualsiasi prova, la valutazione rimane la stessa.
Una conseguenza importante di questa osservazione è che ad ogni passo noi possiamo confrontare il codice potenziale con le prove che abbiamo già fatto: ci aspettiamo che il risultato di questo confronto sia uguale alle valutazioni ottenute dalle prove giocate. Se noi selezionassimo tutte le possibili combinazioni in base alle possibili valutazioni delle prove, ci aspetteremmo di trovare la sequenza segreta nel gruppo con valutazione pari alle prove precedenti.
Se anche uno solo dei confronti tra l'NCP e le prove passate risultasse differente dalla valutazione della prova, avremmo la certezza che la sequenza in nostro possesso non sarebbe il codice segreto: l'NCP verrebbe quindi scartato e si procederebbe alla ricerca di un nuovo codice potenziale.
Se accettassimo l'NCP come nuova prova, esso potrebbe essere la combinazione segreta; se non lo è, nel peggiore dei casi, avrà ristretto il nostro spazio di ricerca.

Naturalmente è poco efficiente valutare la consistenza di un cromosoma solo al termine della creazione dello stesso, quindi il processo di controllo ha luogo durante tutte le fasi della riproduzione (potremmo parlare di una 'creazione guidata') ed alla fine si effettua solamente un controllo formale.

5.3 Un esempio pratico

Vediamo ora in pratica un esempio di ricerca del codice[6].
Supponiamo che il Master Mind scelga come sequenza segreta $(1, 5, 1, 3)$[7], scegliendo tra sei possibili 'colori'.
La prima prova sarà del tutto casuale: supponiamo per esempio di avere scelto

[5] Gary Darby, 'Delphi for fun', 'www.delphiforfun.org/Programs/MasterMind.htm'.

[6] L'esempio qui riportato è stato ottenuto facendo girare il codice sviluppato, con il comando 'mmag8(soluz=c(1,5,1,3),ppresi=0.32,ptpresi=0.73,kkk=7.5,seme=2,ncol=6)'.

[7] D'ora in poi identificheremo i colori come numeri interi, quindi ogni volta che parleremo di 'colore', intenderemo 'il numero intero che corrisponde ad un certo colore reale'.

la sequenza $(3, 6, 5, 3)$.
Quindi avremo:

Secret code: (1,5,1,3)

1: 3 6 5 3 [1,1]

La sequenza, poiché al momento è l'unica, sarà anche la migliore prova (CFG). Operiamo quindi su di essa per mezzo del crossover: estraiamo dalla sequenza tanti colori quanti sono i picchetti neri (uno in questo caso) e li inseriamo in una nuova sequenza (NCP) nella stessa posizione; estraiamo poi dai rimanenti colori della sequenza tanti colori quanti sono i picchetti bianchi (ancora uno) li inseriamo nell'NCP in un posizione differente.
Supponiamo di avere estratto in maniera casuale come 'nero' il '3' in quarta posizione e come 'bianco' il '5'. Modifichiamo quindi (sempre in maniera casuale) la posizione del bianco estratto. L'NCP sarà, per esempio, $(5, *, *, 3)$, dove indichiamo con il simbolo '*' un gene mancante. Notiamo in questa scrittura l'evidente analogia con la Teoria degli Schemi: lo schema considerato verrà mantenuto (a meno di incoerenza con prove precedenti) e valutato.
Per completare la sequenza utilizziamo l'operatore di mutazione: come descritto sopra saranno favoriti nell'estrazione i geni 'nuovi' (cioè non ancora utilizzati). Completata la sequenza, ne controlleremo la consistenza. In questo caso, essendo la seconda prova derivata direttamente dalla precedente, non ci saranno problemi.
La situazione sarà quindi la seguente:

Secret code: (1,5,1,3)

1: 3 6 5 3 [1,1]
2: 5 2 4 3 [1,1]

In accordo con quanto detto in precedenza considereremo questa prova come la nostra 'Prova Corrente Favorita' (ricordiamo infatti che, a parità di picchetti bianchi e neri, il CFG sarà la prova giocata più recentemente). Partendo quindi dall'ultima sequenza, opereremo in maniera analoga a quanto descitto sopra e otterremo la terza prova:

Secret code: (1,5,1,3)

1: 3 6 5 3 [1,1]
2: 5 2 4 3 [1,1]
3: 5 3 5 1 [0,3]

La terza sequenza risulta essere migliore delle precedenti; essa sarà quindi il nostro nuovo punto di partenza.
Notiamo innanzi tutto che le estrazioni dei geni non saranno puramente casuali, ma saranno estrazioni casuali pesate[8]. In particolare i colori maggiormente considerati saranno:

• '1', dato che l'unica presenza di questo colore coincide con un incremento della valutazione della prova;
• '5', dato che siamo certi della presenza di almeno un '5' nella soluzione (dalla terza prova);
• '6', dato che è presente una sola volta (nella prima prova) e la sua esclusione non ha portato a particolari vantaggi.

Al contrario i colori meno 'pesanti' dal punto di vista probabilistico saranno:

• '2' e '4' perché la loro esclusione nel passaggio dalla seconda alla terza prova ha portato ad un miglioramento della valutazione;
• '3' perché è il colore più presente nelle prove (ben quattro presenze in tre prove).

Il primo passo è estrarre tre geni dal CFG e cambiarne la posizione per mezzo dell'operatore di crossover. Come già accennato, per questioni di efficienza, verificheremo subito la consistenza della sequenza generata: supponiamo per esempio di trovare come NCP $(1, *, 3, 5)$. Questo sarebbe consistente con la terza prova, ma non con le prime due: se lo confrontiamo per esempio con la prima, otteniamo [0,2], valore differente dalla valutazione conosciuta. Analogamente la sequenza $(3, *, 1, 5)$ sarà consistente con la prima e la terza prova, ma otterrà un risultato di [0,2] se confrontato con la seconda.
Trovato un codice potenziale che ci soddisfi, procediamo quindi con l'operatore di mutazione per completare la sequenza. Notiamo che anche con questo

[8]Per una spiegazione più dettagliata del calcolo dei pesi si rimanda alla prossima sezione.

operatore ci adoperiamo in modo da assicurare una maggiore efficienza: supponiamo che l'NCP trovato sia $(*, 5, 1, 3)$ (consistente con le tre prove); prima di procedere all'estrazione del gene mancante facciamo una 'scrematura' dell'insieme dei possibili colori.
L'idea è quella di inserire un possibile colore nella posizione mancante e valutare la consistenza della sequenza completa trovata rispetto alle tre prove. Se per esempio inseriamo un '6', l'NCP sarà $(6, 5, 1, 3)$: esso risulterà non consistente con la prima prova (valutazione [1,2]) e quindi escluderemo il '6' dall'insieme dei colori possibili.
Analogamente escluderemo i colori '2','4','5' (non compatibili con la seconda prova) e '3' (per la prima). Ci rimarrà quindi solamente un colore, che completerà la sequenza a $(1, 5, 1, 3)$.
Dopo aver visto che la sequenza è consistente, cosa che, in questo caso, è una semplice formalità, la giochiamo:

Secret code: (1,5,1,3)

1: 3 6 5 3 [1,1]
2: 5 2 4 3 [1,1]
3: 5 3 5 1 [0,3]
4: 1 5 1 3 [4,0]

La nostra ricerca è dunque terminata con la scoperta del codice segreto al quarto tentativo.

5.4 Spiegazione del codice R

Il codice seguente è stato scritto seguendo i passi descritti sopra e prendendo spunto dall'articolo di Alexandre Temporel e Tim Kovacs, 'A heuristic hill climbing algorithm for Mastermind'[9].
Nell'articolo i due autori, dopo aver descritto il problema, presentano come possibile soluzione al gioco del Master Mind un algoritmo derivato dal 'Ran-

[9]Reperibile sul web all'indirizzo 'http://www.cs.bris.ac.uk/Publications/pub_info.jsp?id=2000067'.

dom Mutation Hill Climbing' adattato al Master Mind[10] chiamato dagli stessi autori 'Stochastic Hill Climber'. Esso si propone di minimizzare due valori del gioco: il numero di prove necessarie per scoprire la combinazione segreta e il numero di combinazioni (prove potenziali) analizzate, ma non utilizate come tentativi.
L'algoritmo proposto suggerisce in particolare l'operatore di crossover e di mutazione descritti in precedenza, ma non indica come calcolare i pesi dei singoli geni nelle numerose estrazioni casuali presenti nell'algoritmo. Anche la scelta del metodo di generazione delle singole prove, in modo da minimizzare il numero di 'giri a vuoto', e altri piccole ottimizzazioni sono lasciate al lettore.
Analizziamo in dettaglio il codice.

5.4.1 Parametri di input e variabili

Innanzi tutto si possono notare un gran numero di parametri di input (ben sei): questi servono per la selezione del Nuovo Codice Potenziale, per impostare il codice segreto (che il programma dovrà scoprire) e per assegnare il seme di generazione casuale dei numeri.

In particolare:

- 'soluz' è un vettore che rappresenta la soluzione scritta in forma numerica (di default soluz = sample(c(1:8),5,replace=TRUE)). È da notare che la lunghezza della sequenza da forzare può essere scelta dall'utente semplicemente variando la lunghezza del vettore in input;

- 'ppresi' è un numero usato per modificare la probabilità di estrazione dei colori già presenti nell'NCP prima della mutazione (di default ppresi = 0.32). Se per esempio, dopo aver effettuato il crossover, il mio NCP è $(3, 1, *, *)$, le probabilità associate ai due colori '3' e '1' verranno moltiplicate per 'ppresi';

- 'ptpresi' è analogo al precedente, ma influisce sui colori appena estratti per

[10] Eric B. Baum, Dan Boneh, and Charles Garrett. 'Where Genetic Algorithms Excel'. Reperibile presso la pagina internet 'http://citeseer.ist.psu.edu/baum95where.html'.

mezzo dell'operatore di mutazione (di default ppresi = 0.73). Per esempio, se devo estrarre tramite l'operatore di mutazione due colori per completare l'NCP, estrarrò il primo, modificherò la probabilità associata al colore appena estratto moltiplicandola per 'ptpresi' ed infine estrarrò il secondo colore;

- 'kkk' serve ad aumentare la probabilità di estrazione dei colori non ancora provati (di default ppresi = 7.5) perché vengano scelti con maggiore facilità dall'operatore di mutazione. Uno degli scopi principali di questo operatore, infatti, è quello di introdurre un certa dose di esplorazione nell'algoritmo. Come le due variabili precedenti, anche questa viene semplicemente moltiplicata per il valore della probabilità associata al colore;

- 'seme' serve ad impostare il seme per la generazione casuale dei numeri (di default seme = NULL);

- 'ncol' indica il numero di colori possibili fra i quali si deve scegliere per cercare la sequenza segreta (di default ncol = 8).

Molte sono anche le variabili utilizzate nel codice. Possiamo però raggrupparle in tre gruppi:

- le variabili di utilità, che sono utilizzate dalle altre variabili e servono principalmente a rendere più generale l'algoritmo, come 'ncolori' (numero di colori possibili), 'ncase' (lunghezza della sequenza da scoprire) o 'possibili' (i colori possibili, cioé che potrebbero comparire nella soluzione);

- le variabili per il mantenimento in memoria delle prove, come 'mm' (matrice di dati in cui sono memorizzati i tentativi precedentemente fatti di scoprire la soluzione, con i rispettivi risultati), 'cseq' (vettore in cui vengono tenute in memoria alcune sequenze 'cattive' per velocizzare la ricerca dell'NCP), 'ncp' (vettore in cui viene memorizzato il Nuovo Codice Potenziale) o 'cfg' (vettore contenente la Prova Corrente Favorita);

- le variabili utilizzate per il calcolo delle probabilità di estrazione, come 'probabilita' (vettore dei pesi dei colori durante le estrazioni) o 'soluzione'

(vettore contenente il codice da scoprire).

Vi sono inoltre numerose altre variabili di minore importanza. Per una lettura del codice si rimanda all'appendice.

La parte iniziale di codice serve per inizializzare le principali variabili e per effettuare alcuni controlli sui parametri di input. In seguito viene estratta la prima prova, completamente casuale, da sottoporre al 'Code Maker'. Questa prova diventa la nuova Prova Corrente Favorita (CFG); il Nuovo Codice Potenziale (NCP) deriverà da essa, non essendoci alcuna altra prova precedente.
Il programma inizia quindi a cercare il codice segreto seguendo la strategia descritta nelle sezioni precedenti. Nei prossimi paragrafi descriveremo alcune piccole particolarità del codice.

5.4.2 La correzione delle probabilità

Come abbiamo visto in precedenza, non esiste una probabilità di mutazione intesa come possibilità che un gene venga mutato: la valutazione della Prova Corrente Favorita indica esattamente quanti geni mutare (anche se non ci dice quali).
La correzione delle probabilità[11] dei singoli colori si basa su alcuni semplici passi. Per ogni sequenza provata e valutata dal Master Mind:

- si aggiornano i vettori di frequenza (per i colori presenti nella sequenza corrente) e di non-frequenza (per i colori assenti);

- si calcola la valutazione media dei colori presenti ('valutazione della sequenza'[12] / 'lunghezza della sequenza') e di quelli mancanti (('Lunghezza della sequenza' - 'valutazione della sequenza') / 'numero di colori mancanti');

[11] D'ora in avanti, quando parleremo di probabilità intenderemo 'il peso non normalizzato assegnato ad un certo elemento'. Questo piccolo abuso ci permetterà di esprimerci con maggiore semplicità.

[12] La valutazione della sequenza è intesa come semplice somma dei colori indovinati ('bianchi' + 'neri').

- si aggiorna la media delle valutazioni per i singoli colori, sia quando sono presenti nelle prove (vettore 'media_pres'), sia quando sono assenti ('media_ass');

- si confronta la sequenza provata con la Prova Corrente Favorita da cui è stata generata e si attribuisce la differenza di valutazione ai nuovi colori introdotti e (con segno inverso) ai colori che sono stati scartati dal CFG (vettore 'cambi');

- si calcola la probabilità generale (vettore 'probabilita'), semplicemente sommando i vettori ottenuti ('cambi', 'media_pres' e 'media_ass') ed una costante ('10');

- se il colore non è mai stato utilizzato (quindi la corrispondente casella del vettore 'frequenza' sarà uguale a zero), si moltiplica la probabilità corrispondente per *kkk* (parametro fornito in input all'utente).

Le probabilità così trovate vengono inoltre aggiornate prima e durante l'esecuzione dell'operatore di mutazione:

• in seguito al crossover, se il colore è presente nell'NCP, si moltiplica la probabilità corrispondente per *ppresi*;
• in seguito ad ogni estrazione fatta dall'operatore di mutazione, se il colore è stato estratto dall'operatore, si moltiplica la probabilità corrispondente per *ptpresi*.

Le probabilità così trovate vengono quindi utilizzate per le estrazioni dei geni nei vari passi dell'algoritmo.

5.4.3 Consistenza delle prove

Nel gioco del Master Mind lo scopo primario è minimizzare il numero di tentativi. È quindi molto importante che le prove siano **consistenti** con le sequenze valutate in precedenza. In altre parole ogni sequenza provata deve poter essere potenzialmente la sequenza segreta. Poichè la generazione di strighe senza particolari vincoli provocherebbe un aumento enorme di 'giri a vuoto' (perchè le stringhe non consistenti sarebbero scartate), nell'algoritmo si cerca di guidare

la scelta eliminando quei colori che provocherebbero sicuramente uno scarto. Per questo scopo nel codice è presente una opportuna funzione: la 'accseq', i cui parametri di input sono: 'ncp' (il nuovo codice potenziale da testare), 'possibili' (i possibili colori da estrarre), 'mm' (matrice con le prove precedenti) e 'i'(numero della prova corrente). Questa serve a stabilire se il frammento di Nuovo Codice Potenziale ('ncp') proposto, eventualmente completato dai colori 'possibili', può portare ad una sequenza accettabile o meno (confrontata con le 'i' prove precedenti, tenute in memoria nella matrice 'mm'). Nel primo caso, appena viene trovata una sequenza coerente con le prove precedenti, la funzione segnala la 'bontà' del frammento; nel secondo caso, non trovando sequenze accettabili, la funzione segnala al programma che il frammento deve essere scartato.
Mediamente, quindi, il programma utilizzerà molto tempo per verificare la coerenza dei singoli frammenti, ma la convergenza (ossia la scoperta del codice segreto) è certa in un tempo ragionevole.

5.5 Analisi Statistica sulla bontà dell'algoritmo

Lo scopo principale del nostro codice è minimizzare il numero di prove effettuate per trovare il codice segreto e, in seconda battuta, minimizzare il tempo di calcolo (ovvero i 'giri a vuoto' provocati dallo scarto di sequenze non consistenti). Come termine di confronto per valutarne l'efficienza abbiamo usato i risultati di altri algoritmi. Inoltre, per capire la reale potenza degli algoritmi genetici, abbiamo sviluppato un codice alternativo che risolve il Master Mind secondo un criterio di minimizzazione del numero medio di tentativi.
Tutte le prove sono state fatte con *R*[13] nella versione 2.2.0 sotto *WindowsXP*, utilizzando un *Acer Travelmate 291LMI (Intel Pentium M 1.4 GHz, 512 MB DDR SDRAM)*.

[13]Scaricabile liberamente da internet, all'indirizzo 'www.r-project.org'

5.5.1 Il codice alternativo

Il nuovo codice sviluppato nasce dalla necessità di poter valutare il Master Mind 'Genetico' sia in termini di velocità di esecuzione, che in termini di minimizzazione di prove.
L'idea che abbiamo seguito nello scrivere il codice è la seguente: supponiamo di avere un insieme di soluzioni possibili $\{x_1, ..., x_k\}$, ognuna di esse avrà una probabilità $1/k$ di essere il codice segreto. Il numero di possibili combinazioni che rimarrebbero se noi giocassimo un certa sequenza x_i sarà quindi:

Prova	*Soluzione*	*Rimasti*	*Probabilità*
x_1	x_1	$n_{1,1} = 0$	$1/k$
x_1	x_2	$n_{1,2}$	$1/k$
x_1	x_3	$n_{1,3}$	$1/k$
...	...	...	...
x_k	x_k	$n_{k,k} = 0$	$1/k$

Tabella 5.1: Possibili sequenze rimaste.

Il numero medio di combinazioni che rimarrebbero se sottoponessi al Master Mind la sequenza x_i è dunque

$$\overline{n}_{i,*} = \frac{\sum_{j=1}^{k} n_{i,j}}{k}$$

La minimizzazione di questa quantità ci dovrebbe portare ad una soluzione in un numero basso di tentativi.

Nella pratica questo algoritmo ha il suo punto debole nella gestione della matrice delle possibili prove e nel calcolo della formula riportato qui sopra: la matrice infatti deve contenere tutte le sequenza ancora possibili, per calcolare il valore associato ad ognuna di esse, mentre, per quel che riguarda il calcolo, notiamo che, se ad un certo momento avessimo una matrice con k righe, dovremmo effettuare '$k * (k-1)$' operazioni di confronto.

Per affrontare questi problemi abbiamo tentato alcune modifiche nel codice, ma i risultati non sono sempre stati positivi.

Una importante considerazione è che, dal punto di vista matematico, prima di effettuare qualsiasi prova le sequenze possono essere divise in gruppi a seconda del numero e della frequenza dei singoli colori. Se per esempio consideriamo il caso del classico Master Mind con quattro case e sei colori possibili, potremo dividere l'insieme di tutte le possibili combinazioni in cinque insiemi:

- le soluzioni del tipo '(a, a, a, a)';
- le soluzioni del tipo '(a, a, a, b)';
- le soluzioni del tipo '(a, a, b, b)';
- le soluzioni del tipo '(a, a, b, c)';
- le soluzioni del tipo '(a, b, c, d)';

con a, b, c, d colori possibili.
Questa semplificazione porta ad una drastica riduzione del numero di calcoli (nell'esempio considerato dovremo fare $5 * 1296 = 6480$ confronti anziché $1296 * 1295 = 1678320$).
Nel momento in cui riceviamo delle nuove informazioni (cioé giochiamo la sequenza e otteniamo la valutazione) questo discorso non può più essere fatto e dobbiamo tornare alla regola del '$k * (k - 1)$' perché la conoscenza acquisita modifica il valore delle differenti prove.
Per ridurre questo numero abbiamo provato a scrivere un secondo codice che tenesse in memoria alcuni confronti: è chiaro infatti che se confrontiamo la i-esima sequenza con la j-esima, il risultato sarà equivalente al confronto tra la j-esima e la i-esima. Purtroppo, nonostante il numero di confronti venga ridotto notevolmente (da '$k*(k-1)$' a '$k+(k-1)+...+(k-(k+1))+(k-k)$'), il mantenimento in memoria e la gestione della corrispondente matrice di dati provoca un aumento del tempo utilizzato superiore al risparmio (sempre in termini di tempo) dato dal minor numero di operazioni.

Per ottimizzare questo algoritmo si potrebbero compiere ulteriori analisi e prove: per esempio si potrebbe studiare come nuove informazioni influiscono sulla distribuzione delle possibili sequenze oppure stabilire quali tipologie di sequenze diano mediamente maggiori informazioni (per esempio abbiamo notato che, come prima prova, le sequenze di tipo '(a, a, b, c)' o, nel Master Mind con cinque case e otto colori, '(a, a, b, c, d)' sono preferibili per minimizzare il

numero medio di prove). Queste analisi vanno però ben oltre i nostri obbiettivi; inoltre potrebbero rendere il metodo troppo legato a certe condizioni e quindi meno robusto.

Per la lettura del codice R commentato, si rimanda il lettore all'appendice.

5.5.2 Confronto

Per effettuare un confronto abbiamo utilizzato i risultati di vari algoritmi sviluppati da diversi programmatori:

• Radu Rosu[14], che usa una particolare ricerca pseudo-casuale (guidata però per alcuni aspetti);
• Luís Bento[15], J.J. Merelo[16], Alexandre Temporel e Tim Kovacs[17], che si basano sugli algoritmi genetici;

oltre naturalmente ai nostri due codici.
I risultati dei programmi sono riportati nella tabella (5.2).

	Numero di prove	
	4x6	**5x8**
Rosu	4.66	5.88
Bento	-	6.86
Merelo	4.132 (sd = 1.00)	5.904 (sd = 0.975)
Temporel - Kovacs	4.64 (sd = 0.857)	5.834 (sd = 0.934)
Codice Alternativo	4.335 (sd = 0.6983)	-
MM Genetico	4.646 (sd = 0.88072)	5.897 (sd = 0.95955)

Tabella 5.2: Numero medio e deviazione standard delle prove.

[14]'Analysis of the game of Mastermind - the m^n case.', reperibile sul web all'indirizzo 'http://www.csc.ncsu.edu/academics/undergrad/Reports/rtrosu/index.html'.
[15]'MasterMind by Evolutionary Algorithms.', reperibile sul web all'indirizzo 'http://laseeb.isr.ist.utl.pt/publications/papers/acrosa/sac99/sac99.html'.
[16]'Finding a needle in a haystack using hints and evolutionary computation: the case of Genetic Mastermind.', reperibile sul web all'indirizzo 'http://geneura.ugr.es/~jmerelo/newGenMM/newGenMM.html'.
[17]'A heuristic hill climbing algorithm for Mastermind.', reperibile sul web all'indirizzo 'http://www.cs.bris.ac.uk/Publications/pub_info.jsp?id=2000067'.

I nostri dati sono stati ottenuti simulando la distribuzione delle possibili sequenze: per questo nel mastermind con quattro case e sei colori abbiamo provato il codice utilizzando come sequenza segreta tutte le possibili sequenze del tipo $(1, *, *, *)$ (216 possibili combinazioni, ognuna provata con cento possibili semi differenti di generazione casuale), mentre per il mastermind con cinque case e otto colori abbiamo utilizzato le possibili sequenze del tipo $(1, *, *, *, *)$ (4096 possibili combinazioni, ognuna provata con cento possibili semi differenti di generazione casuale).
Notiamo innanzi tutto che il codice alternativo risulta migliore per il *4x6*, ma, poiché il numero di calcoli (e di conseguenza il tempo di calcolo) risulta eccessivo, per il *5x8* non possiamo fornire statistiche soddisfacenti. Questo sottolinea ancora una volta la potenza di 'propagazione implicita' dell'informazione data dagli algoritmi genetici.

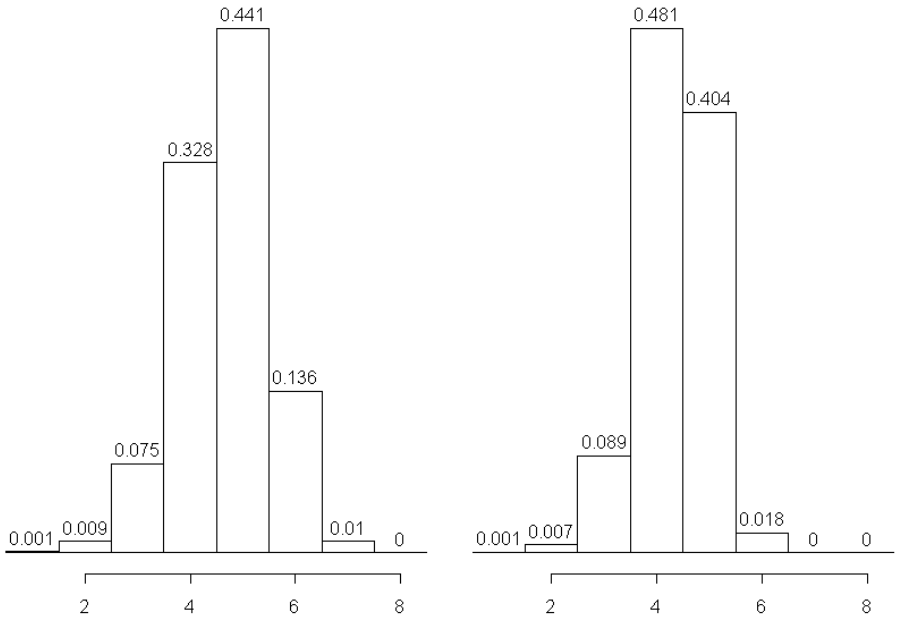

Figura 5.1: *Confronto fra i risultati del Master Mind Genetico (a sinistra) e del Codice Alternativo (a destra) considerando quattro case e sei colori.*

Possiamo dunque dire che gli algoritmi genetici hanno successo con maggior difficoltà quando si può ottimizzare in tempi ragionevoli una funzione con

metodi esatti, mentre mostrano la loro potenza proprio dove il tempo di calcolo non è più sostenibile con altri algoritmi. Nei nostri test abbiamo infatti notato come il fattore tempo sia già presente nel Master Mind *4x6*: mentre il codice genetico raramente supera i due decimi di secondo di calcolo di cpu, il secondo programma impiega fino a sedici secondi (la maggior parte dei quali viene utilizzato per analizzare le sequenze rimaste dopo la prima prova).
Come abbiamo già notato, il Master Mind Genetico ottiene risultati soddisfacenti soprattutto per il caso del *5x8*; inoltre, come possiamo vedere nella tabella (5.3), il codice sviluppato dovrà valutare mediamente meno sequenze rispetto agli altri.

	Numero di valutazioni	
	4x6	**5x8**
Rosu	1295	32515
Bento	-	1029.9
Merelo	279 (sd = 188)	2171 (sd = 1268)
Temporel - Kovacs	41.2 (sd = 49.0)	480.1 (sd = 826.6)
MM Genetico	30.88 (sd = 106.88)	155 (sd = 999.49)

Tabella 5.3: Numero medio e deviazione standard del numero di valutazioni.

Infine possiamo ritenerci soddisfatti anche per quel che riguarda il numero massimo di prove: secondo una delle regole del gioco del Master Mind 'commerciale', colui che deve 'forzare' il codice perde se non riesce ad indovinarlo in dieci tentativi. Come si può vedere nella figura (5.2) in 409600 prove del *5x8* questo limite non è stato mai superato e solo in 12 occasioni (pari allo 0.003%) è stato raggiunto (basti pensare che la fortunosa scoperta in un'unica soluzione del codice è stata fatta 13 volte !!!).

Possiamo quindi ritenerci soddisfatti dei risultati ottenuti dal codice sviluppato. Un miglioramento potrebbe essere possibile grazie ad alcuni accorgimenti: per esempio si potrebbe effettuare un'analisi delle sequenze e delle interazioni fra di esse, per indirizzare la ricerca del codice verso cammini più brevi.

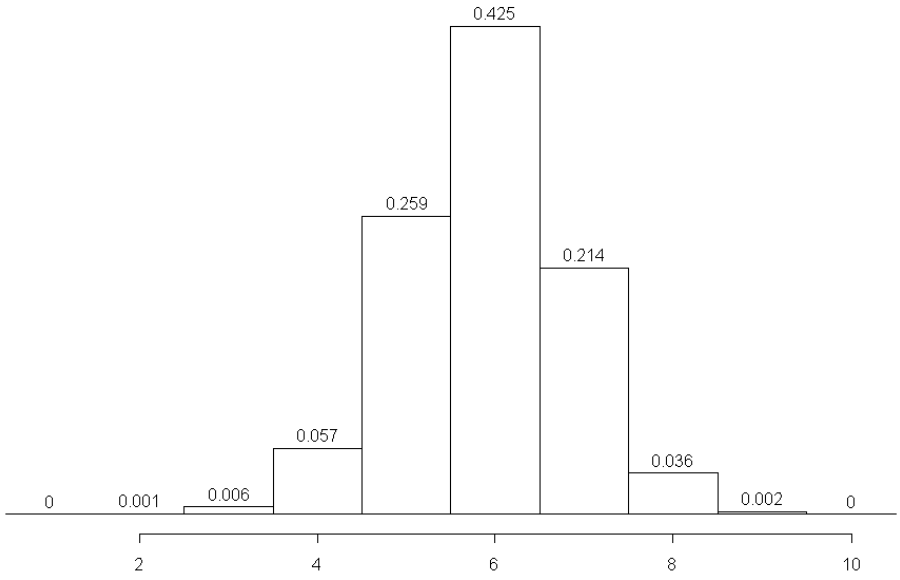

Figura 5.2: *Distribuzione delle numero di prove per il Master Mind Genetico con cinque case e otto colori.*

Queste aggiunte potrebbero quindi migliorare il codice, ma potrebbero anche renderlo dipendente dalle prove effettuate facendolo diventare troppo specifico e comprometterne la robustezza.

Capitolo 6

Il Problema del Commesso Viaggiatore

Il problema del commesso viaggiatore, o 'TSP' (dall'inglese *Traveling Salesman Problem*) è un problema di teoria dei grafi che potremmo definire 'storico' e che ha catturato le attenzioni e gli sforzi di molti studiosi allo scopo di trovarne una soluzione nei casi più complicati.

Il problema è concettualmente molto semplice: '*si vogliono visitare n città diverse, tornando al punto iniziale, senza ripassare per una città già vista. Dati i costi di viaggio tra le città, come si può organizzare l'itinerario per minimizzare la spesa totale?*'

Nonostante questa apparente semplicità, la soluzione è tutt'altro che banale, tanto che questo problema è inquadrato nella classe dei problemi *NP-completi* ('Non deterministico Polinomiale - tempo Completo'), cioé in pratica è irrisolvibile con un algoritmo lineare standard se la complessità del problema è abbastanza elevata: in un problema con n città bisognerebbe calcolare $(n-1)!$ (($n-1$) fattoriale) possibili cammini, nel peggiore dei casi, per trovare il percorso minimo. Lo spazio di ricerca per il TSP è infatti l'insieme di tutte le possibili permutazioni delle n città.
Ogni singola permutazione produce una soluzione, ovvero un tour completo delle n città. La soluzione ottima è la permutazione che produce i minimi costi di viaggio.

Il problema è di considerevole importanza pratica, al di là delle ovvie applicazioni nella logistica e nei trasporti. Un esempio classico è la costruzione di circuiti stampati, nella pianificazione del percorso del trapano per creare i fori nella piastra. Nelle applicazioni di foratura o di rifinitura automatica eseguite da robot, le 'città' sono pezzi da rifinire o fori (di varie dimensioni) da praticare, e il 'costo di viaggio' include i necessari tempi morti.
Durante gli ultimi decenni sono stati proposti molti algoritmi per trovare la soluzione ottima[1] ed anche la comunità dei ricercatori degli algoritmi genetici ha mostrato un forte interesse per il problema.
Le strategie di soluzione più comuni possono essere divise in tre grandi gruppi:

- progettare algoritmi per trovare la soluzione esatta, ragionevolmente veloci solo per problemi con un numero di 'città' relativamente basso;

- progettare algoritmi euristici, cioè algoritmi che producono soluzioni probabilmente buone, ma impossibili da provare essere ottimali (gli algoritmi genetici appartengono a questa categoria);

- trovare un caso specifico del problema (sottoproblema) per il quale sia possibile o una soluzione esatta o un'euristica migliore.

Per quel che riguarda gli algoritmi genetici, la strategia di ricerca della soluzione ottima sarà in questo caso più 'affine', rispetto all'esempio precedente del Master Mind, alla teoria tradizionale formulata da Holland: ci saranno due genitori per ogni figlio, avremo la possibilità di testare ad ogni generazione una popolazione di potenziali soluzioni e di manipolarle secondo operatori (selezione, crossover e mutazione) meno 'caratteristici'. Questi possono comunque assumere numerose forme e varianti: nei prossimi paragrafi vedremo i principali operatori proposti nel corso degli anni.

[1] Si veda D.S. Johnson, 'Local Optimization and the Traveling Salesman Problem' (1990) in 'Proceedings of the 17^{th} Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science (pp.446-461)' di M.S. Paterson, Springer-Velag.

6.1 Codifica

I ricercatori di algoritmi genetici sono abbastanza d'accordo sul fatto che la rappresentazione binaria non è la più adatta per il TSP: trasformare una stringa di interi (1, 2, ..., n), dove ogni numero corrisponde ad una città, in una stringa binaria, è un'operazione che non offre alcun vantaggio. Ciò non significa che sia impossibile progettare un algoritmo genetico per il TSP con rappresentazione binaria e operatori genetici classici, ma che in tale problema questa non è una scelta naturale.
D'altronde anche la rappresentazione mediante vettore di interi non è esente da difetti: gli operatori genetici classici producono frequentemente doppioni di alcune città e assenza di altre. Questo significa che sarà necessario imporre qualche variazione agli operatori genetici affinché siano in grado di ricombinare le informazioni provenienti dai genitori in modo significativo, producendo discendenti validi.

Sono stati proposti molti approcci, che si diversificano sia per il modo di codificare il problema, che per gli operatori genetici usati. È possibile compiere una divisione in due gruppi principali: gli algoritmi che utilizzano una rappresentazione tramite vettori e quelli che si servono di matrici. Nelle prossime sezioni vedremo le principali idee proposte nelle due categorie.

6.1.1 Rappresentazione con vettori

Sono state proposte tre differenti rappresentazioni, ognuna delle quali introduce operatori genetici propri. Per facilitare la comprensione verrà usato un esempio con nove città numerate da '1' a '9'.

Rappresentazione adiacente

Un percorso è codificato in **rappresentazione adiacente** se in una lista di n città, la città j è elencata nella posizione i se e solo se ci si muove dalla città i alla città j.
Per esempio il vettore (3,4,5,8,9,7,2,1,6) rappresenta il percorso (1 3 5 9 6 7 2 4

8). In questo modo ogni itinerario è rappresentato con una sola lista (in altre parole due itinerari diversi avranno sicuramente una rappresentazione diversa), mentre alcune stringhe non rappresentano degli itinerari validi: infatti si possono formare dei percorsi parziali con la presenza di cicli, come, ad esempio, (3,8,5,9,4,7,2,6,1) che si frammenta in (1 3 5 4 9) e (2 8 6 7).
Naturalmente questo fenomeno è di ostacolo alla risoluzione del problema. Inoltre l'operatore di incrocio tradizionale non è adatto per questa rappresentazione, perché porta al fenomeno della frammentazione in cicli e per usarlo sarebbe necessaria la continua correzione con un algoritmo riparatore. Per ovviare a questo problema sono stati definiti numerosi operatori d'incrocio che però risultano piuttosto complicati e spesso poco funzionali.

Rappresentazione ordinale

Un percorso è codificato in **rappresentazione ordinale** se schematizza un cammino in modo che, in una lista di n città, l'*i-esimo* elemento della lista è un numero tra 1 e $(n - i + 1)$. L'idea su cui si basa è la seguente: data una lista ordinata di città che serve come punto di riferimento per i cammini, il vettore che rappresenta il cromosoma indica passo dopo passo la posizione della successiva città nel tour rispetto alla lista di riferimento.
Se per esempio consideriamo come lista ordinata di riferimento la lista $c =$ (1,2,3,4,5,6,7,8,9), il tour (3 5 1 6 2 8 7 9 4) è rappresentato con la lista di riferimenti $l =$ (3,4,1,3,1,2,2,2,1), che viene interpretata come segue:

• il primo numero della lista l è 3; quindi si prende la terza città della lista c, che diventerà la prima città del tour, e la si toglie dalla lista c. Il tour parziale è (3 ...).
• il prossimo numero della lista l è 4; quindi si prende la quarta città nella corrente lista c, la si elimina dalla lista e la si inserisce come prossima città del tour. Il tour parziale è (3 5 ...).
• il numero successivo nella lista l è 1, quindi si prende la prima città della corrente lista c, cioè 1, e la si toglie da c. Il tour parziale è (3 5 1 ...).

Questa procedura continua fino a quando la lista c non è vuota.
Il vantaggio principale di questa rappresentazione è che può essere applicato

l'incrocio classico senza preoccuparsi della validità dei nuovi cammini generati. Per esempio, dati p_1 =(2,5,6,6|1,2,1,2,1) e p_2 =(1,3,2,5|1,1,2,1,1), corrispondenti ai percorsi (2 6 8 9 1 4 3 7 5) e (1 4 3 8 2 5 7 6 9), se incrociamo i due vettori nel punto denotato dal simbolo '|', si ottengono i discendenti s_1= (2,5,6,6|1,1,2,1,1) e s_2= (1,3,2,5|1,2,1,2,1), che corrispondono ai tour (2 6 8 9 1 3 5 4 7) e (1 4 3 8 2 6 5 9 7).
Osserviamo che con questa rappresentazione il crossover non modifica la parte a sinistra del punto d'incrocio, mentre la parte a destra viene sostanzialmente mutata. Il problema principale è che se si creano dei buoni blocchi costitutivi, non è detto che vengano trasmessi alle generazioni successive, anzi se tali tratti di stringa si formano nella parte di percorso a destra del punto di incrocio, certamente verranno distrutti. I risultati sperimentali[2], in effetti hanno evidenziato questo difetto.

Rappresentazione sentiero

Un percorso è codificato in **rappresentazione sentiero** se la lista rappresenta la successione delle città visitate. Questa è da molti ritenuta la più naturale tra le codifiche di un percorso. Così il vettore (5,6,4,8,2,9,1,7,3) rappresenta il percorso (5 6 4 8 2 9 1 7 3).
Sono stati definiti molti operatori di incrocio che supportano questa codifica: poiché questo è il metodo che utilizzeremo nel nostro codice, li descriveremo in maggior dettaglio in seguito.

6.1.2 Rappresentazione con matrici

Le rappresentazioni tramite vettori sono piuttosto semplici e 'naturali'. Ci si potrebbe però chiedere se non esistano altri metodi per codificare il TSP che possano suggerire nuovi modi di effettuare l'incrocio o altri operatori di ricombinazione. Agli inizi degli anni '90 ci sono stati tre tentativi indipendenti di costruire un programma evolutivo usando le matrici per rappresentare i cromo-

[2]J.J. Grefenstette, R. Gopal, B. Rosmaita, D. Van Gucht, 'Genetic Algorithm for the TSP', (1985), in J.J. Grefenstette, 'Proceedings of the First International Conference on Genetic Algorithms (pp.160-168)', Lawrence Erlbaum Associates, Hillsdale.

somi, dovuti a Fox e McMahon (1991), Seniw (1991) e Homaifar e Guan (1991).

Primo metodo

Fox e Mahon[3] hanno rappresentato un tour come una matrice binaria M i cui elementi $m_{i,j}$ (riga i e colonna j) sono uguali a '1' se e solo se la città i compare nel tour prima della città j. In questa rappresentazione una matrice $(n \times n)$ rappresenta un tour se ha le seguenti proprietà:

- il numero di '1' è esattamente $\frac{n \cdot (n-1)}{2}$;
- $m_{i,p_0} = 0, \ \forall \ 1 \leq i \leq n$ (con p_0 città di partenza del cammino);
- se $m_{i,j} = 1$ e $m_{j,k} = 1$, allora $m_{i,k} = 1$.

Se il numero di '1' nella matrice è inferiore a $\frac{n \cdot (n-1)}{2}$, ma le altre due richieste risultano soddisfatte, allora le città sono '*parzialmente ordinate*': questo significa che è possibile completare tale matrice in almeno un modo ed ottenere un tour legale.
Per esempio il tour (2 3 5 1 9 4 7 6 8) è rappresentato dalla matrice:

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

Questa rappresentazione incapsula tutte le informazioni relative alla sequenza, includendo sia la microtopologia delle singole connessioni città a città, che la macrotopologia dei predecessori e dei successori.
Su questa rappresentazione sono stati usati due operatori binari: *intersezione* e *unione*. Essi sono in grado di combinare le caratteristiche di entrambi i genitori (come l'operatore di incrocio), rispettando le tre condizioni richieste.

[3] B.R. Fox, M.B. McMahon, 'Genetic Operators for Sequencing Problems', (1991), in G. Rawlins 'Foundations of Genetic Algorithms' (pp. 284-300).

L'*intersezione* è basata sull'osservazione che l'intersezione dei *bits* di entrambe le matrici può crearne una nuova in cui:

- il numero di '1' non è superiore a $\frac{n\cdot(n-1)}{2}$;
- le ulteriori due richieste descritte in precedenza sono entrambe soddisfatte.

Eventualmente potremo completare la matrice ottenuta in modo da avere un tour legale.
Per esempio p_1 =(1 2 3 4 5 6 7 8 9) e p_2 =(4 1 2 8 7 6 9 3 5) sono rappresentati rispettivamente dalle due matrici:

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

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

L'intersezione di queste due matrici viene effettuata confrontando le posizioni corrispondenti secondo le regole dell'AND logico:

- 0 AND 0 = 0;
- 0 AND 1 = 0;
- 1 AND 0 = 0
- 1 AND 1 = 1

Il risultato è la seguente matrice:

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

Questa matrice rappresenta un tour parziale. Nel passaggio successivo l'operatore di intersezione selezionerà uno dei genitori e aggiungerà alcuni '1' nel discendente sistemando righe e colonne. Un possibile risultato dopo aver completato il secondo passaggio dell'operatore di intersezione è la seguente matrice che rappresenta il tour (1 2 4 8 7 6 3 5 9):

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

L'*unione* è basata sull'osservazione che un sottoinsieme di bits di una matrice può essere combinato con un sottoinsieme di bits dell'altra matrice, a patto che questi due sottoinsiemi abbiano intersezione vuota, creando una nuova matrice soddisfacente le proprietà richieste.
L'operatore suddivide l'insieme delle città in due gruppi disgiunti. Per il primo gruppo di città, copia i bits dalla prima matrice e per il secondo gruppo di città copia i bits dalla seconda matrice. Quindi completa la matrice in modo che rappresenti una sequenza di città facendo un'analisi delle righe e colonne (come per l'intersezione).

Per esempio, se si considerano i due precedenti genitori p_1 e p_2, con partizione delle città nei due insiemi {1,2,3,4} e {5,6,7,8,9}, si ottiene la seguente matrice che dovrà essere completata in una seconda fase:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	0	1	1	1	-	-	-	-	-
2	0	0	1	1	-	-	-	-	-
3	0	0	0	1	-	-	-	-	-
4	0	0	0	0	-	-	-	-	-
5	-	-	-	-	0	0	0	0	0
6	-	-	-	-	1	0	0	0	1
7	-	-	-	-	1	1	0	0	1
8	-	-	-	-	1	1	1	1	1
9	-	-	-	-	1	0	0	0	0

Risultati sperimentali su differenti topologie di città rivelano che gli operatori di intersezione e unione permettono di avere dei progressi anche quando non viene usata l'opzione di salvaguardare l'individuo migliore.

Secondo metodo

Il secondo metodo di rappresentazione tramite matrici è stato proposto da Seniw[4]: l'elemento $m_{i,j}$ della matrice contiene un '1' se e solo se il tour va dalla città i direttamente alla città j.
Questo significa che per ogni riga e ogni colonna nella matrice c'è una sola posizione diversa da zero. Tuttavia non tutte le matrici con questa caratteristica rappresentano un tour completo, perché potrebbero presentarsi uno o più cicli che si richiudono su se stessi senza dare la possibilità di completare il tour.
Una caratteristica particolare di questo approccio è che i cicli non vengono corretti immediatamente, anzi vengono permessi, almeno in una prima fase, nella speranza che possano evidenziarsi dei sottogruppi efficienti in modo naturale. Solo al termine del programma evolutivo i cromosomi sono ridotti ad un singolo tour attraverso successive combinazioni di coppie di 'sottotour' usando un algoritmo deterministico.
Non sono ammessi i sottotour formati da una sola città, anzi, viene fissato un limite inferiore '$q = 3$' città perché bisogna favorire la formazione di buoni gruppi, ma bisogna evitare la totale frammentazione.

[4]D. Seniw, 'A Genetic Algorithm for the Traveling Salesman Problem', (1991), MSc Thesis, University of North Carolina AT Charlotte.

Sono stati definiti i due operatori genetici classici: mutazione e incrocio.
La mutazione seleziona casualmente alcune posizioni nella matrice e scambia il valore dei bit contenuti.
L'incrocio inizia a costruire un discendente ponendo tutti i bit a zero, quindi esamina i due genitori casella per casella e quando scopre un bit identico nella stessa riga e colonna, lo ricopia nel discendente (prima fase). In seguito (seconda fase) si copiano gli '1' alternativamente dai due genitori, facendo attenzione a non violare le regole. Infine se compaiono righe nulle, dovranno essere completate in modo adeguato (fase finale).
Poiché tradizionalmente il crossover produce due discendenti, l'operatore verrà eseguito una seconda volta con i cromosomi genitori scambiati.
Per esempio, dati i genitori:

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

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

La prima e la seconda fase nella creazione di un discendente producono:

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

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

Dopo la fase finale il primo discendente è σ_1 =(1 5 6 2 3 4 9 7 8) ovvero:

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

Questo metodo diede un ragionevole risultato su diversi test dalle 30 alle 512 città. Purtroppo non è risultata chiara l'influenza del parametro 'q', cioè del minimo numero di città che possono comporre un ciclo, rispetto alla qualità del risultato finale. Un'altra perplessità è che l'algoritmo che ricompone i diversi 'sottotour' è tutt'altro che ovvio e può non essere facile applicarlo a problemi molto grandi.

Terzo metodo

Come nel caso precedente, la rappresentazione proposta da Homaifar e Guan[5] impone che l'elemento $m_{i,j}$ della matrice contiene un '1' se e solo se il tour va dalla città i alla città j. Tuttavia gli operatori genetici usati sono diversi.
L'operatore di incrocio può essere a punto singolo o a due punti: esso scambia tutte le posizioni nelle matrici dei due genitori dopo il punto, o i punti, selezionati; ovviamente sarà necessario un algoritmo riparatore per assicurare che in ogni riga e colonna ci sia esattamente un '1' e per impedire la formazione di cicli annidati nel tour.
Un esempio dell'incrocio a due punti sui tour (1 2 4 3 8 6 5 7 9) e (1 4 3 6 5 7 2 8 9), permette di evidenziare i vari passaggi.
Vengono selezionati casualmente due tagli: tra la seconda e la terza colonna il primo e tra la sesta e la settima colonna il secondo.

[5] A. Homaifar, S. Guan, 'A New Approach on the Traveling Salesman Problem by Genetic Algorithm', (1991), Technical Report, North Carolina & T State University.

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

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

Quindi gli elementi compresi tra i punti di taglio vengono scambiati:

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

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

Entrambi i discendenti sono illegali, ma il numero totale di '1' in ogni matrice è nove, quindi è corretto.

Il primo passo dell'algoritmo riparatore sposterà qualche uno all'interno della matrice in modo che tutte le righe e tutte le colonne abbiano precisamente un solo uno. Per esempio, nel discendente a sinistra l'algoritmo può spostare un '1' da $m_{1,4}$ in $m_{8,4}$ e da $m_{3,8}$ in $m_{2,8}$; nel discendete di destra può spostare gli '1' da $m_{2,4}$ in $m_{3,4}$ e da $m_{8,6}$ in $m_{1,6}$. Dopo questa prima fase dell'algoritmo riparatore, il primo discendente rappresenta un tour legale, mentre il secondo è composto da due sottotour: (1 6 5 7 2 8 9) e (3 4).

Il secondo passaggio dell'algoritmo riparatore dovrebbe quindi essere applicato soltanto al secondo discendente: l'algoritmo taglia e collega i due cicli in modo da produrre un tour legale, ma cerca di tenere conto dell'esistenza di collegamenti già esistenti nei genitori. Per esempio potrebbe scegliere di collegare la città '2' con la città '4', poiché il collegamento (2 4) era già presente in uno

dei genitori. In tal caso il discendente sarebbe (1 6 5 7 2 4 3 8 9).
I risultati indicano che il programma evolutivo con l'incrocio a due punti e l'operatore di inversione (un particolare operatore che modifica la 'direzione' di sottopercorsi e ne valuta la bontà) dà dei risultati ottimi su un TSP con 30-100 città.

6.2 Scelta delle prove

Per il TSP risulta piuttosto naturale definire una funzione fitness strettamente legata alla lunghezza del percorso trovato: poiché si vuole minimizzare quest'ultimo valore, la funzione di valutazione è spesso definita come l'inverso della lunghezza del percorso o una funzione ad esso collegata.
Un esempio molto utilizzato (che noi implementeremo nel nostro codice) è $\frac{\sum_{i=0}^{n} c_i}{c_i}$, dove c_i è il costo della soluzione *i-esima* (ed abbiamo n soluzioni ad ogni generazione).

Scelta la funzione di valutazione, vi sono innumerevoli ulteriori opzioni che possono essere utilizzate per migliorare l'algoritmo. Una di queste è la scelta del numero di individui che sostituiremo ad ogni generazione: si può decidere per una strategia 'elitista' (cioé sostituire solo una parte della popolazione ad ogni giro) oppure si può stabilire di sostituire tutti i percorsi. Una scelta molto diffusa è quella di conservare solo l'elemento migliore e sostituire tutti gli altri cammini.
Altri operatori piuttosto interessanti e particolari sono:

• il 'packing', che si occupa di lasciare invariato un unico cammino fra tutti quelli aventi la stessa lunghezza e trasforma casualmente (o secondo criteri stabiliti) tutti gli altri;
• il 'Judgment Day' (l'operatore 'giorno del giudizio'): se attraverso una particolare valutazione il programma ritiene di essere fermo in un massimo locale, per evitare la 'stagnazione' salva il tour migliore e trasforma tutti gli altri.

6.3 L'operatore di crossover

Gli operatori di crossover per il TSP sono strettamente legati alla codifica scelta per il problema. Abbiamo visto nei capitoli precedenti i più diffusi, tralasciando volutamente quelli relativi alla 'rappresentazione sentiero': tratteremo ora con maggior respiro i principali crossover legati a questa codifica vettoriale.

Sono stati definiti molti operatori di incrocio che supportano questa codifica. Essi vengono chiamati brevemente *PMX*, *OX*, *CX*, *OBX* e *ERX*.

6.3.1 PMX (Partially-Mapped Crossover)

Questo operatore, proposto da Goldberg e Lingle[6], costruisce un discendente scegliendo due punti di taglio casualmente.
Per esempio supponiamo di avere due genitori, p_1 =(1 2 3|4 5 6 7|8 9) e p_2 =(4 5 2|1 8 7 6|9 3), che vengono tagliati nei punti indicati dal simbolo '|'.
La parte compresa tra i tagli viene scambiata: σ_1 =(_ _ _|1 8 7 6|_ _) e σ_2 =(_ _ _|4 5 6 7|_ _). Questo passaggio definisce delle corrispondenze tra le città scambiate: '1 $\leftrightarrow$ 4'; '8 $\leftrightarrow$ 5'; '7 $\leftrightarrow$ 6'; '6 $\leftrightarrow$ 7'.
Ora è possibile inserire nella prima e terza parte dei discendenti (quelle ancora vuote) le città che non creano conflitto con quelle già inserite. Avremo quindi: σ_1 =(_ 2 3|1 8 7 6|_ 9) e σ_2 =(_ _ 2|4 5 6 7|9 3).
Le città non ancora inserite vengono piazzate tramite gli abbinamenti sopra definiti: nella prima posizione di p_1 c'è la città '1', ma essendo già presente nel primo discendente σ_1, metterò '4', visto che avevo la relazione '1 $\leftrightarrow$ 4'. Così per tutti gli altri.
I due discendenti saranno: σ_1 =(4 2 3 1 8 7 6 5 9) e σ_2 =(1 8 2 4 5 6 7 9 3).

[6]D.E. Goldberg, R. Lingle, 'Alleles, loci and the TSP', (1985), in J.J. Grefenstette 'Proceedings of the First International Conference on Genetic Algorithms' (pp.154-159), Lawrence Erlbaum Associates, Hillsdale.

6.3.2 OX (Order Crossover)

Anche questo operatore, proposto da Davis[7] inizia il suo cammino dalla scelta casuale di due punti di taglio.
Per esempio, supponendo di avere gli stessi genitori dell'esempio precedente, sceglieremo: p_1 =(1 2 3|4 5 6 7|8 9) e p_2 =(4 5 2|1 8 7 6|9 3). Quindi si copia il segmento compreso tra i punti di taglio nei discendenti: σ_1 =(_ _ _|4 5 6 7|_ _) e σ_2 =(_ _ _|1 8 7 6|_ _).
Partendo dal secondo taglio di p_2, si considerano tutte le città del tour, si cancellano quelle già presenti in σ_1 e i numeri rimasti vengono copiati nello stesso ordine, dopo la seconda linea di taglio, in σ_1. Raggiunta la fine della stringa continuiamo dalla prima posizione.
La sequenza di città del secondo genitore, a partire dal secondo taglio è (9 3 4 5 2 1 8 7 6); da queste si tolgono le città già presenti nel primo discendente (4 5 6 7) e rimangono (9 3 2 1 8). Questa sequenza è rimpiazzata nel primo discendente a partire dal secondo taglio: σ_1 =(2 1 8 4 5 6 7 9 3). Analogamente si otterrà il secondo discendente: σ_2 =(3 4 5 1 8 7 6 9 2).

6.3.3 CX (Cycle Crossover)

Questo operatore, proposto da Oliver[8], costruisce discendenti in modo che ogni città con la relativa posizione provenga da un genitore.
Spieghiamo il meccanismo di questo operatore con un esempio; data la coppia di genitori p_1 =(1 2 3 4 5 6 7 8 9) e p_2 =(4 1 2 8 7 6 9 3 5) si costruisce il primo discendente: si prende la prima città dal primo genitore ('1') e poi si guarda la città corrispondente nel secondo genitore (in questo caso è '4').
Entrambe le città verranno inserite nel discendente nella stessa posizione che occupano in p_1: avremo quindi σ_1 =(1 _ _ 4 _ _ _ _ _); analogamente in corrispondenza della città '4' di p_1 c'è la città '8' in p_2, che verrà inserita in σ_1

[7] L. Davis, 'Applying Adaptive Algorithms to Epistatic Domains', (1985), in 'Proceedings of the International Joint Conference on Artificial Intelligence', (pp.162-164), Lawrence Erlbaum Associates, Hillsdale.

[8] I.M. Oliver, D.J. Smith, J.R.C. Holland, 'A Study of Permutation Crossover Operators on the Traveling Salesman Problem', (1987), in J.J. Grefenstette 'Proceedings of the Second International Conference on Genetic Algorithms', (pp.224-230), Lawrence Erlbaum Associates, Hillsdale, NJ.

nella stessa posizione occupata in p_1: σ_1 =(1 _ _ 4 _ _ _ 8 _).
Continuando in questo modo, le prossime città da inserire sono '3' e '2', ma la città '2' riconduce alla città '1' che è stata già inserita, pertanto completa un ciclo. Il discendente fino a questo punto è σ_1 =(1 2 3 4 _ _ _ 8 _); per completarlo si copiano le posizioni ancora vuote dall'altro genitore: σ_1 =(1 2 3 4 7 6 9 8 5). In modo analogo si costruisce il secondo discendente scambiando il ruolo dei genitori: σ_2 =(4 1 2 8 5 6 7 3 9).

Dai risultati sperimentali, effettuati confrontando questi primi tre operatori[9], l'operatore che ha dato gli esiti migliori è l'OX, che è risultato superiore dell'11% rispetto al PMX e del 15% rispetto al CX.

6.3.4 OBX (Order-Based Crossover)

L'operatore 'OBX' effettua il crossover in questa maniera: prende due genitori, sceglie alcune posizioni a caso, elimina dal primo genitore gli elementi del secondo genitore che si trovano nelle posizioni date e infine rimpiazza i posti vuoti con gli stessi elementi tolti, ma seguendo l'ordine che essi hanno nel primo genitore. Per creare un secondo discendente si ripete poi il tutto invertendo i due genitori.
Prendiamo per esempio i due genitori p_1 =(2 6 9 8 5 4 3 1 7) e p_2 =(1 5 9 3 7 2 8 4 6). Scegliamo quindi le posizioni '1', '3', '7', '9' da p_2, che corrispondono alle città '1', '9', '8', '6', ed eliminiamo queste ultime da p_1; avremo σ_1 =(2 _ _ _ 5 4 3 _ 7).
Completiamo σ_1 con gli stessi valori tolti ma seguendo l'ordine di p_2: σ_1 =(2 1 9 8 5 4 3 6 7). Effettuiamo lo stesso iter per σ_2, scegliendo ad esempio le posizioni '2', '3', '5', '8', avremo σ_2 =(6 9 5 3 7 2 8 4 1).

[9] I.M. Oliver, D.J. Smith, J.R.C. Holland, 'A Study of Permutation Crossover Operators on the Traveling Salesman Problem', (1987), in J.J. Grefenstette, 'Proceedings of the Second International Conference on Genetic Algorithms', (pp.224-230), Lawrence Erlbaum Associates, Hillsdale, NJ.

6.3.5 ERX (Edge Recombination Crossover)

Un problema emerso osservando gli operatori d'incrocio discussi sopra è che molti di questi tengono memoria della posizione e dell'ordine delle città, piuttosto che dei collegamenti fra le città stesse, dato quest'ultimo molto più significativo nella costruzione di un tour, in quanto per formare dei buoni blocchi costitutivi sono necessarie informazioni sui collegamenti fra le città o piccoli gruppi di città. Quindi un buon operatore dovrebbe cercare di esplorare al meglio i possibili collegamenti fra le città nei genitori, piuttosto che tenere memoria della posizione e dell'ordine delle città.
Un gruppo di ricercatori[10] ha proposto questo operatore che riesce a trasferire oltre il 95% dei bordi presenti nei genitori, dove per bordi si intende la vicinanza tra due città nel cromosoma.
Per esempio per il tour (7 5 3 1 4 8 6 9 2) si hanno i seguenti bordi: (7 5), (5 3), (3 1), (1 4), (4 8), (8 6), (6 9), (9 2), (2 7). Notiamo che questo operatore è stato proposto per problemi di TSP 'simmetrici', nei quali cioé la distanza tra due città è la stessa sia che si vada da una all'altra che viceversa. Quindi, per quel che riguarda i bordi, $(ab) = (ba)$, poiché interessa la relazione tra le città e non il verso in cui si compie il percorso.
Un discendente verrà costruito esclusivamente con le coppie presenti in entrambi i genitori, mediante l'ausilio di una lista contenente i collegamenti diretti di ogni città con le altre, in entrambi i tour dei genitori. Ovviamente per ogni città ci saranno almeno due e al più quattro città nella lista.
Per esempio, se i genitori sono p_1 =(7 5 3 1 4 8 6 9 2) e p_2 =(4 5 3 1 7 8 9 2 6), la lista dei collegamenti è:

Città	1	2	3	4	5	6	7	8	9
Bordi	3,4,7	9,7,6	5,1	1,8,6,5	7,3,4	8,9,2,4	2,5,1,8	4,6,7,9	6,2,8

Ora si selezione una città di partenza da uno dei due genitori, per esempio la '7' in p_1, e si legge nella tabella con chi confina (cioè {2,5,1,8}). La scelta fra le città a disposizione ricade su quella con il minor numero di collegamenti:

[10] L.D. Whitley, T. Starkweather, D'A. Fuquay, 'Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator', (1989), in J. Schaffer, 'Proceedings of the Third International Conference on Genetic Algorithms', (pp.133-140), Morgan Kaufmann Pubblishers.

nel caso considerato la città '8' confina con quattro città, mentre le altre hanno solo tre collegamenti. In tal caso la scelta ricade casualmente su una delle città con il minor numero di collegamenti.
Supponiamo che sia scelta la città '1': σ_1 =(7 1 _ _ _ _ _ _ _).
Si va quindi a controllare nella lista con quali città confina la città '1' (con {3,4,7}) e tra queste si sceglierà la città '3', perché ha solo due collegamenti.
In conclusione il discendente potrebbe essere σ_1 =(7 1 3 5 4 8 9 2 6).
Tale metodo di selezione incrementa la possibilità di completare un tour con tutti gli spigoli selezionati dai genitori anziché scelti a caso.
È possibile apportare una piccola modifica a questo operatore: se una città confina con un'altra in entrambi i genitori, allora quest'ultima viene evidenziata sulla lista ed ha la precedenza sulle altre città confinanti. In una serie di esperimenti si è visto che la procedura fallisce molto raramente (1%-1.5%).
L'operatore di incrocio ER è stato testato su tre TSP di 30, 50 e 75 città e in tutti i casi ha riportato risultati migliori rispetto agli altri operatori.

6.4 L'operatore di mutazione

Nel TSP esistono diversi operatori caratteristici che possono essere considerati come operatori di mutazione. Uno di questi è l'*operatore di inversione*: nella sua versione più diffusa (detta 'inversione semplice') seleziona due punti nel cromosoma e la porzione di stringa compresa tra questi due punti viene invertita.
Per esempio (1 2|3 4 5 6|7 8 9) diventa (1 2|6 5 4 3|7 8 9).
Alcuni approfondimenti teorici[11] indicano che questo operatore può risultare utile per ricercare buoni ordini nelle stringhe. E' stato provato[12] che in un TSP di 50 città, un algoritmo genetico con l'inversione migliora il sistema col solo incrocio, ma un incremento nel numero dei punti di taglio diminuisce l'efficacia

[11] J.H. Holland, 'Adaptation in Natural and Artificial Systems', (1975), University of Michigan Press, Ann Arbor.
[12] L.D. Whitley, T. Starkweather, D'A. Fuquay, 'Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator', (1989) in J. Schaffer, 'Proceedings of the Third International Conference on Genetic Algorithms', (pp.133-140), Morgan Kaufmann Pubblishers.

dell'operatore. Tuttavia l'inversione, come la mutazione, è un operatore 'unario' che può solo accompagnare l'operatore di incrocio, che è l'unico in grado di ricombinare le informazioni.

Per quel che riguarda la classica mutazione presentiamo ora due diversi operatori (riguardanti la rappresentazione sentiero) che possono essere usati anche nello stesso codice: la 'micromutazione' e la 'macromutazione'.

Per '*micromutazione*' si intende lo spostamento casuale, con una probabilità piuttosto bassa, di una città in un'altra posizione (sempre scelta casualmente) nel cammino. Per esempio, partendo dal percorso p =(1 2 3 4), potremmo (tramite generazione di numeri casuali) estrarre la città '3' ed inserirla al primo posto del nostro vettore; avremo quindi p =(3 1 2 4).
Questo operatore permette quindi di inserire nella ricerca nuovi 'punti' dello spazio di ricerca in maniera casuale.

L'operatore di '*macromutazione*' si occupa invece di spostare blocchi di città da un punto ad un altro del cammino. Per esempio, partendo dal percorso p =(1 2 3 4 5 6), si opera in questo modo: si decide se effettuare la mutazione (tramite generazione di numeri casuali confrontati con probabilità assegnate piuttosto basse); se si opera, si estrae casualmente una città dalla quale partire (nel nostro esempio supponiamo di estrarre la città '5'); si estrae la lunghezza del tratto da mutare (supponiamo di estrarre '3', quindi avremo estratto p_e =(5 6 1) e ci rimarrà p_r =(2 3 4)); si estrae il punto in cui inserire il codice estratto nel tratto rimasto (per esempio '2'); infine si combinano i due sottocammini (inserendo p_e in posizione '2' nel cammino p_r, avremo p_m =(2|5 6 1|3 4)).
Questo operatore si occupa di esplorare nuovi punti dello spazio di ricerca, tenendo conto però di eventuali sottocammini interessanti.

Notiamo che gli operatori di mutazione presentati non sono in contrapposizione uno con l'altro: potremo quindi trovarli in un codice senza che uno escluda gli altri.

6.5 Spiegazione del codice R

Il codice sviluppato per questo algoritmo è molto meno 'competitivo' rispetto al codice precedente del Master Mind: lo scopo è unicamente quello di descivere un'applicazione molto interessante per gli algoritmi genetici, implementando in R la teoria che abbiamo fin qui visto, senza però la pretesa di creare un codice innovativo o avanzato. La principale motivazione è che i termini di paragone per questo problema sono eccessivamente avanzati (e costosi): basti pensare che negli ultimi anni, per risolvere problemi di TSP, sono state utilizzate reti formate da centinaia di processori o software di calcolo distribuito (nei quali vengono utilizzati un gran numero di computer che svolgono calcoli ed interagiscono scambiandosi informazioni e risultati).

Il codice si basa su di una codifica vettoriale di tipo 'sentiero', su di una funzione di valutazione del tipo $\frac{\sum_{i=0}^{n} c_i}{c_i}$ (dove c_i è il costo della soluzione $i\text{-}esima$ ed abbiamo n soluzioni ad ogni generazione), su di un crossover di tipo 'order' (OX) ed utilizza sia la 'micromutazione' che la 'macromutazione' definite qui sopra. Si è dunque puntato ad una codifica piuttosto semplice e veloce, ma che dia risultati soddisfacenti.

6.5.1 Valori di input e di output

I possibili parametri di input impostabili dall'utente sono:

- 'v_costi', che deve contenere la matrice dei costi dei collegamenti sotto forma di vettore (di default $v\_costi = NULL$, ma è necessario un valore corretto: questa impostazione serve unicamente per creare un messaggio di errore e terminare il codice);

- 'n_citta', che serve ad indicare al programma il numero di città a cui si riferisce la matrice v_costi (di default $n\_citta = sqrt(length(v\_costi))$);

- 'n_formiche', che serve ad indicare il numero di individui (possibili soluzioni) ad ogni generazione (di default $n\_formiche = n\_citta$);

- 'n_gen', che serve ad indicare il numero di generazioni che il programma deve fare (di default $n\_gen = (10 * (n\_citta)^2)/n\_formiche$). Questo valore non è strettamente vincolante poiché terminate le generazioni impostate, il programma chiederà all'utente se si è soddisfatti del risultato (e quindi se si vuole terminare l'esecuzione) o se si vuole continuare a creare nuove generazioni;
- 'seme', che serve ad impostare il seme di generazione casuale (di default $seme = NULL$).

Il programma fornirà inoltre in output una matrice contenente la popolazione finale ('pop') ed una matrice contenente l'evoluzione della soluzione minima ('evol[,1]') e della media delle soluzioni ('evol[,2]') nel corso delle generazioni. Appena prima di terminare l'esecuzione, inoltre, il programma si occupa di disegnare un grafico dell'evoluzione del valore minimo e medio delle soluzioni.

6.5.2 Aspetti particolari

Nel nostro programma abbiamo deciso di seguire una strategia elitista, ma relativa solo al migliore individuo: ad ogni generazione si effettua un ricambio totale dei cromosomi, con l'unica eccezione del cammino più breve (un solo individuo). Questo porta ad una maggiore velocità dell'algoritmo, ma forse provoca una perdita di informazione importante nel ricambio generazionale.

Per quel che riguarda la mutazione abbiamo deciso di includere sia la 'micromutazione' che la 'macromutazione' per la ricerca di nuovi punti nello spazio delle soluzioni. La probabilità della mutazione è calcolata tenendo conto della diffusione della soluzione migliore attuale nella popolazione e del numero di generazioni passate dall'ultimo miglioramento: il programma tiene in memoria una variabile ('mutaz') calcolata in questo modo:

$$mutaz \ = \ ((fetta/1000) \ + \ 0.001) \ + \ (cambiamenti * 0.00001)$$

dove 'fetta' è la percentuale di popolazione uguale alla migliore soluzione e 'cambiamenti' è il numero di generazioni passate dall'ultimo miglioramento.

Questo valore è però limitato superiormente a 0.1, onde evitare eccessiva casualità.
Abbiamo inoltre deciso di rendere mutualmente esclusive le due mutazioni: se effettuo la macromutazione su di un cammino, per non distruggere la nuova sequenza creata, non effettuerò la micromutazione.

6.6 Un esempio pratico

Per condurre test sugli algoritmi TSP è stata sviluppata la libreria TSPLIB[13], che contiene istanze d'esempio del problema TSP e relative varianti. Molti di questi esempi sono liste di città e schemi di circuiti stampati.

Provando il nostro programma con questi dati abbiamo notato come ottenga dei risultati buoni per quel che riguarda problemi con numero di città non elevato, mentre faccia fatica a trovare buone soluzioni di problemi più complicati.

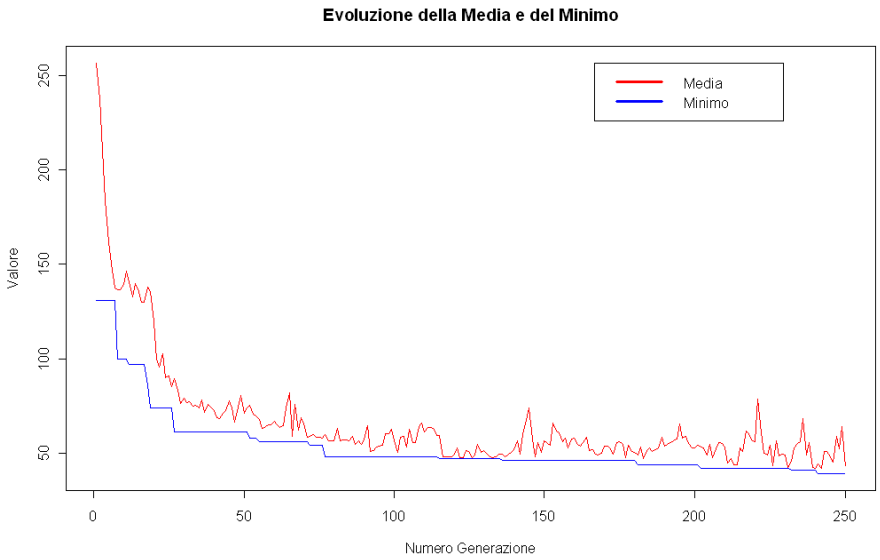

Figura 6.1: *Evoluzione del minimo e del valor medio nel problema con 17 città.*

[13]Sul web all'indirizzo 'http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/'.

In particolare presentiamo alcune prove effettuate con i dati presenti nei files 'br17' e 'ftv55', rispettivamente problemi con 17 e 55 città (di tipo asimmetrico).

Nel problema con 17 città, nella quasi totalità delle prove effettute, il programma trova una soluzione ottima (pari a 39) o molto vicina ad essa (pari a 40) in meno di 250 generazioni, effettuate con 17 individui ognuna[14].

Notiamo come il numero di combinazioni differenti possibili sia di quasi 21 mila miliardi (16! = 20922789888000), mentre il nostro codice ne valuta appena 4250 (17∗250) per trovare una soluzione ottima o molto prossima all'ottimo.

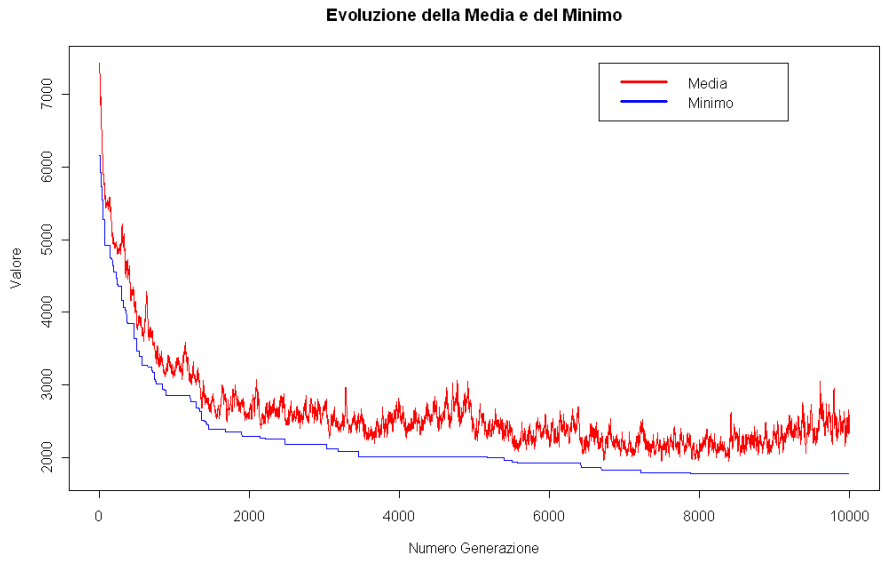

Figura 6.2: *Evoluzione del minimo e del valor medio nel problema con 55 città.*

Per quanto riguarda il problema con 55 città, invece, si evidenzia un buon andamento iniziale della 'discesa', dovuto principalmente a generazioni iniziali con valutazione piuttosto bassa (ovvero con lunghezza del cammino molto grande), ma quando la popolazione arriva ad un minimo locale difficilmente

[14]Il grafico nella figura (6.1) è stato ottenuto effettuando 250 generazioni con seme di generazione casuale '1' e 'n_formiche=17'.

riesce ad uscirne (nel grafico (6.2) questo problema è evidenziato da lunghi segmenti orizzontali che rappresentano la mancanza di un miglioramento per centinaia di generazioni). Il risultato finale è comunque discreto: considerando 55 individui ad ogni generazione, in 3000 generazioni si ottengono percorsi di lunghezza pari a 2100-2200 (contro una soluzione ottima di 1608); in 10000 generazioni si colma buona parte della distanza mancante, arrivando a percorsi pari a 1750-1900[15]. Non sembrano risultati eccellenti, ma bisogna anche considerare che in 10000 generazioni, con 55 individui per generazione, arriviamo a testare 550000 possibili soluzioni, contro le $2.31 * 10^{71}$ (cioé 2 seguito da 71 zeri) possibili combinazioni differenti.

Possibili miglioramenti dell'algoritmo potrebbero quindi essere fatti sia riguardo la popolazione iniziale (attraverso la generazione di sequenze secondo tecniche specifiche del TSP o l'immissione da parte dell'utente) sia nella generazione dei discendenti e il superamento di minimi locali.

[15]Il grafico in figura (6.2) è stato ottenuto facendo 10000 generazioni con seme di generazione casuale '0' e 'n_formiche=55'.

Appendice A

Codice R del Master Mind

In questa appendice presentiamo il codice R completo e commentato del gioco del Master Mind sviluppato secondo la teoria degli algoritmi genetici.

A.1 Codice R commentato

```
################################
# funzione confrseq(vett,matr) #
################################
# ritorno 1 se sequenza "cattiva" vett è gia' stata registrata
#  in matr, 0 altrimenti

confrseq<-function(vett,matr)
{
righe<-length(matr)/length(vett)
colonne<-length(vett)

for(ii in 1:righe)   # righe
   {for(jj in 1:colonne)   # colonne
      {if(matr[(ii-1)*colonne+jj]!=vett[jj])
         {break
         }
       if(jj==colonne)   # ho trovato la sequenza
```

```
         {return (1)
         }
      }
   }

return (0)
}



###################################
# funzione accseq(vett,poss,mm,i) #
###################################
# controllo che vett sia una sequenza accettabile,
#  se completata con i colori "possibili", rispetto
#  alle prove gia' presenti nella matrice mm
# ritorno 1 se sequenza accettabile, 0 se non accettabile

accseq<-function(vett,poss,mm,i)
{
v2<-vett
puntatori<-rep(0,length(vett))
for(j in 1:length(v2))
   {if(vett[j]==0)
      {puntatori[j]=1
       v2[j]<-poss[puntatori[j]]
      }
   }
# genero una sequenza completa "a caso" da vett
#   --> se esiste --> sequenza accettabile
while(TRUE)
   {uscita<-0
    for(j in 1:i)     # prove
       {
```

```
# confronto v2 con mm[j,1:length(v2)] con valutazione
#   mm[j,length(v2)+1] e mm[j,length(v2)+2]
#  se trovo che non e' accettabile --> uscita<-1 + break
#  altrimenti continuo semplicemente

       # controllo quanti ce ne sono di uguali nella stessa posizione
       #  (ug - neri) e in posizione diversa (ug2 - bianchi)
        ug<-0
        ug2<-0
        v3<-mm[j,]
        v4<-v2

       # prima i neri
        for(jj in 1:length(v2))
           {if(v2[jj]==v3[jj])
               {ug<-ug+1
                v2[jj]<-0
                v3[jj]<-0
               }
           }

       # poi i bianchi
        for(jj in 1:length(v2))   # v2[jj]
           {if(v2[jj]==0)
               {next      # incremento jj
               }

            for(oo in 1:length(v2))   # v3[oo]
               {if(jj==oo)
                   next    # incremento oo
                if(v3[oo]==0)
                   {next   # incremento oo
                   }
                if(v2[jj]==v3[oo])
```

```
                    {ug2<-ug2+1
                     v2[jj]<-0
                     v3[oo]<-0
                     break   # incremento jj
                    }
                }
            }

          v2<-v4

          if(ug!=mm[j,length(v2)+1])
             {uscita<-1
              break
             }
          if(ug2!=mm[j,length(v2)+2])
             {uscita<-1
              break
             }
         }

# ho trovato una sequenza completa accettabile --> ritorno 1
     if(uscita==0)
        return (1)

    # altrimenti --> non ho trovato nessuna sequenza accettabile
    #  --> se posso genero la prossima seq. completa
    #  --> se non ce ne sono piu' --> ritorno 0
     riporto<-1
     for(jj in length(vett):1)
        {if(puntatori[jj]==0)    # posizione fissa
           {if(jj==1)    # ho finito le sequenze
              {return (0)
              }
            next
```

```
          }
        if(riporto==1)
          {puntatori[jj]<-puntatori[jj]+1
           riporto<-0
          }
        if(puntatori[jj]>length(poss))
          {if(jj==1)   # ho finito le sequenze
             {return (0)
             }
           puntatori[jj]<-1
           riporto<-1
          }
        v2[jj]<-poss[puntatori[jj]]
# posso chiudere prima perche' ho gia' la mia sequenza
#  (i successivi non cambiano)
        if(riporto==0)
          {break
          }
       }
   }   # fine while(TRUE)

}



#######################
# funzione mmag8(...) #
#######################
# mmag8(soluz=c(1,7,5,7,5),ppresi=0.32,
#       ptpresi=0.73,kkk=7.5,seme=0,ncol=8)
# mmag8(soluz=c(1,7,5,7,5),ncol=8)

mmag8<-function(soluz=sample(c(1:ncol),5,replace=TRUE),ppresi=0.32,
                ptpresi=0.73,kkk=7.5,seme=NULL,ncol=8)
```

```
{
# eventualmente da cambiare
ncolori<-ncol
ncase<-length(soluz)

# altre strutture
neri<-ncase+1
bianchi<-ncase+2
# matrice delle prove + neri + bianchi
mm<-matrix(data=NA,nrow=100,ncol=ncase+2)
possibili<-c(1:ncolori)
soluzione<-soluz   # soluzione scelta
sol<-NULL
cseq<-NULL   # per le sequenze "cattive"
# inoltre ho ncp --> nuovo codice potenziale
#          e cfg --> current favourite guess

# strutture per il calcolo delle probabilita'
probabilita<-rep(10,ncolori)
cambi<-rep(0,ncolori)
media_pres<-rep(0,ncolori)
media_ass<-rep(0,ncolori)
frequenza<-rep(0,ncolori)
nonfrequenza<-rep(0,ncolori)



# controllo colori\soluz
if(max(soluz)>ncol)
   {cat("Errore nei parametri di output: max(soluz)>ncol.\n")
    return(c(ppresi,ptpresi,kkk,soluz,0,0,seme))
   }
if(min(soluz)<=0)
   {cat("Errore nei parametri di output: min(soluz)<=0.\n")
```

```
    return(c(ppresi,ptpresi,kkk,soluz,0,0,seme))
   }
if(min(ncol)<=0)
   {cat("Errore nei parametri di output: min(ncol)<=0.\n")
    return(c(ppresi,ptpresi,kkk,soluz,0,0,seme))
   }


if(is.null(seme)==FALSE)
   {set.seed(seme)
   }

# Stampo a video la soluzione
cat("\nSecret code: [",soluzione,"]\n--------------------\n")

# PASSO 1: primo giro a caso
 cfg<-sample(possibili,ncase,replace=TRUE,prob=probabilita[possibili])
 mm[1,1:ncase]<-cfg
 ncp<-cfg
 cfg[neri]<-0
 cfg[bianchi]<-0


i<-1   # numero di tentativi
giri<-1   # numero di giri
avuoto<-0   # numero di giri a vuoto
scar<-NULL
gutil<-NULL

while(i<100)    # e' come se fosse un while(TRUE)
   {# valutazione
    sol<-soluzione
    ncp2<-ncp   # per la valutazione
```

```
# prima i neri
 mm[i,neri]<-sum(ncp==sol)
 z<-ncp==sol
 for(g in 1:ncase)
    {if(z[g])
       {sol[g]<-0
        ncp2[g]<-0
       }
    }
# poi i bianchi
 mm[i,bianchi]<-0
 for(k in 1:ncase)
    {for(j in 1:ncase)
        {if(ncp2[k]==0)
            break   # esci dal for(j) e incrementa k
         if(sol[j]==0)
            next    # incrementa j
         if(ncp2[k]==sol[j])
           {sol[j]=0
            mm[i,bianchi]<-mm[i,bianchi]+1
            break   # esci dal for(j) e incrementa k
           }
        }
    }

# se la valutazione della ncp e' migliore o uguale alla cfg
#    --> allora cfg<-ncp
#  n.b.: tengo in ncp il vecchio cfg
#        per il calcolo delle probabilita'
 ncp<-cfg
 if(cfg[neri]+cfg[bianchi]<mm[i,neri]+mm[i,bianchi])
    cfg<-mm[i,1:bianchi]
 else if(cfg[neri]+cfg[bianchi]==mm[i,neri]+mm[i,bianchi])
        {if(cfg[neri]<=mm[i,neri])
```

```
              cfg<-mm[i,1:bianchi]
            }

   # se la valutazione e' [ncase,0] --> ho vinto
    if(mm[i,neri]==ncase)
      {cat("# ",i," : ",mm[i,1:ncase],"[",mm[i,neri],",
           ",mm[i,bianchi],"] (",avuoto," giri a vuoto)
            --> HO TROVATO IL CODICE !!!\n\n")
       return(c(ppresi,ptpresi,kkk,soluz,i,giri,seme))
# le 3 prob. impostate dall'utente + soluzione + seme + numero prove
      }


   # se lo score e' 0 --> tolgo i colori del cfg dai possibili
    if(mm[i,neri]+mm[i,bianchi]==0)
      {for(ii in 1:ncase)
          {possibili<-possibili[which(possibili!=mm[i,ii])]
          }
      }

   # stampo a video il risultato del tentativo
    cat("# ",i," : ",mm[i,1:ncase],"[",mm[i,neri],",
        ",mm[i,bianchi],"] (",avuoto," giri a vuoto)
         --> cfg=[",cfg[1:ncase],"]\n")

    avuoto<-0   # numero di giri a vuoto
    giri<-giri+1

   # ora aggiorno i parametri per il calcolo delle probabilita'
   #  devo aggiornare: - zzfit = fit corrente = neri + bianchi
   # - media_presenza =
   #   (se c'e') (media_presenza*frequenza) + (zzfit/ncase)
   # - frequenza++ (se c'e')
   # - media_ass =
```

```
#   (se non c'e')(media_ass*nonfreq)+((4-zzfit)/length(assenti))
# - nonfreq++ (se non c'e')
# - media_pres = media_pres/freq
# - media_ass = media_ass/nonfreq
# - prob = prob(0)(cioe' 10) + cambi + media_pres + media_ass
#
# - cfg --> x_c+y_c
# - delta_fit = zzfit - cfg
# - NUOVI nella prova --> c = c + (delta_fit/delta_cambi)
# - NUOVI nel cfg --> c = c - (delta_fit/delta_cambi)

# Durante l'estrazione:
# - se freq=0 --> prob = prob*kkk
# - usati nel cfg...
# - appena estratti

 zzfit<-mm[i,neri]+mm[i,bianchi]
 assenti<-c(1:ncolori)

# presenze
 for(j in 1:ncase)
    {media_pres[mm[i,j]] <- (media_pres[mm[i,j]]*frequenza[mm[i,j]]) +
                             (zzfit/ncase)
     frequenza[mm[i,j]] <- frequenza[mm[i,j]] + 1
     media_pres[mm[i,j]] = media_pres[mm[i,j]] / frequenza[mm[i,j]]

     assenti<-assenti[which(assenti!=mm[i,j])]
    }

# assenze
 if(length(assenti)>0)
   {for(j in 1:length(assenti))
       {media_ass[assenti[j]] <-
          (media_ass[assenti[j]]*nonfrequenza[assenti[j]]) +
```

```
            ((ncase-zzfit)/length(assenti))
         nonfrequenza[assenti[j]] <- nonfrequenza[assenti[j]] + 1
         media_ass[assenti[j]] = media_ass[assenti[j]] /
              nonfrequenza[assenti[j]]
        }
   }

# cambi --> il vecchio cfg --> in ncp
 zzcfg<-ncp[neri]+ncp[bianchi]
 deltafit<-zzfit-zzcfg
# ho in scar gli indici dei colori precedentemente scartati
#  --> sono i "nuovi" nel cfg
# ho in gutil gli indici (della nuova prova) fissi (presi dal cfg)
#  --> prendo i "nuovi" nella prova
 if(length(scar)>0)
   {# nuovi nel cfg
    cambi[ncp[scar]] <- cambi[ncp[scar]] - (deltafit/length(scar))
   }
# metto in scar gli indici per i "nuovi" della prova
 if(length(gutil)>0)
   {scar<-c(1:ncase)
    for(j in 1:length(gutil))
       {scar<-scar[which(scar!=gutil[j])]
       }
    # nuovi nella prova
    cambi[mm[i,scar]] <- cambi[mm[i,scar]] + (deltafit/length(scar))
   }
 scar<-NULL
 gutil<-NULL

# calcolo delle probabilita' e correggo le probabilita di freq==0
 for(j in 1:ncolori)
    {probabilita[j] <- 10 + cambi[j] + media_pres[j] + media_ass[j]
     if(frequenza[j]==0)
```

```
          probabilita[j]<-probabilita[j]*kkk
      }



# HO AGGIORNATO LE PROBABILITA' (in probabilita)
# Ora estraggo i neri e i bianchi dal cfg

# inizializzazione del nuovo codice potenziale
 ncp<-rep(0,ncase)
 a<-NULL

# Tolgo gli impossibili dal cfg
#   --> in a metto gli indici dei possibili
 for(j in 1:ncase)
    {if(sum(cfg[j]==possibili)>0)
        a<-c(a,j)
    }

# tengo in memoria le sequenze cattive
 a2<-a
 ncp2<-ncp

 while(TRUE)
    {# neri --> tengo la posizione
     if(cfg[neri]>0)
       {if(length(a)>1)
           b<-sample(a,cfg[neri],prob=probabilita[a])
        else
           b<-a
        ncp[b]<-cfg[b]
        for(j in 1:cfg[neri])
          {a<-a[which(a!=b[j])]
          }
```

```
     gutil<-b
    }

# bianchi --> li shifto
 if(cfg[bianchi]>0)
   {if(length(a)>1)
      {b<-sample(a,cfg[bianchi],prob=probabilita[a])
       d<-which(ncp==0)   # d E' SEMPRE MAGGIORE DI 1
       c<-sample(d,cfg[bianchi])
       while(sum(b==c)!=0)
          c<-sample(d,cfg[bianchi])
      }
    else    # caso in cui ho 1 possibile solo --> length(a)==1
      {b<-a
       d<-which(ncp==0)   # d E' SEMPRE MAGGIORE DI 1
       c<-sample(d,cfg[bianchi])
       while(sum(b==c)!=0)
          c<-sample(d,cfg[bianchi])
      }
    ncp[c]<-cfg[b]
    gutil<-c(gutil,c)
   }
 else
    b<-NULL

# se e' una sequenza buona, continuo.
# Altrimenti ricomincio (a<-a2 e ncp<-ncp2)
 if(is.null(cseq)==FALSE)
   {ll<-confrseq(ncp,cseq)
    if(ll==1)   # sequenza cattiva
      {a<-a2
       ncp<-ncp2
       giri<-giri+1
       avuoto<-avuoto+1
```

```
       next
      }
   }


# mancanti --> mutazione
# in a --> quelli rimanenti dal giro nero e in b
#      --> quelli utilizzati nel giro bianco
# quelli di a non usati da b --> scartati
 if(length(b)>0)
   {for(j in 1:length(b))
      {a<-a[which(a!=b[j])]
      }
   }
 c<-c(1:ncase)
 if(length(a)>0)
   {for(j in 1:length(a))
      {c<-c[which(c!=a[j])]
      }
   }

# ora in a ho quelli da scartare (gli indici di ...)
#  e in c quelli gia' utilizzati (gli indici di ...)
# controllo che sia una sequenza accettabile
 poss<-possibili

 if(length(a)>0)
   {for(k in 1:length(a))
      {poss<-poss[which(poss!=cfg[a[k]])]
      }
   }

 if(length(poss)>0)
   ll<-accseq(ncp,poss,mm,i)
```

```
      else
        if(sum(ncp==0)>0)
           ll=0   # sequenza non accettabile

      if(ll!=1)   # sequenza non accettabile
        {cseq<-c(cseq,ncp)
         a<-a2
         ncp<-ncp2
         giri<-giri+1
         avuoto<-avuoto+1
         next
        }
      else   # sequenza accettabile
         break
     }   # fine del while(TRUE)


# ora in a ho quelli da scartare (gli indici di ...)
#   e in c quelli gia' utilizzati (gli indici di...)
# ncp (nuovo codice potenziale) - mm - possibili (numeri possibili)
# Durante l'estrazione:
#                - se freq=0 --> prob = prob*kkk
#                - usati nel cfg...
#                - appena estratti

# prima tengo in memoria "a" per il calcolo delle probabilita'
#   al giro successivo
 scar<-a

 b<-ncp==0

# ora correggo le probabilita di "appena usati dal cfg"
 probabilita[cfg[c]]<-probabilita[cfg[c]]*ppresi
```

```
if(sum(b)!=0)    # se c'e' n'e' almeno uno da inserire
  {for(j in 1:ncase)
     {if(b[j])   # devo inserire un elemento in j
        {poss<-possibili
         if(length(a)>0)
           {for(k in 1:length(a))
              {poss<-poss[which(poss!=cfg[a[k]])]
              }
           }
         poss2<-NULL
        # guardo se sono possibili
         for(kk in 1:length(poss))
           {ncp[j]<-poss[kk]
            ll<-accseq(ncp,poss,mm,i)
            if(ll==1)   # sequenza accettabile
              {poss2<-c(poss2,poss[kk])
              }
           }

         # ora ho in poss2 --> i numeri buoni da estrarre
         # estraggo e correggo la probabilita'
         #         nel colore appena estratto
          if(length(poss2)>1)
            {ncp[j]<-sample(poss2,1,prob=probabilita[poss2])
            }
          else if(length(poss2)==1)
                 {ncp[j]<-poss2
                 }
               else
                 {cat("Errore !!!\n")
                  cat("Nessun colore possibile da estrarre !!!\n")
                  return (c(ppresi,ptpresi,kkk,soluz,-i,giri,seme))
                 }
          probabilita[ncp[j]]<-probabilita[ncp[j]]*ptpresi
```

```
            }
          }
        }

   # la gioco
    i<-i+1
    mm[i,1:ncase]<-ncp

   }   # fine del while(i<100)
}   # fine della funzione
```

Appendice B

Codice R del TSP

In questa appendice presentiamo il codice R completo e commentato del 'problema del commesso viaggiatore' (TSP) sviluppato secondo la teoria degli algoritmi genetici.

B.1 Codice R commentato

```
tspAG<-function(v_costi=NULL,n_citta=sqrt(length(v_costi)),
                n_formiche=n_citta,
                n_gen=(10*(n_citta)^2)/n_formiche,seme=NULL)
{
m<-n_formiche # numero di formiche
n<-n_citta # numero di citta'
ngen<-as.integer(n_gen) # numero di generazioni da fare
c2<-NULL # costo in ogni generazione
minimo<-NULL # 1 casella = 1 generazione
media<-NULL # 1 casella = 1 generazione
pesi<-NULL # prob. di riproduzione
genitori<-NULL  # array di due caratteri per la riproduzione

costi<-matrix(data=v_costi,nrow=n,ncol=n,byrow=TRUE)
# se d(x,y)=d(y,x) byrow=TRUE e' facoltativo
formiche<-matrix(data=NA,nrow=m,ncol=n)
```

```
formicheBU<-matrix(data=NA,nrow=m,ncol=n)

if(is.null(v_costi))
  {cat("Il vettore dei costi non puo' essere NULL\n")
   return(NULL)
  }

if(is.null(seme)==FALSE)
  {set.seed(seme)
  }

med<-0
mini<-0
fetta<-0
cambiamenti<-0
# Prima generazione
for(i in 1:m) # per ogni formica
  {formiche[i,]<-sample(c(1:n),n)

  # Valutazione (calcolo del fitness)
  #  c=costo
   c<-0
  # per ogni citta' tranne la chiusura del cerchio
   for(j in 1:(n-1))
     {c<-c+costi[formiche[i,j],formiche[i,j+1]]
     }
  # chiudo il cerchio
   c<-c+costi[formiche[i,n],formiche[i,1]]
   c2<-c(c2,c)
   if(c<mini || mini==0)
    {mini<-c
     fetta<-0
     cambiamenti<-0
    }
```

```
   if(c==mini)
    {fetta<-fetta+1
    # conservo il migliore (1 solo) (tenendo la valutazione)
     migliore<-formiche[i,]
    }
   med<-med+c
  }

minimo<-c(minimo,mini)
med<-med/m
media<-c(media,med)
fetta<-(fetta/m)*100
cat("\nPrima generazione: ",fetta,"% a ",mini,"(media = ",med,")\n")
med<-0

# assegnazione della probabilita' per la riproduzione
pesi<-sum(c2)/c2  # maggiore e' il costo e minore e' la probabilita'

genfatte<-1

while(genfatte<ngen)
  {cat("Generazione ",genfatte," su ",ngen,". (")

  # calcolo delle probabilita' di mutazione
   mutaz<-(fetta/1000)+0.001 # da 0.001 a 0.101
   mutaz<-mutaz+(cambiamenti*0.00001)
   if(mutaz>0.1)
     mutaz<-0.1

  # calcolo delle probabilita' di riproduzione
   estratti<-0
   fetta<-1

  # tengo il migliore
```

```
  estratti<-estratti+1
  formicheBU[estratti,]<-migliore
  med<-med+mini
  c2<-mini

 # riproduzione
  while(estratti<m)
    {# genitori
     genitori<-sample(c(1:m),2,prob=pesi)

    # ora ho i miei due genitori
    #  --> prendo k citta' da genitori[1] e
    #  le metto in formicheBU[estatti,1:k]
    #   --> (two point cross over)
     estratti<-estratti+1
     k<-sample(c(2:(n-1)),1)
     formicheBU[estratti,1:k]<-formiche[genitori[1],1:k]

    # ora trovo in genitori[2] la citta' formicheBU[estratti,k]
    #    e chiudo il giro
     k2<-formicheBU[estratti,k]
     k2<-which(formiche[genitori[2],]==k2)

    # arr --> array che parte dalla successiva a k2
    #         e fa il giro
     arr<-NULL
     if(k2+1>n)
       arr<-c(formiche[genitori[2],1:n])
     else
       {arr<-formiche[genitori[2],(k2+1):n]
        arr<-c(arr,formiche[genitori[2],1:k2])
       }

    # da arr devo togliere
```

```
#  gli elementi di formiche[genitori[1],1:k]
 for(i in 1:k)
   {arr<-arr[which(arr!=formiche[genitori[1],i])]
   }

 if(k<n) # altrimenti formicheBU[estratti] e' gia' pieno
   formicheBU[estratti,(k+1):n]<-arr

# ora eventuale mutazione del nuovo creato
# MACROMUTAZIONE
 u<-runif(1)
 if(u<mutaz) # mutazione
   {vett<-formicheBU[estratti,] # n colonne
    partenza<-sample(1:n,1)
    quante<-sample(1:(n-2),1)
    if(partenza+quante-1<=n)
      {vett3<-vett[partenza:(partenza+quante-1)]
       if(partenza-1<1)
         vett2<-NULL
       else
         vett2<-vett[1:(partenza-1)]
       if((partenza+quante)<=n)
         vett2<-c(vett2,vett[(partenza+quante):n])
      }
    else
      {vett3<-vett[partenza:n]
       quante2<-quante-(n-partenza+1)
       vett3<-c(vett3,vett[1:quante2])
       vett2<-vett[(quante2+1):(partenza-1)]
      }

   # in vett3 ho il taglio,
   #   mentre in vett2 ho quello che mi rimane
    doveinserisco<-sample(1:(n-quante),1)
```

```
  # inserisco dopo a "doveinserisco"
   vett<-vett2[1:doveinserisco]
   vett<-c(vett,vett3)
   if(doveinserisco<(n-quante))
     vett<-c(vett,vett2[(doveinserisco+1):(n-quante)])
   formicheBU[estratti,]<-vett
   }
else
   {#  MICROMUTAZIONE
    temp<-formicheBU[estratti,1:n]
    formicheBU[estratti,1:n]<-rep(0,n)
    kk<-c(1:n)
    for(gg in 1:n)
      {u<-runif(1)
       if(u<(mutaz/10)) # mutazione
         {cambio<-sample(kk,1)
          while(cambio==gg)
             cambio<-sample(kk,1)
          formicheBU[estratti,cambio]<-temp[gg]
          temp[gg]<- -1
          kk<-kk[which(kk!=cambio)]
         }
      }
    kk<-1
    for(gg in 1:n)
      {if(formicheBU[estratti,gg]!=0)
         {next
         }
       while(temp[kk]==-1)
          kk<-kk+1
       formicheBU[estratti,gg]<-temp[kk]
       kk<-kk+1
      }
   }
```

```
  # Valutazione (calcolo del fitness)
  #  c=costo
   c<-0
   # per ogni citta' tranne la chiusura del cerchio
   for(j in 1:(n-1))
     {c <- c +
           costi[formicheBU[estratti,j],formicheBU[estratti,j+1]]
     }
  # chiudo il cerchio
   c<-c+costi[formicheBU[estratti,n],formicheBU[estratti,1]]
   c2<-c(c2,c)

  # calcolo del minimo e conseguente fetta di popolazione
   if(c<mini)
     {mini<-c
      fetta<-0
      cambiamenti<-0
     }
   if(c==mini)
     {fetta<-fetta+1
     # conservo il migliore (1 solo) (tenendo la valutazione)
      migliore<-formicheBU[estratti,]
     }
   med<-med+c
  }

minimo<-c(minimo,mini)
med<-med/m
media<-c(media,med)
fetta<-(fetta/m)*100
cat(fetta,"% a ",mini,". Media =",med,".")
cat(" Nessun cambio da ",cambiamenti," generazioni).\n")
med<-0
```

```
# assegnazione della probabilita' per la riproduzione
 pesi<-sum(c2)/c2
# maggiore e' il costo e minore e' la probabilita'

 formiche<-formicheBU
 genfatte<-genfatte+1
 cambiamenti<-cambiamenti+1

# SI VUOLE CONTINUARE ?
 if(genfatte>=ngen)
   {while(TRUE)
     {cat("Evoluzione completa. Si vuole continuare ?\n")
      cat(" (s=si, n=no, v=visualizza)\n")
      c<-readline()
      if(c=="s")
        {cat("Per quante generazioni ?\n")
         c<-readline()
         c<-ngen+as.integer(c)
         if(is.na(c))
           {next
           }
         ngen<-c
         break
        }
      else if(c=="n")
        {break
        }
           else if(c=="v")
                  {finale<-matrix(data=NA,nrow=m,ncol=n+1)
                   for(i in 1:m)
                     {finale[i,1:n]<-formiche[i,1:n]
                      finale[i,n+1]<-c2[i]
                     }
```

```
                    # inserimento per colonne
                     e<-c(minimo,media)
                     evoluzione<-matrix(data=e,nrow=ngen,ncol=2,
                     dimnames=list(NULL,c("min","media")))

                     print(finale)
                     rm(finale)
                    }
       }
     }  # fine del "si vuole continuare"
  }  # fine del while(genfatte<ngen)

finale<-matrix(data=NA,nrow=m,ncol=n+1)
# contiene la generazione finale

for(i in 1:m)
  {finale[i,1:n]<-formiche[i,1:n]
   finale[i,n+1]<-c2[i]
  }

# inserimento per colonne
e<-c(minimo,media)
evoluzione<-matrix(data=e,nrow=ngen,ncol=2,
                   dimnames=list(NULL,c("min","media")))

plot(evoluzione[,1],type='l',ylim=c(min(evoluzione[,1]),
         max(evoluzione[,2])),col='blue',
         main='Evoluzione della Media e del Minimo',
         xlab='Numero Generazione',ylab='Valore')
lines(evoluzione[,2],type='l',col='red')
legenda<- c(expression(paste("Media"),paste("Minimo")))
legend((length(evoluzione[,1]))-(length(evoluzione[,1])/3),
        max(evoluzione[,2]),legend=legenda,lty=1,lwd=3,
```

```
        col=c('red','blue'))

return(list(pop=finale,evol=evoluzione))
}
```

Appendice C

Codice R del Master Mind Alternativo

In questa appendice presentiamo il codice R alternativo del gioco del Master Mind descritto nel quinto capitolo.

C.1 Codice R commentato

```
####################################
# funzione valuta(prove[i,],m[j,]) #
####################################
# ritorna la valutazione della prova a rispetto alla prova b
#  in termini di (nero,bianco)
#nb<-valuta(m[i,],m[j,])

valuta<-function(a,b)
{neri<-0
bianchi<-0
lungh<-length(a)

if(lungh!=length(b))
  {cat("Errore nella valutazione: i due vettori")
   cat(" hanno dimensione diversa.\n")
```

```
   return(c(0,0))
  }

# prima i neri
z<-a==b
neri<-sum(z)
for(i in 1:lungh)
  {if(z[i])
    {a[i]<-0
     b[i]<-0
    }
  }

# poi i bianchi
for(i in 1:lungh)
  {for(j in 1:lungh)
    {if(a[i]==0)
       break  # esci dal for(j) e incrementa i
     if(b[j]==0)
       next   # incrementa j
     if(a[i]==b[j])
      {b[j]<-0
       bianchi<-bianchi+1
       break # esci dal for(j) e incrementa i
      }
    }
  }

return(c(neri,bianchi))
}
```

```
############################
# funzione cerca(m,l,s,k) #
############################
# m --> matrice
# l --> lunghezza di una riga della matrice
# s --> stringa precedente
# k --> variabile interna
# questa funzione serve a cercare tutte le possibili combinazioni
#  "matematicamente differenti" (tris,coppia,doppia coppia,...)

cerca<-function(m,l,s,k)
{for(i in 1:k)
  {j<- k-i

   if(j<1)
    return(m)

   t<-c(s,i,j,rep(0,l-length(s)-2))
   t<-t[order(t)]

  # guarda che non ci sia gia' in m
   messo<-0
   for(jj in 1:dim(m)[1])
    {if(sum(m[jj,]==t)==l)
      {messo<-1
       break
      }
    }
   if(messo==0)  # non c'e' ancora
    {m<-rbind(m,t,deparse.level=0)
    }

   s2<-c(s,i)
```

```
   m<-cerca(m,l,s2,j)
  }
return(m)
}
```

```
#########################
# funzione crea(n,poss) #
#########################
# m2<-crea(ncase,possibili)
# funzione che serve ad ottimizzare la prima prova

crea<-function(n,poss)
{
po<-NULL
s<-NULL
temp<-rep(0,n-1)
temp<-c(temp,n)
po<-rbind(po,temp,deparse.level=0)  # temp e' ordinato
po<-cerca(po,n,s,n)

# ora in "po" ho tutte le possibili combinazioni
# al massimo devono avere tanti colori diversi
#            quanti colori possibili
if(length(poss)<n)
  {po2<-NULL
   for(i in 1:dim(po)[1])
    {temp<-po[i,which(po[i,]!=0)]
     if(length(temp)<=length(poss))
      {po2<-rbind(po2,po[i,],deparse.level=0)
      }
```

```
    }
   po<-po2
   po2<-NULL
  }

# per ognuna di queste combinazioni provo un giro
# quindi mi gioco la migliore (col valore minore)
prove<-NULL
for(i in 1:dim(po)[1])
  {temp<-po[i,which(po[i,]!=0)]
   prove2<-NULL
   possib<-poss

   for(j in 1:length(temp))
    {if(length(possib)>1)
      {estr<-sample(possib,1)
      }
     else
      {estr<-possib
      }

     possib<-possib[which(possib!=estr)]
     estr<-rep(estr,temp[j])
     prove2<-c(prove2,estr)
    }
   prove2<-sample(prove2,n)
   prove<-rbind(prove,prove2,deparse.level=0)
  }

# ora ho in "prove" i "campioni" da provare

temp<-rep(poss[1],n)
m<-matrix(data=temp,ncol=n,byrow=TRUE)
punt<-rep(1,n)
```

```
esci<-0

while(TRUE)
  {riporto<-1

   for(i in length(temp):1)
    {if(riporto==1)
      {punt[i]<-punt[i]+1
       riporto<-0
      }
     if(punt[i]>length(poss))
      {if(i==1) # ho finito le sequenze
        {esci<-1
         break
        }
       punt[i]<-1
       riporto<-1
      }
     temp[i]<-poss[punt[i]]
# posso chiudere prima perche' ho gia' la mia sequenza
#  (i successivi non cambiano)
     if(riporto==0)
      {break
      }
    }

   if(esci==1)
     break
   m<-rbind(m,temp,deparse.level=0)
  }

# ora ho in "prove" i "campioni" da provare
# e in m la matrice con tutte le generazioni
```

```
minimo<- -1
nn<-0
xmin<-NULL

for(i in 1:dim(prove)[1])
  {ntot<-0
   xminBU<-NULL

   for(j in 1:dim(m)[1])
    {nb<-valuta(prove[i,],m[j,])

    # e tengo in xminBU [numeroriga, neri, bianchi]
     xminBU<-rbind(xminBU,c(j,nb),deparse.level=0)
    }

  # ora ho in xminBU il confronto fra prove[i,] e tutti gli altri
  # prendo tutti le possibili combinazioni nero\bianco
   u<-unique(xminBU[,2:3])
  # quindi ora u e' una matrice (o vettore) nx2,
  #  con colonne [neri,bianchi]
   if(is.matrix(u)==TRUE)
    {for(j in 1:dim(u)[1])  # per ogni riga di u
      {ntot <- ntot +
                 (sum(xminBU[,2]==u[j,1] & xminBU[,3]==u[j,2])^2)
      }
    }
   else
    {# e' un vettore --> xminBU[,2] e xminBU[,3] sono tutte uguali
     ntot <- ntot + ((dim(xminBU)[1])^2)
    }
cat("Prova: ",prove[i,],". Valore: ",ntot,".")
cat(" (Valore medio: ",ntot/dim(m)[1],").\n")
   if(ntot==minimo)
    {nn<-nn+1
```

```
     chimin<-sample(c(chimin,i),1,prob=c(nn,1))
     if(chimin==i)
      {xmin<-xminBU
       xminBU<-NULL
      }
    }

   if(ntot<minimo || minimo==-1)
    {minimo<-ntot
     chimin<-i
     nn<-1
     xmin<-xminBU
     xminBU<-NULL
    }
  }

m<-rbind(prove[chimin,],m,deparse.level=0)

return(list(m=m,xmin=xmin))
}
```

```
########################
# funzione mmalt3(...) #
########################
# funzione principale

mmalt3<-function(soluz=sample(c(1:ncol),5),ncol=8,
                                seme=rnorm(1,0,1000))
{
ncolori<-ncol
```

```
ncase<-length(soluz)
neri<-ncase+1
bianchi<-ncase+2
# matrice delle prove + neri + bianchi
mm<-matrix(data=NA,nrow=100,ncol=ncase+2)
soluzione<-soluz  # soluzione scelta
pp<-0   # numero di prove fatte

set.seed(seme)

# controllo colori\soluz
if(max(soluz)>ncol)
  {cat("Errore nei parametri di output: max(soluz)>ncol.\n")
   return(c(soluz,0,seme))
  }
if(min(soluz)<=0)
  {cat("Errore nei parametri di output: min(soluz)<=0.\n")
   return(c(soluz,0,seme))
  }
if(min(ncol)<=0)
  {cat("Errore nei parametri di output: min(ncol)<=0.\n")
   return(c(soluz,0,seme))
  }


# Stampo a video la soluzione
cat("\nSecret code: [",soluzione,"]\n--------------------\n")

# Passo 1: creazione di tutti le possibili soluzioni
m<-crea(ncase,c(1:ncolori))
xmin<-m$xmin
m<-m$m

# in m ho: la prima riga e' quella da provare.
```

```
#  tutto il resto e' m davvero
# la provo e la valuto
pp<-pp+1

mm[pp,1:ncase]<-m[1,]
mm[pp,neri:bianchi]<-valuta(mm[pp,1:ncase],soluz)

# se la valutazione e' [ncase,0] --> ho vinto
if(mm[pp,neri]==ncase)
  {cat("# ",pp," : ",mm[pp,1:ncase],"[",mm[pp,neri],",")
   cat(mm[pp,bianchi],"] --> HO TROVATO IL CODICE !!!\n\n")
   return(c(soluz,pp,seme))
  }

m<-m[2:dim(m)[1],]
# scrematura
#  la matrice xmin --> colonne: [numeroriga, neri, bianchi]
m<-m[xmin[which(xmin[,2]==mm[pp,neri] &
     xmin[,3]==mm[pp,bianchi]),1],]

# stampo a video il risultato del tentativo
cat("# ",pp," : ",mm[pp,1:ncase],"[",mm[pp,neri],",")
cat(mm[pp,bianchi],"] (Numero di soluzioni possibili:")
if(is.matrix(m)==TRUE)
  cat(dim(m)[1],")\n")
else
  cat("1 )\n")


while(TRUE)
 {minimo<- -1
  nn<-0
  xmin<-NULL
```

```
if(is.matrix(m)==TRUE)
 {for(i in 1:dim(m)[1])
  {ntot<-0
   xminBU<-NULL

   for(j in 1:dim(m)[1])
    {if(j==i)
      next

     nb<-valuta(m[i,],m[j,])

    # e tengo in xminBU [numeroriga, neri, bianchi]
     xminBU<-rbind(xminBU,c(j,nb),deparse.level=0)
    }

  # ora ho in xminBU il confronto fra m[i,] e tutti gli altri
  # prendo tutti le possibili combinazioni nero\bianco
   u<-unique(xminBU[,2:3])
  # quindi ora u e' una matrice (o vettore) nx2,
  #  con colonne [neri,bianchi]
   if(is.matrix(u)==TRUE)
    {for(j in 1:dim(u)[1]) # per ogni riga di u
      {ntot <- ntot +
               (sum(xminBU[,2]==u[j,1] & xminBU[,3]==u[j,2])^2)
      }
    }
   else
    {# e' un vettore --> xminBU[,2] e xminBU[,3] sono tutte uguali
     ntot <- ntot + ((dim(xminBU)[1])^2)
    }

   if(ntot==minimo)
    {nn<-nn+1
     chimin<-sample(c(chimin,i),1,prob=c(nn,1))
```

```
      if(chimin==i)
       {xmin<-xminBU
        xminBU<-NULL
       }
     }

    if(ntot<minimo || minimo==-1)
     {minimo<-ntot
      chimin<-i
      nn<-1
      xmin<-xminBU
      xminBU<-NULL
     }
   }

  riga<-chimin

 # la provo e la valuto
  pp<-pp+1
  if(pp>100)
   {cat("Errore: ci vogliono piu' di 100 prove !!!\n")
    return(c(soluz,pp,seme))
   }
  mm[pp,1:ncase]<-m[riga,]
  mm[pp,neri:bianchi]<-valuta(mm[pp,1:ncase],soluz)

  if(mm[pp,neri]==ncase)
   {# se la valutazione e' [ncase,0] --> ho vinto
    cat("# ",pp," : ",mm[pp,1:ncase],"[",mm[pp,neri],",")
    cat(mm[pp,bianchi],"] --> HO TROVATO IL CODICE !!!\n\n")
    break
   }

 # scrematura
```

```
  #  la matrice xmin --> colonne: [numeroriga, neri, bianchi]
   m<-m[xmin[which(xmin[,2]==mm[pp,neri] &
        xmin[,3]==mm[pp,bianchi]),1],]

  # stampo a video il risultato del tentativo
   cat("# ",pp," : ",mm[pp,1:ncase],"[",mm[pp,neri],",")
   cat(mm[pp,bianchi],"] (Numero di soluzioni possibili:")
   if(is.matrix(m)==TRUE)
    cat(dim(m)[1],")\n")
   else
    cat("1 )\n")

   }  # fine if(is.matrix(m)==TRUE)
  else
  {# m non e' una matrice --> e' un vettore
   #     --> e' l'unica soluzione possibile
   pp<-pp+1
   if(pp>100)
    {cat("Errore: ci vogliono piu' di 100 prove")
     cat(" (... proprio alla fine ...) !!!\n")
     return(c(soluz,pp,seme))
    }
   mm[pp,1:ncase]<-m
   mm[pp,neri:bianchi]<-valuta(mm[pp,1:ncase],soluz)

# se la valutazione e' [ncase,0] --> ho vinto
   if(mm[pp,neri]==ncase)
    {cat("# ",pp," : ",mm[pp,1:ncase],"[",mm[pp,neri],",")
     cat(mm[pp,bianchi],"] --> HO TROVATO IL CODICE !!!\n\n")
     break
    }
   else # non e' la soluzione --> errore !!!
    {cat("# ",pp," : ",mm[pp,1:ncase],"[",mm[pp,neri],",")
     cat(mm[pp,bianchi],"]\n")
```

```
     cat("ERRORE: non ho piu' soluzioni disponibili !!!\n")
     break
    }
  }
 }  # fine while(TRUE)

return(c(soluz,pp,seme))
# ritorno la soluzione e il numero di tentativi fatti per trovarla
}
```

Bibliografia

[1] Carlo Carta, Carla Corda,
'*Algoritmi Genetici*',
'http://web.tiscali.it/vitaartificiale/veloce.htm'.

[2] Dario Floreano, Claudio Mattiussi,
'*Manuale sulle reti neurali*', seconda edizione, Il Mulino, 2002.

[3] Cinzia Pisani,
'*Gli algoritmi genetici ed alcune applicazioni*',
Tesi di Laurea in Matematica, Università di Torino, 2001,
'http://alpha01.dm.unito.it/personalpages/cerruti/studenti/Pisani'.

[4] AAVV,
'*Wikipedia*',
'http://it.wikipedia.org/wiki/Pagina_principale'.

[5] AAVV,
'*R Help On Line v. 2.2.0*', 2005,
'www.r-project.org'.

[6] AAVV,
'*TSPLib*',
'http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/'.

[7] Eric B. Baum, Dan Boneh, Charles Garrett,
'*Where Genetic Algorithms excel*', 1995,
'http://citeseer.ist.psu.edu/baum95where.html'.

[8] Luís Bento, Luísa Pereira, Agostinho Rosa,
'*Master Mind by Evolutionary Algorithms*',
'http://laseeb.isr.ist.utl.pt/publications/papers/acrosa/sac99/sac99.html'.

[9] J.J. Merelo, J. Carpio, P. Castillo, V.M. Rivas, G. Romero,
'*Finding a needle in a haystack using hints and evolutionary computation: the case of Genetic Mastermind*',
Departamento de Arquitectura y Tecnología de los Computadores,
Universidad de Granada,
'http://geneura.ugr.es/~jmerelo/newGenMM/newGenMM.html'.

[10] Radu Rosu,
'*Analysis of the game of Mastermind - the m^n case*',
Report on research done for the Undergraduate Honors Thesis,
North Carolina State University, 1997,
'http://www.csc.ncsu.edu/academics/undergrad/Reports/rtrosu/index.html'.

[11] Alexandre Temporel, Tim Kovacs,
'*A heuristic hill climbing algorithm for Mastermind*',
'http://www.cs.bris.ac.uk/Publications/pub_info.jsp?id=2000067'.

[12] J.E. Baker,
'*Reducing bias and inefficiency in the selection algorithm*',
in J.J. Grefenstette,
'*Proceedings of the second international conference on genetic algorithms*',
Hillsdale, NJ, Lawrence-Erlbaum Associates, pp. 14-21, 1987.

[13] A.D. Bethke,
'*Genetic Algorithms as Function Optimizers*',
Doctoral Dissertation, University of Michigan, 1980.

[14] Gary Darby,
'*Delphi for fun*',
'www.delphiforfun.org/Programs/MasterMind.htm'.

[15] L. Davis,
'*Genetic Algorithms and Simulated Annealing*',
London, Pitman, 1987.

[16] L. Davis,
'*Applying Adaptive Algorithms to Epistatic Domains*',
in '*Proceedings of the International Joint Conference on Artificial Intelligence*', (pp.162-164),
Lawrence Erlbaum Associates, Hillsdale, 1985.

[17] R. Dawkins,
'*The Blind Watchmaker*',
London, Penguin Books, 1986,
trad. it. '*L'orologiaio cieco*', Milano, Rizzoli, 1988.

[18] B.R. Fox, M.B. McMahon,
'*Genetic Operators for Sequencing Problems*',
in G. Rawlins, '*Foundations of Genetic Algorithms*',
(pp. 284-300), 1991.

[19] D.E. Goldberg,
'*Genetic Algorithms in Search, Optimization and Machine Learning*',
Reading, MA, Addison-Wesley, 1989.

[20] D.E. Goldberg e K. Deb,
'*A comparative analysis of selection schemes used in genetic algorithms*',
in G.J.E. Rawlins, '*Foundations of genetic algorithms*',
San Mateo, CA, Morgan Kaufmann, 1991.

[21] D.E. Goldberg, R. Lingle,
'*Alleles, loci and the TSP*',
in J.J. Grefenstette,
'*Proceedings of the First International Conference on Genetic Algorithms*', (pp.154-159),
Lawrence Erlbaum Associates, Hillsdale, 1985.

[22] J.J. Grefenstette,
'*Optimization of control parameters for genetic algorithms*',
in '*IEEE Transactions on systems, man and cybernetics*',
vol.16, pp.122-128, 1986.

[23] J.J. Grefenstette, R. Gopal, B. Rosmaita, D. Van Gucht,
'*Genetic Algorithm for the TSP*',
in J.J. Grefenstette,
'*Proceedings of the First International Conference on Genetic Algorithms*', (pp.160-168),
Lawrence Erlbaum Associates, Hillsdale, 1985.

[24] F. Hoffmeister e T. Bäck,
'*Genetic Algorithms and Evolution Strategies: Similarities and Differences*',
in H.P. Schwefel e R. Maenner, '*Parallel Problem Solving from Nature*',
pp. 455-469, Berlin, Springer-Verlag, 1991.

[25] J.H. Holland,
'*Adaptation in Natural and Artificial Systems*',
University of Michigan Press, Ann Arbor, 1975.

[26] A. Homaifar, S. Guan,
'*A New Approach on the Traveling Salesman Problem by Genetic Algorithm*',
Technical Report, North Carolina & T State University, 1991.

[27] D.S. Johnson,
'*Local Optimization and the Traveling Salesman Problem*',
in M.S. Paterson,
'*Proceedings of the 17^{th} Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science*',
(pp.446-461), Springer-Velag, 1990.

[28] I.M. Oliver, D.J. Smith, J.R.C. Holland,
'*A Study of Permutation Crossover Operators on the Traveling Salesman Problem*',
in J.J. Grefenstette,
'*Proceedings of the Second International Conference on Genetic Algorithms*', (pp.224-230),
Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

[29] I. Rechenberg,
'*Evolutionstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*',
Stuttgart, Friedrich Fromann Verlag, 1973.

[30] H.P. Schwefel,
'*Numerical optimization of computer models*',
Chichester, Wiley, 1981.

[31] D. Seniw,
'*A Genetic Algorithm for the Traveling Salesman Problem*',
MSc Thesis, University of North Carolina AT Charlotte, 1991.

[32] W.M. Spears,
'*Crossover or Mutation ?*',
in L.D. Whitley, '*Foundations of Genetic Algorithms 2*',
Morgan Kaufmann Pubblishers, 1993.

[33] G. Syswerda,
'*Uniform Crossover in Genetic Algorithms*',
in J.D. Schaffer,
'*Proceedings of the Third International Conference on Genetic Algorithms*',
San Mateo, CA, Morgan Kaufmann, 1989.

[34] D. Whitley,
'*GENITOR: A Different Genetic Algorithm*',
in '*Proceedings of the Rocky Mountain Conference on Artificial Intelligence*',
Denver, CO, 1988.

[35] L.D. Whitley, T. Starkweather, D'A. Fuquay,
'*Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator*',
in J. Schaffer,
'*Proceedings of the Third International Conference*

on Genetic Algorithms',
(pp.133-140), Morgan Kaufmann Pubblishers, 1989.

Ringraziamenti

Innanzi tutto volevo ringraziare il professor Stuart Coles per avermi permesso di fare questa tesi e per il tempo e l'aiuto che mi ha dedicato in questi mesi.

Un grazie va alla mia grande famiglia sparsa in tutta Italia: ovunque vada so di poter sempre contare su qualcuno.

Un grazie enorme va ai miei genitori, mio fratello e a Tigre (buon'anima) che mi hanno sopportato per i primi 26 anni di vita e spero mi debbano sopportare ancora a lungo.

Naturalmente non mi sono dimenticato degli amici, un grazie anche a loro, di tutto cuore.

Un ringraziamento particolare va ai professori e agli ex compagni dello S.M.I.D. di Genova: senza di loro, sicuramente, non sarei arrivato fin qui (come diceva Confucio, 'La felicità più grande non sta nel non cadere mai, ma nel risollevarsi sempre dopo una caduta').

Un ultimo grazie (ultimo non certo per importanza) va a Marco e ad Antonella e alle rispettive famiglie (escluso il topo, naturalmente! ;-)), che mi hanno ospitato con grande gentilezza per tutto l'anno 'Padovano' e mi ha fatto passare delle splendide giornate.

A tutti voi va un grazie gigantesco e un augurio di una vita intensa e felice.

DR