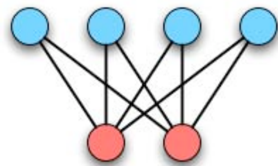
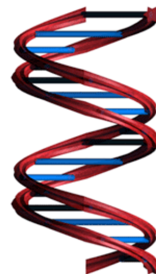
INTELLIGENZA ARTIFICIALE EVOLUTIVA

Sviluppo della coordinazione senso-motoria in agenti robot





Matteo Cortonesi, 4F
Lavoro di Maturità in Fisica
Anno scolastico 2003-2004
Prof. Responsabile: Fiorenzo Sainini

INDICE

1. INTRODUZIONE **3**

2. LE RETI NEURALI **5**

2.1 Il connessionismo 5
2.2 Le reti neurali 6
2.2.1 Il neurone artificiale *7*
2.2.2 L'architettura di una rete neurale *14*
2.2.3 Codice d'esempio *15*

3. GLI ALGORITMI GENETICI **18**

3.1 Introduzione 18
3.2 L'algoritmo 19
3.2.1 Creazione di una popolazione *20*
3.2.2 Inizializzazione casuale dei geni *20*
3.2.3 Decodificazione del genoma *20*
3.2.4 Test del problema e calcolo del fitness dell'individuo *20*
3.2.5 Riproduzione selettiva *21*
3.2.6 Incrocio a coppie *21*
3.2.7 Mutazione *22*
3.3 Codice d'implementazione 23
3.3.1 La routine principale *24*
3.3.2 Generazione casuale della popolazione *26*
3.3.3 Calcolo della lunghezza del percorso *27*
3.3.4 Riproduzione selettiva *27*
3.3.5 L'incrocio *28*
3.3.6 La mutazione *31*
3.3.7 Risultati *31*

4. GLI ORGANISMI ARTIFICIALI **35**

4.1 Vita Artificiale 35
4.2 Evoluzione di organismi artificiali 35
4.2.1 Combinazione reti neurali e algoritmi genetici *35*
4.2.2 Implementazione *36*

5. SVILUPPO DELLA COORDINAZIONE SENSOMOTORIA IN AGENTI ROBOT **37**

5.1 Introduzione 37
5.2 Simulazioni e robot reali 37
5.3 Evoluzione di un robot reale 58
5.3.1 Materiale *58*
5.3.2 La funzione di fitness *59*
5.3.3 L'architettura dell'organismo artificiale *61*
5.3.4 Implementazione e risultati *62*

6. INDICE DEL CONTENUTO DEL CD-ROM **69**

7. RIFERIMENTI BIBLIOGRAFICI **69**

1. Introduzione

Da sempre l'uomo è stato affascinato da macchine che mostravano comportamenti simili a quelli di un organismo vivente. I primi robot apparvero già all'inizio del XVIII secolo con la rivoluzione industriale, ad esempio nell'industria tessile con la progettazione dei telai. Essi erano dei dispositivi meccanici composti da meccanismi il cui compito era quello di svolgere una predeterminata serie di azioni. Ma i primi "veri" robot furono realizzati nel XX secolo, più precisamente negli anni quaranta con lo sviluppo dell'elettronica, dei computer, e dei sensori artificiali. Gli odierni robot sono basati su uno o più microprocessori che possono elaborare i dati inviati dai vari sensori. Pur essendoci stata una rivoluzione in questo campo la maggior parte dei robot presenti oggi non sono molto differenti dai loro antenati. Ciò è dovuto al fatto che essi continuano ad essere programmati per svolgere una predeterminata serie di operazioni benché più complesse. Sono da ritenere intelligenti queste macchine? Molti saranno d'accordo che non lo sono. Allora perché mostrano, a prima vista, caratteristiche simili ad un comportamento intelligente? Semplicemente, esse riflettono l'intelligenza dell'ingegnere che le ha programmate per svolgere quei determinati compiti.

Questa parte dell'Intelligenza Artificiale è basata sul cognitivismo computazionale. Esso è nato infatti alla fine degli anni quaranta del secolo scorso con l'avvento del computer, sfruttando l'analogia tra mente umana e computer, o meglio, tra mente umana e software del computer, che è un insieme di regole e di istruzioni per manipolare i simboli in modo formale, ossia non tenendo conto del loro significato. Allo stesso modo la mente umana viene considerata un insieme di rappresentazioni simboliche (e quando si tratta di interpretare anche i significati essi vengono visti come altri simboli). E proprio grazie a questa analogia gli informatici hanno cercato di dotare il computer di capacità e comportamenti tipici della mente umana, come il riconoscimento del linguaggio parlato, il riconoscimento di oggetti visivi (ad esempio la calligafria), la traduzione da una lingua all'altra, ecc. La risoluzione di problemi come questi, per il cognitivismo computazionale, consiste nell'analizzarli attentamente e scomporli in sottoproblemi a ciascuno dei quali viene applicato un sistema di regole, che si occupa di trovare una soluzione specifica (si veda ad esempio il linguaggio di programmazione *LISP*). Da un punto di vista algoritmico le soluzioni più frequenti implementate consistono nel creare delle grandi *look up table* dove ogni voce corrisponde ad un'espressione del tipo *if-then*. Questi software normalmente devono essere molto performanti, con assenza di errori, avere una memoria abbastanza grande e possedere un hardware con cui comunicare che sia molto preciso in fatto di misurazioni. Ci si può domandare se questi requisiti corrispondano effettivamente alle caratteristiche degli organismi biologici. Quindi quando un organismo artificiale è intelligente?

Molti ricercatori hanno cercato di dare una risposta a questa domanda e sembrano essere giunti ad una conclusione. Un organismo artificiale per essere considerato intelligente deve possedere una caratteristica fondamentale: la capacità di risolvere problemi in modo autonomo mediante un processo di interazione con l'ambiente, ossia essere in grado di adattarsi. E la conseguenza di ciò è l'evoluzione. Ecco perché in questo lavoro lo scopo sarà di trattare l'intelligenza artificiale dal punto di vista evoluzionistico.

Dando uno sguardo però alla storia della psicologia e filosofia si può notare come siano tenuti in considerazione molti altri elementi che caratterizzano l'intelligenza, come ad esempio: le capacità linguistiche, la socializzazione, la coscienza, il lavoro di gruppo ecc. In ogni modo queste proprietà sono semplicemente il risultato del processo di evoluzione di un sistema complesso che si è adattato in un ambiente altrettanto complesso e vario. Esse quindi non devono essere viste come la base costituente dell'intelligenza, ma bensì come delle proprietà emergenti che si sono sviluppate per rispondere a dei problemi imposti dall'ambiente. L'interazione con l'ambiente, dunque, assume un ruolo primario nella fase dello sviluppo di un sistema intelligente. Sembrerebbe infatti assurdo pensare che la mente non tenga in considerazione l'ambiente fisico per evolversi, dato che è nata proprio per fornire comportamenti adeguati per muoversi nell'ambiente stesso.

In questo lavoro si vuole verificare come diversi comportamenti possano emergere da un organismo artificiale senza che essi siano stati "imposti". Esso sarà dotato di un sistema sensomotorio che dovrà interagire con un ambiente ed evolversi sotto i criteri della "selezione naturale". Per condurre quest'esperimento non si utilizzeranno algoritmi della scienza cognitiva computazionale, ma saranno utilizzati gli algoritmi genetici e le reti neurali che derivano dall'approccio connessionista.

2. Le reti neurali

2.1 Il connessionismo

Negli ultimi 15-20 anni la scienza cognitiva computazionale si è indebolita per tre ragioni.

La prima è che le neuroscienze hanno fatto progressi molto grandi e quanto è emerso da studi recenti è che non sembra più così plausibile studiare la mente ignorando il cervello, più in generale ignorando il corpo. È vero che se si riuscisse a costruire un modello puramente astratto di qualche capacità o comportamento si potrebbero poi cercare le strutture corrispondenti nel cervello (come ad esempio individuare le zone che corrispondono ad una specifica funziona, come la vista). Molti neuropsicologi stanno cercando di fare proprio questo. Ma il problema è un altro: ha senso prima spiegare il comportamento con modelli mentali e poi tradurre questi modelli in termini di un sistema fisico come il cervello, invece di cercare fin dall'inizio delle spiegazioni fisico-chimiche che costituiscono i processi del cervello che a loro volta producono il comportamento?

La seconda ragione è che si è cominciato a pensare che l'analogia tra mente e computer non sia più così credibile. Infatti non è mai stato dimostrato che la mente (umana e non) funzioni come un computer. Essa non è il frutto di sé stessa, ma anche dal corpo (soprattutto dal cervello).

La terza ragione è che negli ultimi 20 anni è nato un nuovo approccio dello studio del comportamento che è completamente opposto a quello della scienza cognitiva computazionale. Questo nuovo approccio si chiama connessionismo.

Il connessionismo nasce come un tentativo di trovare un modello della mente più plausibile dal punto di vista biologico. Esso è opposto al cognitivismo poiché cerca di far poggiare il suo modello sulla struttura bio-fisica del sistema nervoso. Non esiste più il rapporto computer-mente; il computer non è più il modello della mente ma è semplicemente uno strumento per l'emulazione della stessa. Il connessionismo è basato su dei *processi paralleli distribuiti* (PPD). Comunemente questi sono associati ad un processore il cui funzionamento è ispirato a quello biologico di un neurone. Questo processore è a sua volta connesso ad altri processori. L'insieme dei processori è definito rete neurale. Ogni processore riceve dei segnali dagli altri processori a cui è connesso che sono filtrati attraverso i pesi delle rispettive connessioni (sinapsi). Questi segnali costituiscono l'input netto del processore che viene nuovamente filtrato attraverso una funzione e propagato ad altri collegamenti in uscita con altri processori che formano nuove sinapsi.

I principali vantaggi del connessionismo sono che esso ha una grande robustezza e tolleranza agli errori, ha una grande flessibilità (ossia si può facilmente adeguare a un nuovo ambiente attraverso l'apprendimento), può avere a che fare con informazioni poco ordinate o confuse, ha una certa facilità di implementazione. Tutte queste caratteristiche vengono esibite dalle reti neurali.

Le reti neurali, che sono un'applicazione del connessionismo, al giorno d'oggi sono sempre più utilizzate in svariati settori, ma soprattutto sono diventate lo strumento d'indagine prediletto di una nuova scienza: la vita artificiale. Questa non si limita a simulare solo il sistema nervoso di un organismo, ma ne simula anche il corpo, l'ambiente circostante e il materiale genetico acquisito. Sarà trattato questo argomento nel capitolo 3.

Si è più volte detto che le reti neurali sono basate sui sistemi biologici. Ciò però non significa che il fine ultimo sia quello di costruire un cervello artificiale. Infatti le reti neurali non possono venire considerate come la traduzione della funzionalità del cervello in termini matematici, esse riproducono solo approssimativamente i circuiti dei neuroni biologici. Quindi è bene sottolineare che l'obbiettivo del connessionismo è quello di produrre macchine ispirate a sistemi neurali biologici che possiedano un comportamento intelligente ma non quello di ricostruire il cervello. Comunque l'obbiettivo contiene due approcci: uno è quello ingegneristico, il quale è più interessato al funzionamento della macchina in sé e della sua applicazione; l'altro è quello scientifico, il quale studia questi sistemi artificiali per poter trarne delle informazioni che potrebbero essere utili per capire meglio come funzionano i sistemi intelligenti e biologici.

2.2 Le reti neurali

Le reti neurali sono nate intorno agli anni '40 grazie a vari contributi interdisciplinari. I primi che elaborarono modelli ispirati a quello di un neurone e alla sua interazione con gli altri furono Warren McCulloch e Walter Pitts. Successivamente psicologi, ingegneri e informatici apportarono un altro grande contribuito introducendo vari algoritmi di apprendimento.

Siccome le reti neurali hanno basi neuro-biologiche è opportuno dare uno sguardo alle principali caratteristiche dei sistemi neurali biologici.

Certo si deve ammettere che al giorno d'oggi le nostre conoscenze sul cervello sono ancora abbastanza limitate per poter spiegare le basi dell'intelligenza, non si può negare però che dal punto di vista anatomico e fisiologico possediamo molte informazioni riguardo alla cellula fondamentale del cervello: il neurone. Un neurone può essere considerato come il mattone elementare del cervello, tutte le operazioni di *calcolo* risiedono in questa unità. Di questa tipologia di cellule, delle quali sono state identificate le principali reazioni biochimiche che regolano la loro attività, ne esistono circa 100 sottoclassi. In ogni modo il loro funzionamento è grosso modo uguale. Il neurone, per comunicare con gli altri, genera un segnale che viene trasmesso come una variazione di energia potenziale elettrica (figura 2.1) che si propaga lungo l'assone (l'*output* del neurone) se la variazione elettrica del corpo del neurone supera una determinata soglia. I dendriti invece corrispondono agli *input* del neurone, essi sono in contatto con gli assoni degli altri neuroni. La connessione tra l'assone e il dendrite è chiamata sinapsi.

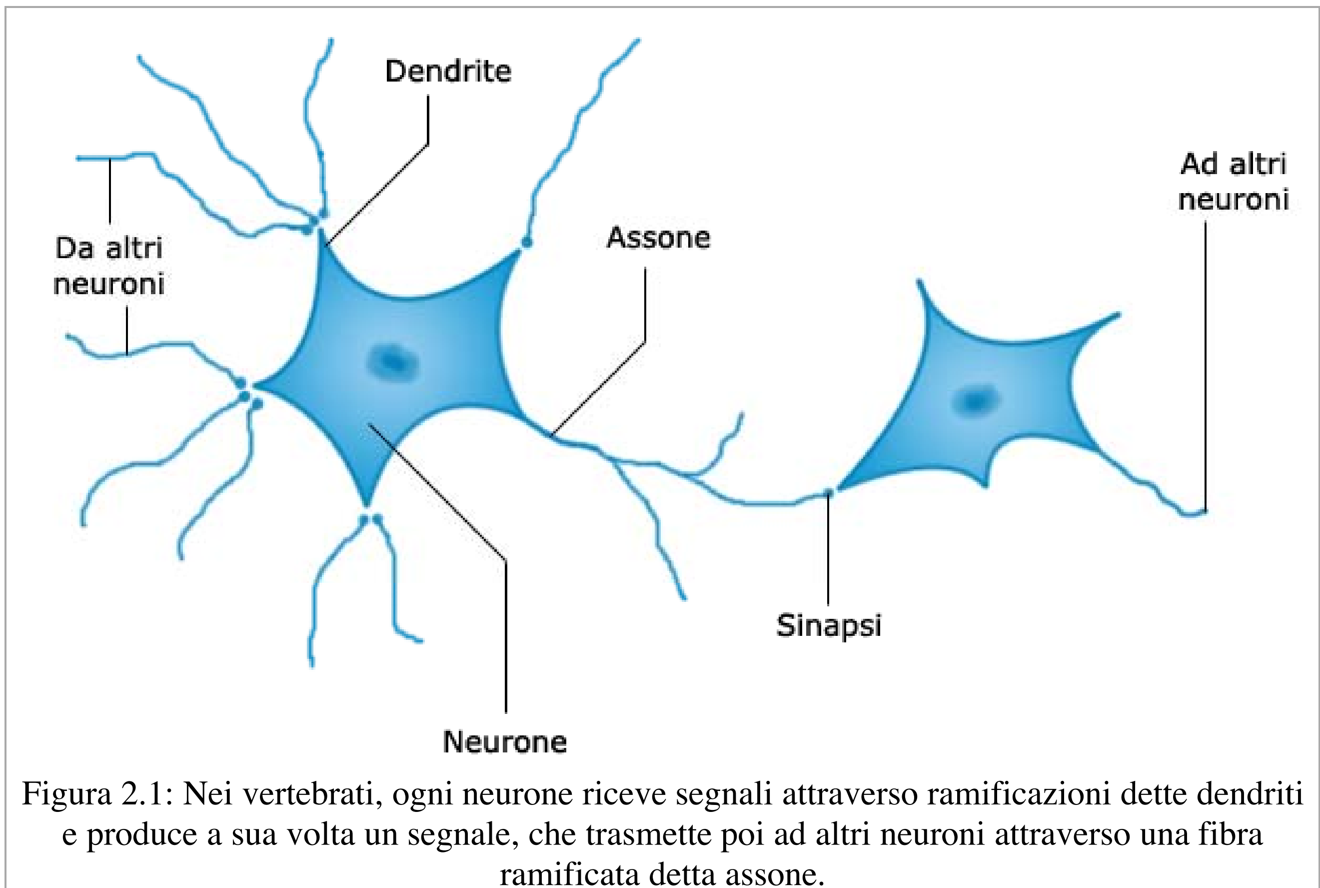


Figura 2.1: Nei vertebrati, ogni neurone riceve segnali attraverso ramificazioni dette dendriti e produce a sua volta un segnale, che trasmette poi ad altri neuroni attraverso una fibra ramificata detta assone.

La funzione della sinapsi è molto importante. Essa infatti, oltre a svolgere altre funzioni, possiede una proprietà particolare, ossia quella di poter influenzare l'impulso elettrico proveniente dall'assone di un altro neurone. Infatti il potenziale elettrico generato da un neurone può solamente essere *nullo* oppure *pieno*, *0* o *1*. Dato che la differenza di potenziale elettrico non è uguale prima e dopo la sinapsi, si parla di potenziale pre-sinaptico e potenziale post-sinaptico; esso dipende dalle caratteristiche biochimiche della sinapsi. Parlando di reti neurali si dice che la sinapsi *pesa* il potenziale pre-sinaptico inibidendolo oppure eccitandolo. A questo punto i potenziali post-sinaptici provenienti dai vari dendriti si propagano fino al corpo del neurone e si sommano (se il valore della somma supera una certa soglia allora il neurone farà ripropagare la differenza di potenziale attraverso il suo assone).

2.2.1 Il neurone artificiale

Il neurone artificiale viene, all'interno del connessionismo, chiamato *processing unit* (PU). Questa è l'unità costituente più piccola di una rete neurale. Il suo ruolo è a grandi linee quello di simulare ciò che fa un neurone biologico ossia prendere in ingresso tutti i segnali che riceve dagli altri neuroni (o da recettori sensoriali, ad esempio i neuroni della retina), sommarli e trasferire l'output agli altri neuroni ai quali è collegato.

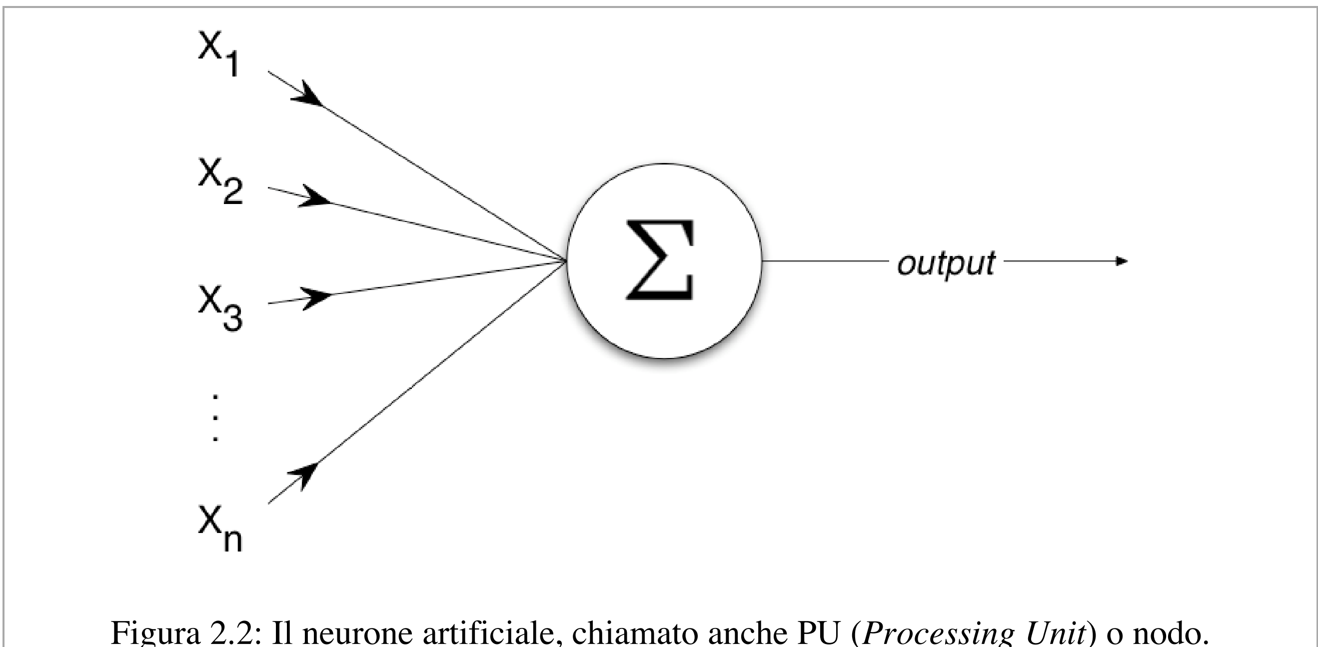


Figura 2.2: Il neurone artificiale, chiamato anche PU (*Processing Unit*) o nodo.

In figura 2.2 possiamo osservare il neurone artificiale. Così come un neurone biologico riceve vari impulsi tramite i dendriti così il neurone artificiale riceve diversi valori in ingresso. Tutti questi valori vengono poi sommati per costituire poi il valore calcolato dalla PU. La somma equivale a:

$$S = x_1 + x_2 + x_3 + \ldots + x_n = \sum_{k=1}^{n} x_k$$

Come visto prima, ogni valore di ingresso è il risultato della modulazione dell'input da parte della sinapsi. Questa proprietà dei sistemi biologici viene ricondotta al concetto matematico di *peso* che solitamente si abbrevia con "*w*" poiché deriva dalla parola inglese *weight*. Il peso di una connessione non è altro che un numero reale (quindi può essere anche negativo e quindi inibitorio) per il quale moltiplicare il valore di input. Così facendo il valore di ingresso potrà avere un effetto positivo o negativo sulla somma globale dei segnali. In questo caso si parla di somma pesata degli input. La struttura del neurone con anche la caratteristica dei pesi è illustrata in figura 2.3.

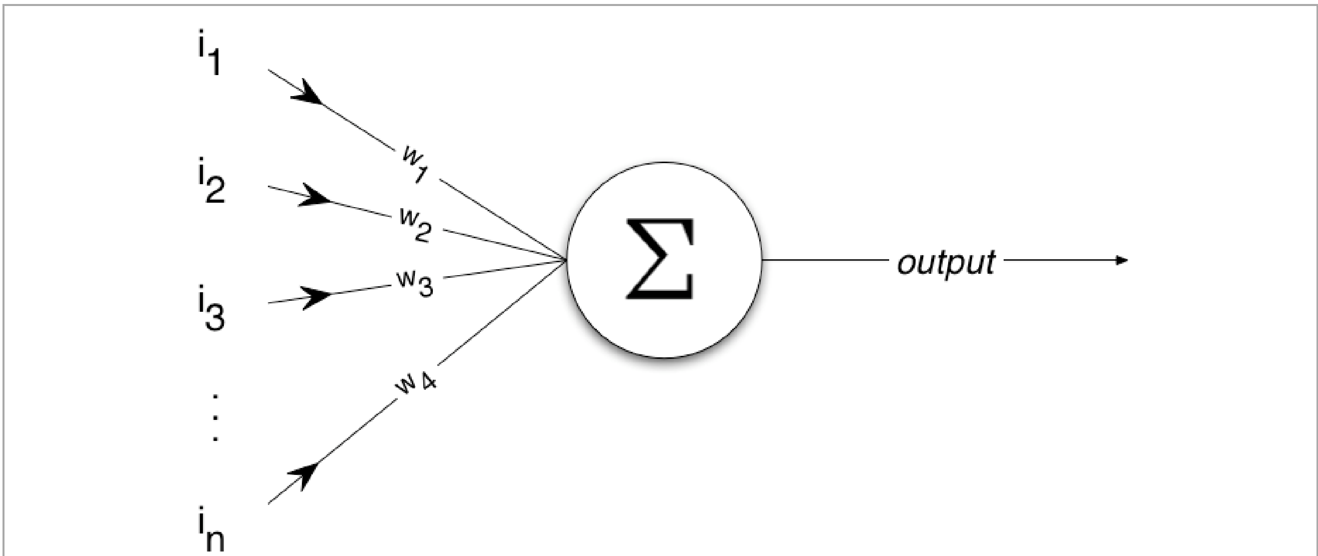


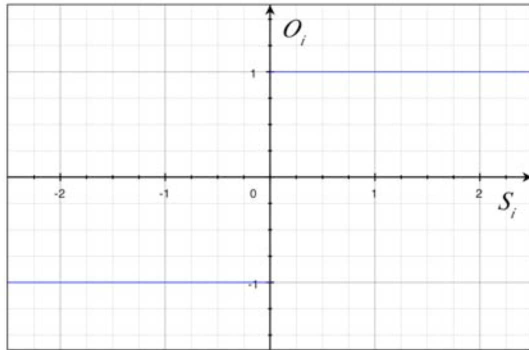
Figura 2.3: I pesi (detti anche sinapsi) *w* determinano quanto il relativo input *i* influenza la somma finale. Se il peso è positivo esso sarà eccitatorio, mentre se è negativo sarà inibitorio.

La somma pesata viene definita in questo modo:

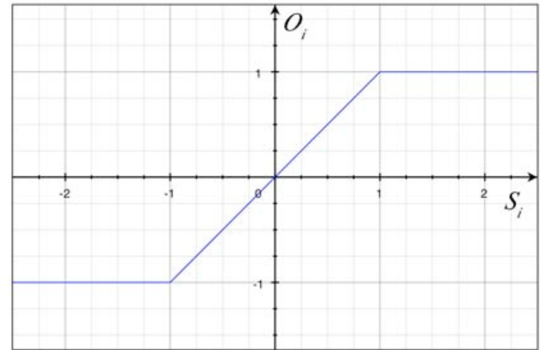
$$S_{pesata} = i_1 \cdot w_1 + i_2 \cdot w_2 + \ldots + i_n \cdot w_n = \sum_{k=1}^{n} i_k \cdot w_k$$

Oltre alla somma pesata viene ora introdotta un'altra proprietà del neurone artificiale ispirata ancora una volta ad una proprietà del neurone biologico. Prima è stato mostrato come il neurone biologico, a livello del soma, somma tutti i potenziali post-sinaptici dei dendriti. Ma in realtà questa somma non è proprio la somma algebrica dei valori entranti; esistono infatti diversi fattori che fanno in modo che il risultato non sia esattamente la somma algebrica. Di solito si tratta di una funzione non lineare. Più semplicemente il neurone artificiale somma gli ingressi pesati e poi modifica il risultato in base ad una determinata funzione *f*. Questa funzione è detta *funzione di trasferimento*.

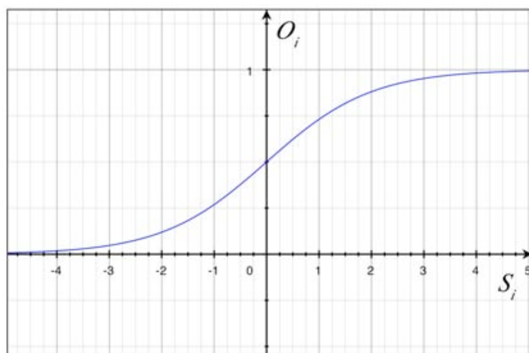
Nella figura 2.4 sono illustrati quattro tipi di queste funzioni che sono spiegate in seguito.



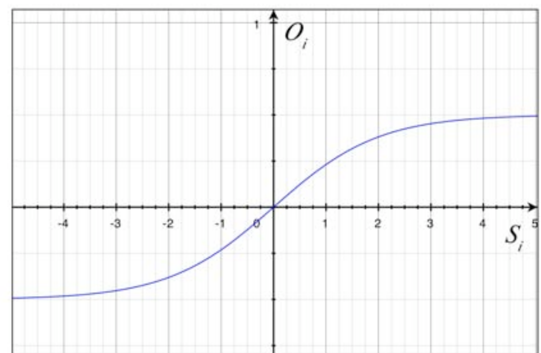
a)



b)



c)



d)

Figura 2.4: Funzioni di trasferimento fra le più usate: a) funzione gradino b) funzione lineare con saturazione c) funzione sigmoide con valori computati compresi fra 0 e 1 d) funzione sigmoide con valori computati compresi fra -1/2 e 1/2

a) *funzione a gradino*: se l'argomento è maggiore o uguale a 0 il valore restituito è 1, altrimenti se esso è negativo viene restituito -1. Questa funzione può essere chiamata anche *funzione segno* o in modo abbreviato *sgn(x)*. Formalmente:

$$\operatorname{sgn}(x) = \begin{cases} 1 & \text{se } x > 0 \\ -1 & \text{se } x < 0 \end{cases}$$

b) *funzione lineare con saturazione*: se il valore di x è compreso tra -1 e 1, la funzione è lineare e restituisce il valore di x. Mentre invece la funzione si appiattisce per i valori x<-1 o per x>1. Formalmente:

$$f(x) = \begin{cases} -1 & \text{se } x < 0 \\ x & \text{se } -1 \le x \le 1 \\ 1 & \text{se } x > 1 \end{cases}$$

c) *funzione sigmoide*: se il valore di x è 0 il valore restituito è 1/2. Con l'aumentare del valore di x la funzione tende asintoticamente ad 1, mentre col diminuire di x essa tende asintoticamente a 0. Formalmente:

$$f(x) = \frac{1}{1 + e^{-\beta x}}$$

dove β è un coefficiente, spesso uguale a 1, che determina con quale velocità la funzione, in prossimità di x=0, tende a crescere fino a 1/2.

d) questa è solamente una piccola variazione della funzione sigmoide in quanto essa ha uno zero in x=0. Semplicemente è traslata di 1/2 verticalmente:

$$f(x) = \frac{1}{1 + e^{-\beta x}} - \frac{1}{2}$$

Essa è una delle più usate poiché permette anche valori negativi e soprattutto poiché è derivabile.

In figura 2.5 è rappresentato un neurone artificiale con tutte le sue caratteristiche.

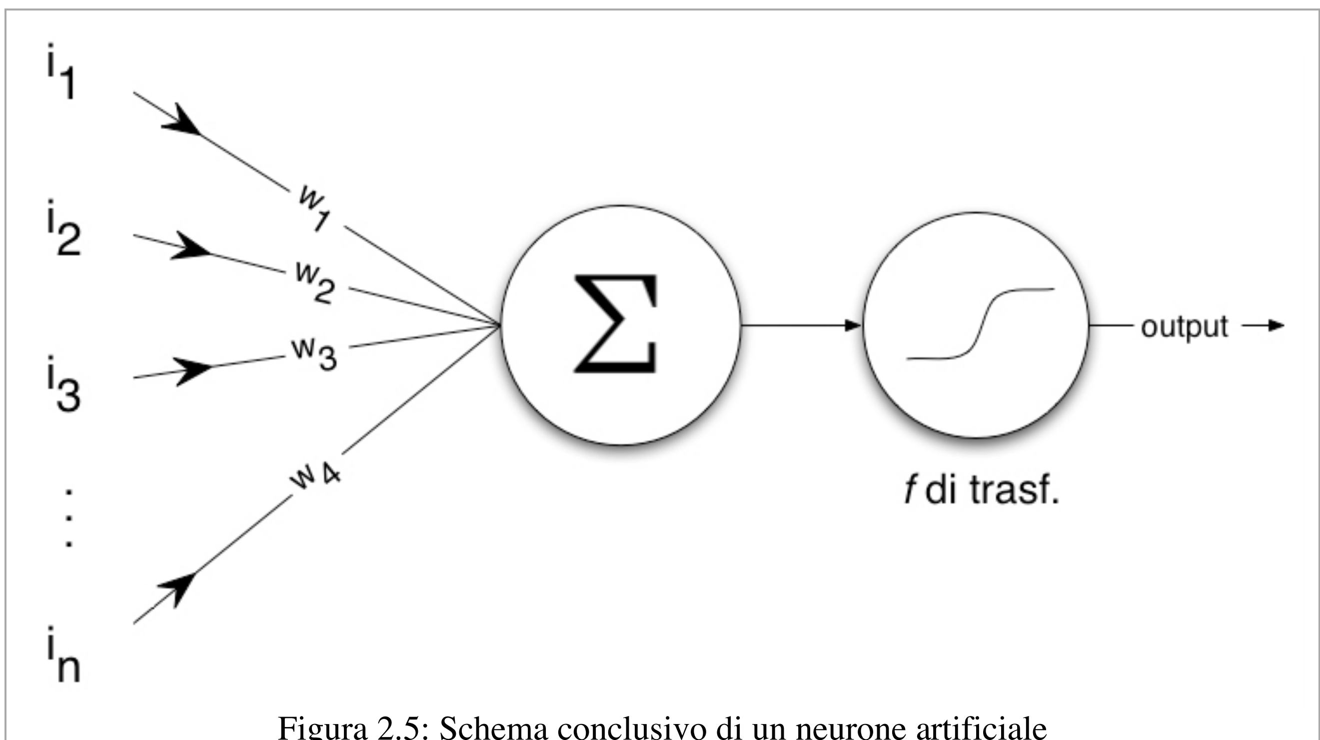


Figura 2.5: Schema conclusivo di un neurone artificiale

Di seguito è riportato il codice in sintassi *BASIC* (listato 1) e *C/C++* (listato 2) per calcolare l'output di un neurone a partire da un'array di valori che corrispondono agli input e da un'altra composta dai valori dei pesi. In questo caso la funzione *Test()* ritornerà circa 8.02.

```
‘ Dichiaro due array da 3 elementi
Dim inputs(2), weights(2) As Double

‘ Definisco la funzione sigmoide che prende come parametri la x e β
Function Sigmoid(x As Double, B As Double) As Double
  Sigmoid = 1 / (1 + Exp(-B * x))
End Function

Function ComputeOutput() As Double
  Dim i As Integer
  Dim somma As Double

  ‘ Con un ciclo iterativo aggiungo il prodotto dell’input con
  ‘ il peso alla variabile della somma totale
  For i = 0 To UBound(inputs)
    somma = somma + inputs(i) * weights(i)
  Next

  ComputeOutput = Sigmoid(SommaTotale, 1)
End Function

Function Test()
  ‘ Imposto dei valori casuali per poter provare l’algoritmo
  inputs(0) = 0.5
  inputs(1) = 0.4
  inputs(2) = -0.8

  weights(0) = 1.02
  weights(1) = -0.94
  weights(2) = 0.001

  MsgBox “L’output del neurone e’:” & ComputeOutput()
End Function
```

Listato 1: Codice in *BASIC* per il calcolo dell’output del neurone.

```c
#include <stdio.h>
#include <math.h>

// Dichiaro due array da 3 elementi e imposto dei valori casuali per poter
// provare l’algoritmo
double inputs[] = {0.5, 0.4, -0.8}, weights[] = {1.02, -0.94, 0.001};

// Definisco la funzione sigmoide che prende come parametri x e β
double Sigmoid(double x, double B) {
  return 1 / (1 + exp(-B * x));
}

double ComputeOutput(void) {
  int i;
  double somma = 0;

  // Con un ciclo iterativo aggiungo il prodotto dell’input con
  // il peso alla variabile della soma totale
  for(i=0;inputs[i]!=NULL;i++) {
    somma += inputs[i] * weights[i];
  }

  return Sigmoid(somma, 1);
}

void Test(void)
  printf("L'output del neurone e': %f\n", ComputeOutput());
```

Listato 2: Codice in *C*/*C*++ per il calcolo dell’output del neurone.

2.2.2 L’architettura di una rete neurale

Una rete neurale è un insieme di nodi, o neuroni artificiali, connessi fra di loro con collegamenti chiamati sinapsi che rappresentano i pesi (figura 2.6). Esisono svariati modi di connessione e questo determina appunto la struttura o architettura delle rete neurale che non ha solo conseguenze pratiche, in quanto la capacità computazionale della rete cambia, ma anche teoriche, che coinvolgono il concetto di apprendimento.

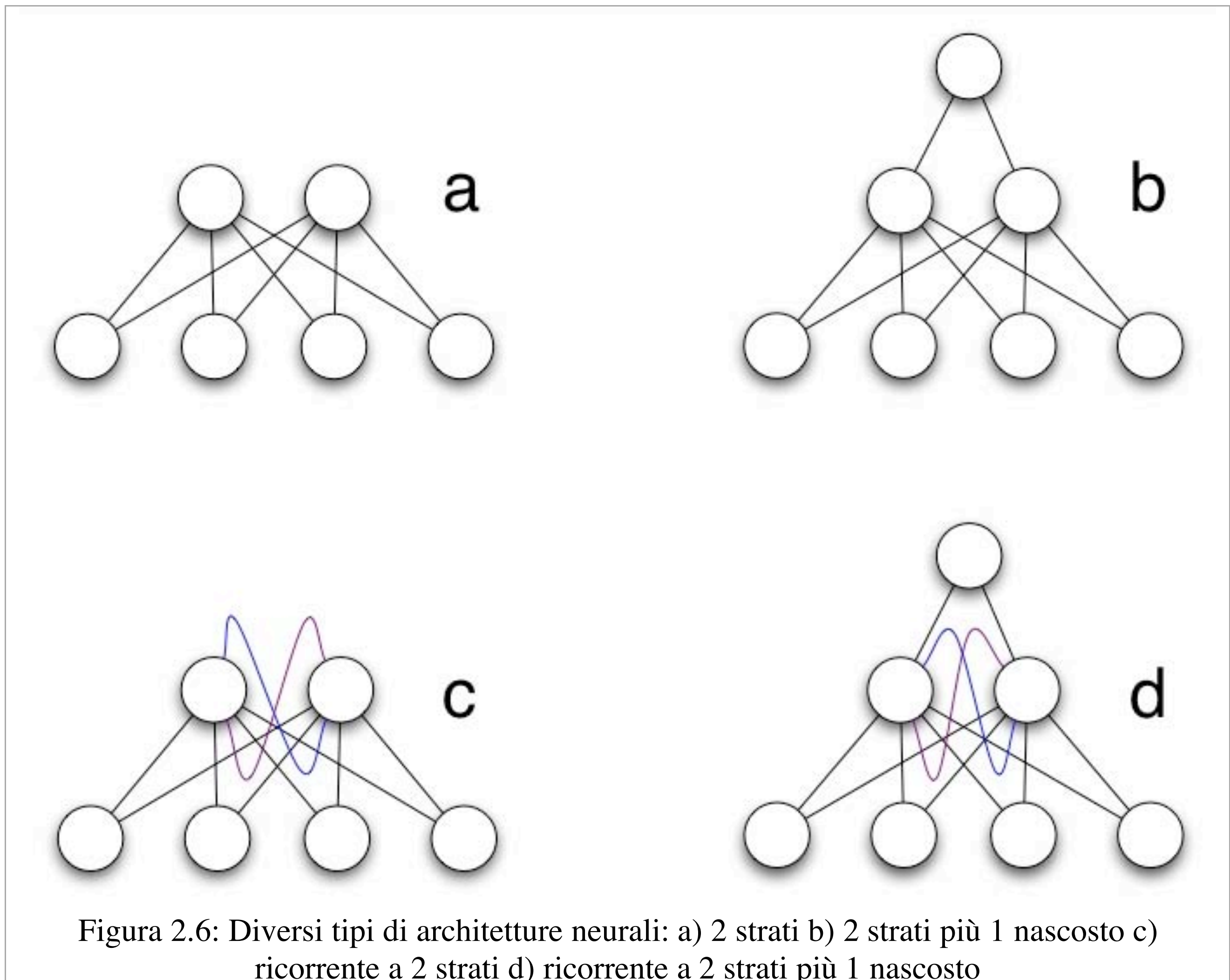


Figura 2.6: Diversi tipi di architetture neurali: a) 2 strati b) 2 strati più 1 nascosto c) ricorrente a 2 strati d) ricorrente a 2 strati più 1 nascosto

In generale una rete neurale è composta da neuroni di input e neuroni di output ed eventualmente da neuroni nascosti. Quando i neuroni di input ricevono dei segnali essi li propagano fino ad arrivare ai neuroni d’uscita.

È possibile anche aggiungere delle connessioni di *feedback* a qualsiasi livello della rete. Ogni neurone può infatti trasmettere un segnale a neuroni di un livello precedente, dello stesso strato oppure a sé stesso. Questo tipo di reti sono di solito chiamate *reti neurali ricorrenti*. In questi casi il funzionamento della rete diventa un po’ più complesso, poiché lo stato di ogni singolo neurone non può essere determinato solamente conoscendo i segnali in entrata, ma occorre conoscere i valori della rete agli istanti precedenti. Per questo quando si lavora con reti che utilizzano connessioni di *feedback* è comodo tenere una copia dei valori dei neuroni allo stato precedente.

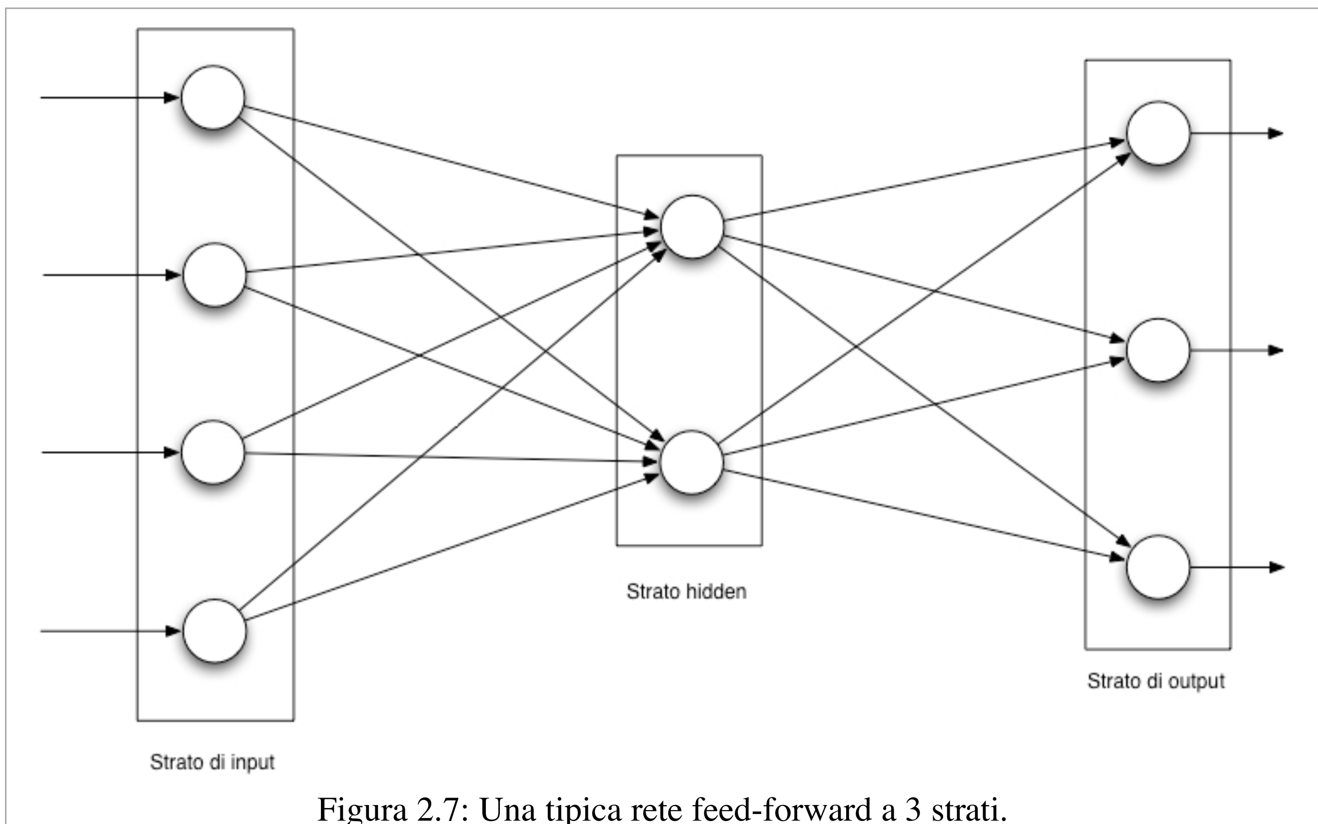


Figura 2.7: Una tipica rete feed-forward a 3 strati.

Le architetture si distinguono principalmente in funzione del numero di *strati* di connessioni sinaptiche, attraverso i quali, il segnale d'ingresso viene propagato per giungere ai neuroni di uscita. Questo tipo di rete è detta *feed-forward network* ed è la più usata.

2.2.3 Codice d'esempio

Di seguito è riportato il codice in sintassi *BASIC* (listato 3) e *C/C++* (listato 4) per calcolare gli output di una rete neurale di tipo *feed-forward* che possiede 3 strati, formati rispettivamente da 4, 2 e 3 neuroni (figura 2.7). La funzione *Test()* ritornerà rispettivamente circa 0.441499; 0.336476; 0.240680 per il neurone numero 1, 2 e 3.

```
' Definisco le array che rappresentano rispettivamente i valori degli
' input, dei neuroni dello strato "nascosto" e di uscita
Dim inputs(3), hidden(1), outputs(2) As Double

' Definisco l'array bidimensionale che contiene
' i valori delle sinapsi inputs-hidden
Dim w_ih(3, 1) As Double

' Definisco l'array bidimensionale che contiene
' i valori delle sinapsi hidden-outputs
Dim w_ho(1, 2) As Double

Function Sigmoid(x As Double, B As Double)
  Sigmoid = 1 / (1 + Exp(-B * x))
End Function
```

```
Function Test()
  Dim i, k As Integer
  Dim somma As Double
  Dim messaggio As String

  ' Inizializzo casualmente le array
  inputs(0) = 0.45
  inputs(1) = 0.11
  inputs(2) = -0.99
  inputs(3) = -0.232
  w_ih(0, 0) = 0.22
  w_ih(1, 0) = 0.456
  w_ih(2, 0) = -0.23
  w_ih(3, 0) = 0.957
  w_ih(0, 1) = -0.255
  w_ih(1, 1) = 0.029
  w_ih(2, 1) = -0.8
  w_ih(3, 1) = -0.91
  w_ho(0, 0) = 0.333
  w_ho(0, 1) = 0.411
  w_ho(0, 2) = -0.25
  w_ho(1, 0) = -0.56
  w_ho(1, 1) = -0.899
  w_ho(1, 2) = -0.453

  For i = 0 To 1
    ' Calcolo il valore dell'i-esimo neurone dello strato hidden
    For k = 0 To 3
      somma = somma + inputs(k) * w_ih(k, i)
    Next
    hidden(i) = Sigmoid(somma, 1)
  Next

  somma = 0
  For i = 0 To 2
    ' Calcolo il valore dell'i-esimo neurone dello strato di output
    For k = 0 To 1
      somma = somma + hidden(k) * w_ho(k, i)
    Next
    outputs(i) = Sigmoid(somma, 1)
  Next

  ' Stampo i valori dei neuroni dello strato di output
  For i = 0 To 2
    MsgBox "Valore del neurone di output numero " & (i + 1) & ": " ¬
                                                            & outputs(i)

  Next
End Function
```

Listato 3: Codice in *BASIC* per il calcolo degli output della rete neurale.

```c
#include <stdio.h>
#include <math.h>

// Definisco le array che rappresentano rispettivamente i valori degli
// input, dei neuroni dello strato "nascosto" e di uscita
double inputs[] = {0.45, 0.11, -0.99, -0.232}, hidden[2], outputs[3];

// Definisco e inizializzo casualmente l'array bidimensionale che contiene
// i valori delle sinapsi inputs-hidden
double w_ih[4][2] = {{0.22, -0.255},
                     {0.456, 0.029},
                     {-0.23, -0.80},
                     {0.957, -0.91}};

// Definisco e inizializzo casualmente l'array bidimensionale che contiene
// i valori delle sinapsi hidden-outputs
double w_ho[2][3] = {{0.333, 0.411, -0.25},
                     {-0.56,-0.899,-0.453}};

double Sigmoid(double x, double B) {
  return 1 / (1 + exp(-B * x));
}

void test(void) {
  int i, k;
  double somma = 0;

  for(i=0; i<2; i++) {
    // Calcolo il valore dell'i-esimo neurone dello strato hidden
    for(k=0; k<4; k++) somma += inputs[k] * w_ih[k][i];
    hidden[i] = Sigmoid(somma, 1);
  }

  for(i=0, somma=0; i<3; i++) {
    // Calcolo il valore dell'i-esimo neurone dello strato di output
    for(k=0; k<2; k++) somma += hidden[k] * w_ho[k][i];
    outputs[i] = Sigmoid(somma, 1);
  }

  // Stampo i valori dei neuroni dello strato di output
  for(i=0;i<3;i++)
    printf("Valore del neurone di output numero %d: %f\n", i+1, outputs[i]);
}
```

Listato 4: Codice in *C/C++* per il calcolo degli output della rete neurale.

3. Gli algoritmi genetici

3.1 Introduzione

Gli algoritmi genetici, spesso chiamati anche in forma abbreviata GA (*Genetic Algorithms*), fanno parte dell'intelligenza artificiale. In particolare sono dei metodi di adattamento che sono usati per risolvere problemi molto complessi o di ottimizzazione. Il principio su cui poggiano sono i processi genetici che avvengono negli organismi biologici. Immaginiamo di avere un gruppo iniziale di organismi, dopo molte generazioni essi si evolveranno secondo i principi della selezione naturale e della sopravvivenza del più forte, come teorizzò per la prima volta Charles Darwin nella sua opera "L'origine delle specie". Imitando questo modello biologico gli algoritmi genetici possono evolvere delle soluzioni a problemi del mondo reale molto complessi, se sono stati codificati opportunamente. Ad esempio possono venire implementati per l'ottimizzazione di funzioni numeriche, per l'*image processing* (ad esempio per l'elaborazione di immagini mediche a raggi X o da satellite), per l'ottimizzazione combinatoria (ad esempio lo studio ed ottimizzazione del traffico in una città), nel *bin packing* (cioè nel disporre in modo ottimale un numero di oggetti su uno spazio limitato, usato tantissimo nel campo dell'industria ad esempio per creare il layout di circuiti integrati), per analisi finanziarie e per classificare in automatico dati medici, transazioni commerciali, immagini. Essi, insomma, sono presenti nella nostra vita quotidiana più di quanto si possa pensare.

In natura, gli individui di una popolazione competono tra di loro per risorse naturali come il cibo, l'acqua, il territorio. Inoltre spesso competono per attrarre un compagno o una compagna. Gli individui che hanno maggiore successo nella sopravvivenza e nella riproduzione avranno un numero relativamente grande di discendenti. A questo punto la prole riceverà i geni dei migliori individui (*fit individuals*). In questo modo, con la combinazioni delle caratteristiche migliori, le nuove popolazioni diventano sempre più adatte all'ambiente in cui vivono. Quest'analogia con il comportamento della natura viene sfruttata in questo modo: la popolazione di individui rappresenta diverse possibili soluzioni al problema. A ogni individuo è associato un punteggio di adattamento (*fitness score* o più semplicemente *fitness*) a seconda di quanto la soluzione al problema sia buona. Esso viene calcolato secondo una funzione di *fitness* che viene definita tenendo in considerazione di vari fattori. Ad esempio se si intendesse calcolare la retta affine che approssima meglio un insieme di punti, la funzione di *fitness* potrebbe venir definita come la somma degli errori di approssimazione al quadrato:

$$\Phi = \left\{ \sum_{i=0}^{n} \left[\left(ax_i + b \right) - y_i \right]^2 \right\}^{-1}$$

Dove $ax_i + b$ rappresenta l'ordinata della funzione affine, y_i l'ordinata del punto i-esimo e n il numero di punti. È da notare che la sommatoria è stata elevata alla -1 in modo da rendere coerente il risultato con la definizione di *fitness*. Infatti più la sommatoria è bassa, e quindi l'errore piccolo, più il valore del *fitness* aumenta.

Gli individui con un *fitness* minore hanno anche meno possibilità di riprodursi e quindi si estingueranno. Infine, se la funzione di *fitness* è stata costruita bene, l'algoritmo convergerà verso la soluzione.

Quello su cui un algoritmo genetico opera è quindi il genotipo di una poplazione. Il genotipo è una codifica dei parametri degli individui (fenotipo), il genotipo di un singolo individuo è detto cromosoma. Questo può essere composto da uno o più geni, ciascuno rappresentante la codifica di un particolare parametro.

Molta della ricerca sugli algoritmi genetici è concentrata su regole empiriche per il miglior funzionamento. Come visto prima le applicazioni sono moltissime, tuttavia non esiste una teoria accettata o una dimostrazione matematica che spieghi esattamente perché essi funzionino e abbiano certe proprietà, cosa che comunque non intendo trattare.

È interessante anche il fatto che spesso gli algoritmi genetici possono essere utilizzati anche in modo "ibrido", ad esempio con altri metodi di intelligenza artificiale come le reti neurali[1].

3.2 L'algoritmo

Innanzi tutto si assume che la soluzione di un problema sia scomponibile in un set di parametri che formano il cromosoma. Spesso viene usato un alfabeto binario. Ad esempio se volessimo codificare il valore delle variabili a e b che costituiscono i parametri di una funzione affine che tenta di approssimare al meglio dei punti, possiamo rappresentare ogni variabile con un numero binario di 8 bit. Il cromosoma conterrà quindi due geni e sarà lungo 16 bit.

In seguito saranno descritti i vari processi che avvengono durante l'esecuzione dell'algoritmo.

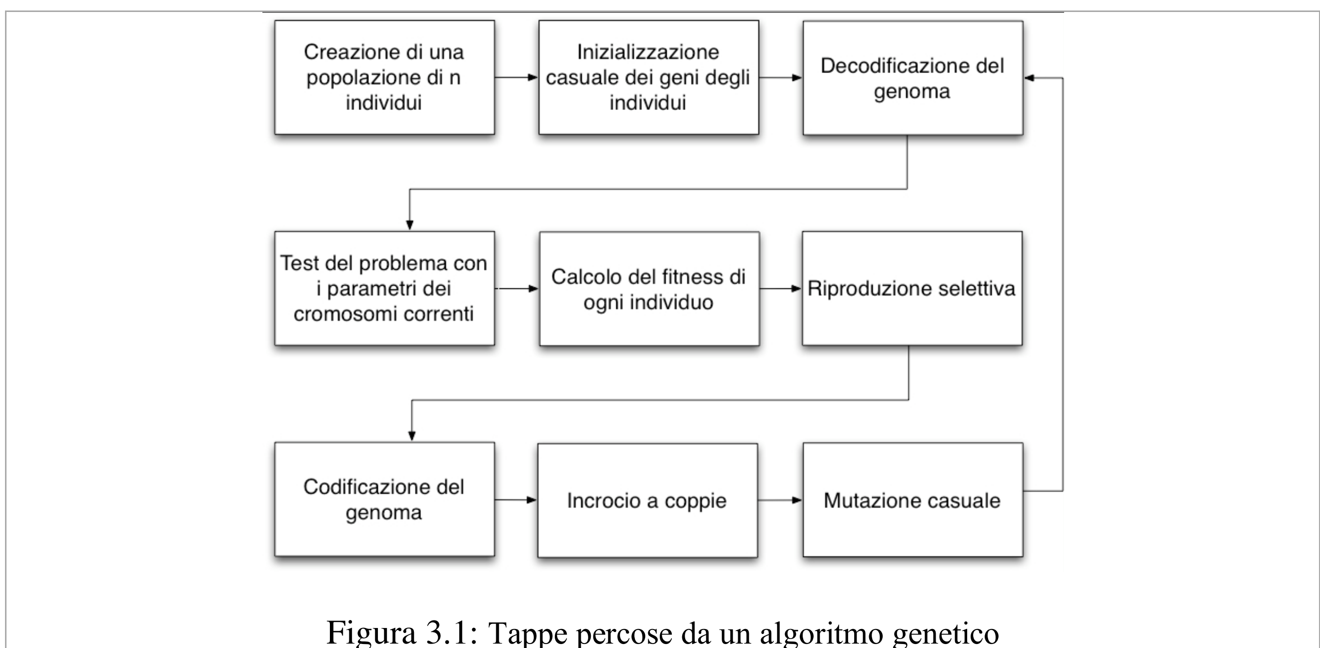


Figura 3.1: Tappe percorse da un algoritmo genetico

[1] Questo approccio sarà trattato nel capitolo 4 "Vita artificiale".

3.2.1 Creazione di una popolazione

Il primo passo dopo la codifica del problema è appunto la creazione di una popolazione. Questo consiste nel creare un certo numero di individui e quindi cromosomi. Il numero di individui non è fisso e non esiste neanche una regola empirica per determinarlo. Nel maggiore dei casi si utilizzano da 30 a 50 individui, ma ribadisco che il numero migliore dipende anche dal tipo di problema. Da una parte, avendo un numero di individui molto basso, si potrebbero escludere le possibilità di esplorare lo spazio delle soluzioni restringendo il campo a soluzioni non ottime. D'altra parte, avendo un numero elevato, si potrebbe rischiare di non avere una convergenza alla soluzione da parte dell'algoritmo dato che lo spazio di esplorazione è troppo vasto.

3.2.2 Inizializzazione casuale dei geni

Dopo aver creato la stringa di bit, o di valori numerici essi vengono generati casualmente in modo da creare potenziali soluzioni casuali.

3.2.3 Decodificazione del genoma

Questa è la fase in cui il genotipo viene usato per creare il fenotipo. In altre parole il genoma viene letto e i geni si trasformano nelle caratteristiche dell'individuo. Per fare un esempio concreto possiamo immaginare il problema della costruzione di un profilo alare che abbia una certa portanza.

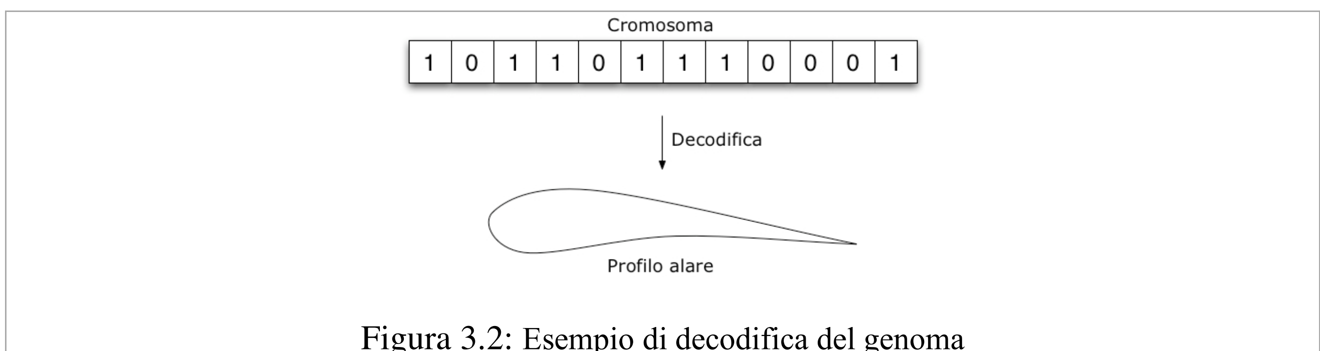


Figura 3.2: Esempio di decodifica del genoma

Il cromosoma contiene solamente dei valori numerici. L'operazione di decodificazione consiste nell'interpretare questi valori come le caratteristiche del profilo alare (figura 3.2).

3.2.4 Test del problema e calcolo del fitness dell'individuo

Tutti gli individui vengono testati, ossia ne viene calcolato il *fitness*. Sempre riferendoci al problema del profilo alare si potrebbe pensare di calcolarlo mediante la somma dei vettori di forza che agiscono sull'oggetto e prendere poi la componente *y* del vettore in modo da ottenere la forza risultante verso l'alto che determina la portata. Un'altra possibilità potrebbe essere quella di costruire una simulazione virtuale di una galleria del vento e lasciare il profilo alare in essa per qualche secondo. Ad ogni passo verrebbe calcolato il fitness. La somma di tutti i valori verrebbe poi divisa per il numero di passi, ottenendo così una media.

3.2.5 Riproduzione selettiva

La riproduzione selettiva consiste nel creare un numero di copie di ciascun cromosoma proporzionale al valore del suo *fitness* e nell'eliminare i cromosomi peggiori. Per far questo si favoriscono gli individui che si sono rivelati migliori durante la loro vita dandogli una probabilità maggiore per contribuire con uno o più figli alla generazione successiva, imitando così il criterio darwiniano di sopravvivenza del più forte. L'insieme degli individui che si riprodurranno è chiamato *mating pool*, "piscina di accoppiamento". Esistono svariati criteri per selezionare gli individui adatti, da trasferire nella *mating pool*.

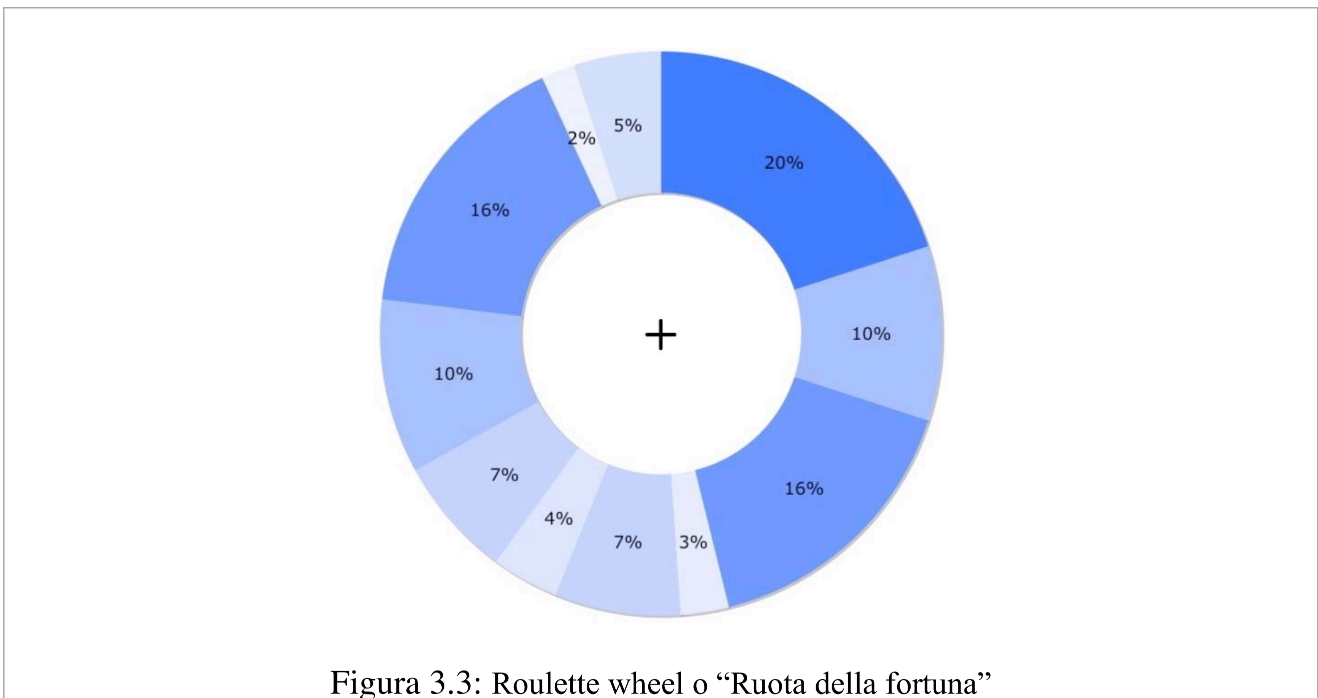


Figura 3.3: Roulette wheel o "Ruota della fortuna"

Uno dei più usati è la *roulette wheel selection*, che può essere tradotto come "ruota della fortuna". Le caselle della ruota sono tante quanti sono gli individui della popolazione ma evidentemente essa è truccata perché la larghezza di ciascuna casella è proporzionale alle prestazioni dell'individuo. Per creare una nuova popolazione di individui che abbia lo stesso numero di quelli di prima basta far girare n volte la ruota, generando ogni volta una copia del cromosoma corrispondente della casella che viene selezionata.

3.2.6 Incrocio a coppie

Dopo aver selezionato gli individui da trasferire nella *mating pool* si ricodifica l'informazione genetica per prepararsi alla manipolazione dei cromosomi. In realtà questa fase, nella pratica, non viene fatta poiché solitamente si tiene sempre la copia del materiale genetico in un array.

L'incrocio consiste nel disporre casualmente i cromosomi in coppie e farli incrociare con una certa probabilità. Più precisamente avviene uno scambio di geni tra gli individui di ciascuna coppia con una certa probabilità. Questa fase è chiamata *crossover*, dal nome del processo biologo che avviene durante la meiosi, durante il quale una cellula diploide si

divide in 4 cellule aploidi. Infatti, nella profase i cromosomi si appaiano commettendo anche l'errore (voluto!) di sovrapporre alcuni dei segmenti dei cromosomi scambiandosi del materiale genetico. In questo modo viene determinata una nuova distribuzione dell'informazione genetica, contribuendo così ad un aumento della variabilità genetica.

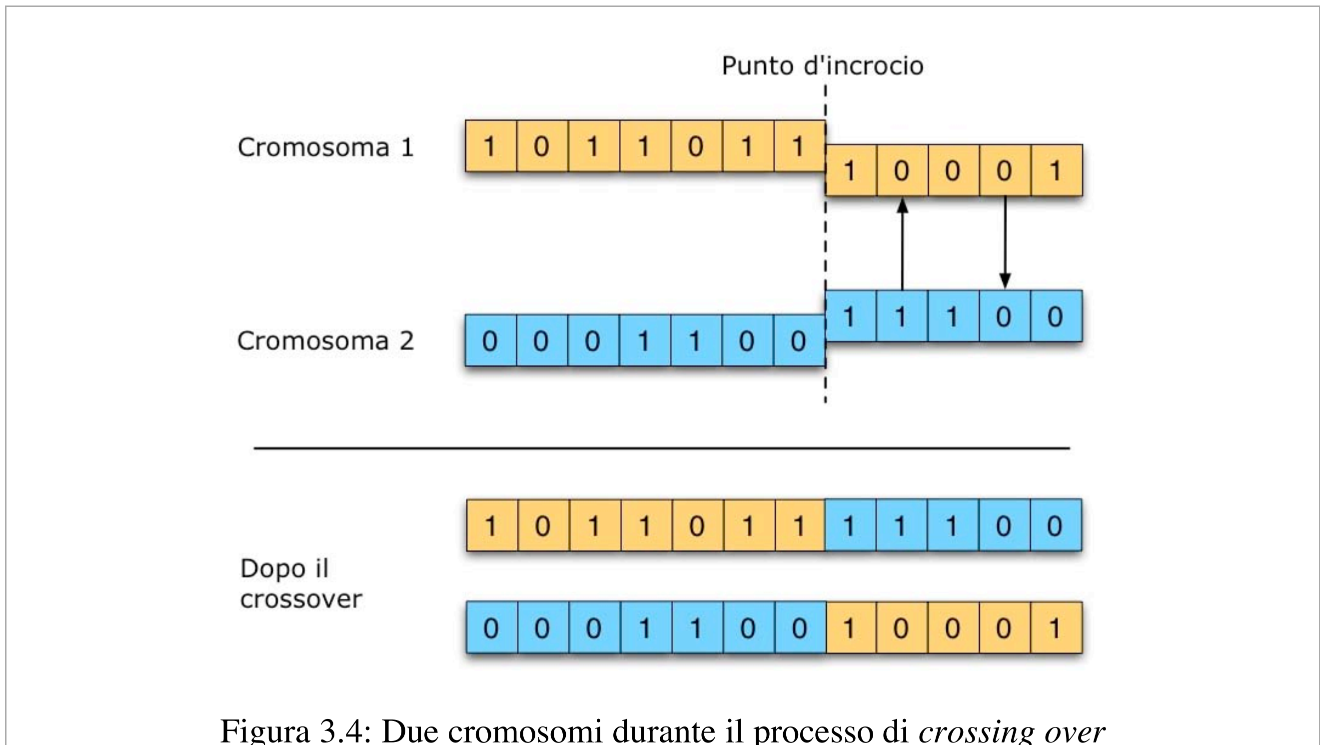


Figura 3.4: Due cromosomi durante il processo di *crossing over*

Anche per questa operazione esistono svariate tecniche, ma quella più nota (e aggiungerei più simile al sistema biologico) consiste nell'allineare i due cromosomi, scegliere uno o più punti d'incroci casuali e scambiare quindi la porzione del materiale genetico rispetto al punto d'incrocio (figura 3.4).

3.2.7 Mutazione

Infine, ciascun gene dei cromosomi della nuova popolazione può subire anche una mutazione con una certa probabilità. Ciò viene fatto per imitare al meglio il modello biologico e per dare nuove possibilità di esplorazione della soluzione. Spesso la mutazione consiste nell'invertire il valore di un bit quando viene usata la codifica binaria. Nel caso invece che ciascun gene sia un numero reale, si può sostituire ad esso un numero casuale oppure aggiungere un valore, anch'esso casuale (che ovviamente potrebbe essere anche negativo).

Dopo la mutazione la nuova generazione avrà una prestazione uguale o anche inferiore alle prestazioni medie della generazione precedente, tuttavia, alcuni individui avranno prestazioni maggiori e quindi una probabilità di riprodursi maggiore. Le prestazioni medie della generazione successiva tenderanno quindi ad essere migliori nel tempo.

3.3 Codice d'implementazione

In questa sezione tratterò un esempio di implementazione degli algoritmi genetici. Tra i vari problemi classici che vengono risolti mediante gli algoritmi genetici ho voluto scegliere quello del commesso viaggiatore, chiamato comunemente TSP (*Traveling Salesman Problem*). Prima di tutto perché mostra molto bene come la routine di un algoritmo genetico non è generale ma nella pratica deve venire adattata al problema e secondariamente perché le sue varianti si trovano nella vita quotidiana, come nel disegno di linee telefoniche e circuiti integrati. Il TSP (figura 3.5) riguarda un commesso viaggiatore che deve effettuare delle consegne a dei negozi distribuiti in varie città. Il problema sta nel trovare il percorso più economico, cioè quel percorso che passa attraverso ciascuna città (nodo), la cui distanza totale percorsa sia minima. Ovviamente il viaggiatore non può visitare una città più di una volta. Questo problema non è da sottovalutare: anche con computer molto potenti, usando degli algoritmi lineari standard il tempo di risoluzione diventa troppo alto. Basti pensare che con solo 50 città si hanno già *50!* possibilità (circa $3.0 \cdot 10^{64}$).

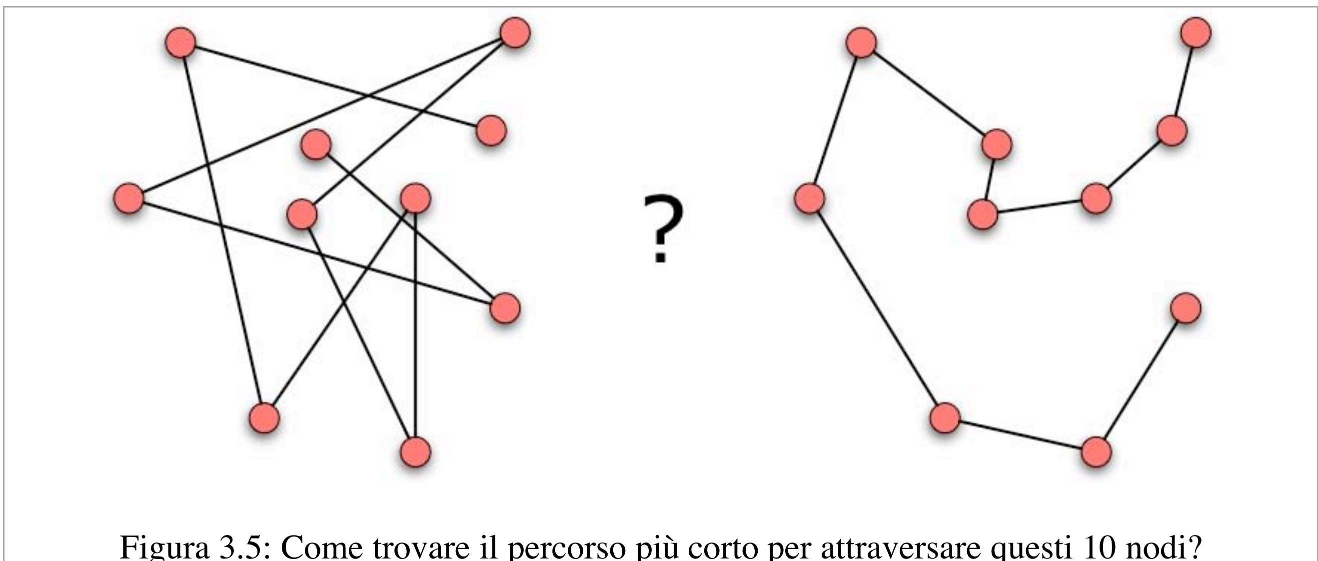


Figura 3.5: Come trovare il percorso più corto per attraversare questi 10 nodi?

Cercando di impostare il problema ci si rende subito conto che la rappresentazione binaria di una città non può funzionare. Questo perché la mutazione potrebbe dar luogo ad un individuo che non rappresenta un percorso valido, infatti si potrebbe ottenere la stessa città due volte nello stesso cromosoma. Inoltre per costruire un cromosoma binario per codificare, ad esempio, 10 città occorrerebbero sequenze di 4 bits, ma alcune di queste corrisponderanno a nessuna città. Ad esempio 1101 corrisponde a 13, ma le città sono solo 10. Problemi analoghi sorgono anche applicando il *crossover*. Tutto questo fa pensare che il modo migliore è quello di rappresentare il cromosoma come un vettore di numeri interi. Quindi per definire il percorso per andare dalla città numero 0 alla città numero 9 si può ad esempio scrivere:

$$\vec{v} = [0 \quad 3 \quad 7 \quad 5 \quad 6 \quad 8 \quad 2 \quad 1 \quad 4 \quad 9]$$

Per rendere più intuitiva la soluzione del problema supponiamo di avere 10 città disposte su una circonferenza. Sarà quindi ovvio che il percorso migliore, più corto, è quello che forma un decagono regolare (figura 3.6).

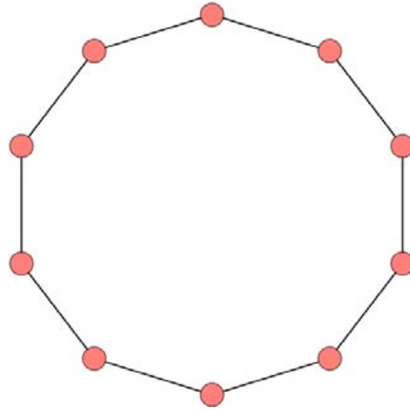


Figura 3.6: 10 nodi disposti su una circonferenza in modo da formare un decagono regolare

3.3.1 La routine principale

I prossimi sotto capitoli saranno dedicati a mostrare e spiegare il codice relativo alla risoluzione del problema sopra indicato. Si procederà partendo dalla routine più esterna per poi esaminare le sotto funzioni, affrontando anche, man mano che si presentano, le problematiche di implementazione. Nel listato 1 è riportata la prima porzione di codice.

```
Const PopSize = 49
Const CrossoverProbability = 0.80
Const MutationProbability = 0.20

Dim Genome(PopSize) As String
Dim Points(9,1), Fitness(PopSize) As Double

Sub Run()
  Dim Length, BestPathLength As Double
  Dim i As Integer

  Points(0,0)=0
  Points(1,0)=5.878
  Points(2,0)=9.511
  Points(3,0)=9.511
  Points(4,0)=5.878
  Points(5,0)=0
  Points(6,0)=-5.878
  Points(7,0)=-9.511
  Points(8,0)=-9.511
  Points(9,0)=-5.878

  Points(0,1)=10
  Points(1,1)=8.09
  Points(2,1)=3.09
  Points(3,1)=-3.09
  Points(4,1)=-8.09
  Points(5,1)=-10
  Points(6,1)=-8.09
```

```
    Points(7,1)=-3.09
    Points(8,1)=3.09
    Points(9,1)=8.09

    GenerateRandomPopulation()

    // Imposto la lunghezza del miglior percorso ad un valore
    // sicuramente più alto di tutti.
    BestPathLength=200

    Do
      For i=0 To PopSize
        Length = ComputePathLength(i)
        Fitness(i) = - Length + 200
        BestPathLength=Min(BestPathLength, Length)
        If BestPathLength < 60 Then
          GoTo SolutionFound
        End if
      Next

      SelectiveReproduction()

      Genome.Shuffle

      DoCrossOver()

      DoMutation()
    Loop

    SolutionFound:

    Return Genome(i)
  End Sub
```

Listato 1: Codice *BASIC* della funzione principale.

Nella parte esterna della funzione vengono definite alcune costanti, quali la grandezza della popolazione e la probabilità che avvenga una mutazione o un *crossover*. È da notare che *PopSize* contiene in realtà il numero degli individui meno uno. Dato che le array posseggono anche l'elemento 0, è quindi più comodo definire la costante in quel modo. Le altre due costanti verranno invece riprese nella descrizione della funzione di mutazione e *crossover*. Vengono poi definite altre variabili globalmente. Si nota che il genoma di tutti gli individui è contenuto nell'array di stringhe *Genome*. Perciò ad esempio il genoma dell'*i-esimo* individuo potrebbe essere *"4930215786"*, e verrebbe richiamato con l'istruzione *Genome(i)*. La variabile *Points* e *Fitness* contengono rispettivamente le coordinate dei punti e i valori dei *fitness* degli individui.

Nella funzione *Run* sono stati definiti i 10 punti appartenenti alla circonferenza. Per calcolarli è stato sufficiente utilizzare la funzione parametrica seguente e far variare t di un delta di $(2\pi) / 10$. È stato scelto un raggio di 10 unità.

$$\begin{cases} x(t) = r \cdot \sin(t) \\ y(t) = r \cdot \cos(t) \end{cases}$$

La prima sottofunzione che incontriamo è *GenerateRandomPopulation* che si occuperà di generare a caso i geni dei cromosomi. Il ciclo principale è quello racchiuso tra *Do* e *Loop*. La prima parte composta da un altro ciclo si occupa di calcolare il *fitness* degli individui calcolando la lunghezza del percorso. È da notare che per rendere il valore del *fitness* coerente con la sua definizione, la lunghezza è stata cambiata di segno (per ribaltare la funzione rispetto all'asse y) e vi si è aggiunto 200 dato che intuitivamente il percorso più lungo sarà sicuramente minore di $2r \cdot n$, dove n è il numero di punti. Il ciclo si fermerà quando il percorso migliore sarà inferiore a 60, ossia quando avrà trovato la soluzione. La funzione ritornerà quindi la stringa del genoma migliore. In seguito vengono selezionati gli individui migliori che compongono la nuova popolazione, viene mescolata casualmente l'array del genoma e vengono applicati gli operatori di *crossover* e mutazione.

3.3.2 Generazione casuale della popolazione

```
Sub GenerateRandomPopulation()
  Dim i, k As Integer
  Dim Value As Integer

  For i=0 To PopSize
    For k=0 To 9
      Do
        Value = Floor(Rnd*10)
      Loop Until InStr(Genome(i), Str(Value)) = 0
      Genome(i) = Genome(i) + Str(Value)
    Next
  Next
End Sub
```

Listato 2: Codice *BASIC* della funzione per generare la popolazione iniziale.

Prima di tutto dobbiamo assumere che il cromosoma sia una stringa che contenga numeri da 0 a 9 che rappresentano le città. Il primo problema che sorge cominciando a pensare come creare la funzione per generare casualmente i cromosomi della prima popolazione è quello dell'univocità dei geni. Infatti non si può permettere che il viaggiatore visiti una città due volte. Per questo motivo l'istruzione *Floor(Rnd*10)*, che genera un numero intero da 0 a 9, è racchiusa in un *loop* che controlla ogni volta se il valore generato si trova già nel genoma. In caso affermativo ripete l'istruzione sopra scritta. Riconosco che non è molto elegante ed efficiente, ma ci fa risparmiare altro codice complesso.

3.3.3 Calcolo della lunghezza del percorso

```
Function ComputePathLength(index As Integer) As Double
  Dim i, LastPointNumber, PointNumber As Integer
  Dim Dx, Dy As Double
  Dim Sum As Double

  // Assegno la prima lettera del genoma numero index
  LastPointNumber = Val(Mid(Genome(index),1,1))
  For i=2 To 10
    // Assegno l'i-esima lettera del genoma numero index
    PointNumber = Val(Mid(Genome(index),i,1))

    Dx = Points(PointNumber,0) - Points(LastPointNumber,0)
    Dy = Points(PointNumber,1) - Points(LastPointNumber,1)
    Sum = Sum + Sqrt(Dx*Dx + Dy*Dy)

    LastPointNumber = PointNumber
  Next

  Return Sum
End Function
```

Listato 3: Codice *BASIC* della funzione per calcolare la lunghezza del percorso.

Questa funzione prende come argomento l'indice del cromosoma e restituisce la lunghezza del percorso corrispondente. Per far questo si tiene sempre conto del punto precedente e del corrente per prenderne le coordinate x e y, sottrarle ed applicare il teorema di Pitagora. Il tutto viene fatto all'interno di un ciclo che fa passare tutti i cromosomi, in modo da ottenere tutte le somme parziali del percorso che sommate costituiscono la lunghezza del tour.

3.3.4 Riproduzione selettiva

```
Sub SelectiveReproduction()
  Dim i,k As Integer
  Dim FitnessSum, Temp, RndValue As Double
  Dim NewGenome(PopSize) As String

  For i=0 To PopSize
    FitnessSum = FitnessSum + Fitness(i)
  Next

  For i=0 To PopSize
    RndValue = Rnd * FitnessSum

    Temp = 0
    For k=0 To PopSize
      Temp = Temp + Fitness(k)
```

```
        If RndValue < Temp Then
          NewGenome(i) = Genome(k)
          Exit
        End if
      Next

    Next

    For i=0 To PopSize
      Genome(i) = NewGenome(i)
    Next
  End Sub
```

Listato 4: Codice *BASIC* della funzione per generare la nuova popolazione.

Questa funzione si occupa di generare la nuova popolazione mediante il sistema della *roulette wheel selection*. Purtroppo il codice per implementare questo metodo di riproduzione non è così intuitivo come l'idea della ruota della fortuna, benchè produca gli stessi risultati. Inizialmente viene assegnata alla variabile *FitnessSum* la somma totale dei valori di *fitness* degli individui. Possiamo immaginarcela come una striscia divisa in diverse celle, che hanno lunghezza maggiore o minore a dipendenza del *fitness* dell'individuo corrispondente. Secondariamente viene selezionato casualmente un punto della striscia, che corrisponderà ad una cella rappresentante l'individuo in questione. In questo modo gli individui con *fitness* più alto avranno una cella più lunga e quindi una maggiore probabilità di venir scelti. Questa operazione viene svolta *n* volte, dove *n* è il numero di individui della popolazione precedente, in modo da ricrearne una di dimensioni uguali. L'ultimo ciclo ricopia semplicemente i genomi dei nuovi individui nella variabile globale *Genome* che sarà poi utilizzata dagli operatori di mutazione e incrocio.

3.3.5 L'incrocio

In questo problema l'operatore standard di *crossover* non può funzionare semplicemente perché scambiandosi dei geni non si ha la certezza che i due cromosomi risultanti contengano una città solo una volta. Quando si presenta un problema del genere la soluzione migliore è quella di restare nel modo più vicino al modello classico del *crossover*, sfruttandone le analogie.

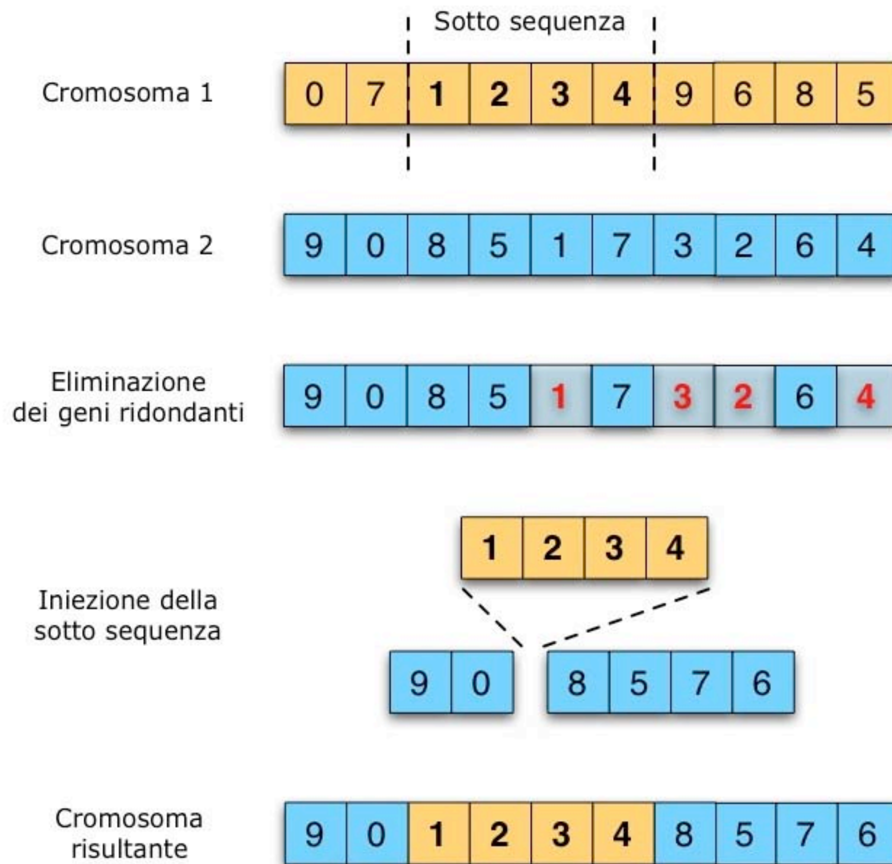


Figura 3.7: Esempio di applicazione del nuovo operatore di incrocio

L'idea non è molto complessa. Si tratta di prendere una sottosequenza di un cromosoma ed iniettarla nell'altro mantenendo però l'univocità delle città (figura 3.7). L'operazione sarà svolta analogamente al contrario, ossia prelevando una sotto sequenza dal cromosoma 2 e iniettandola nel primo. In questo modo si otterranno i due cromosomi voluti.

```
Sub DoCrossOver()
  Dim FirstCuttingPoint, SecondCuttingPoint, i, k, Maximum As Integer
  Dim GenomeCopy, Temp, NewGenes, NewGenome1, NewGenome2 As String

  For i=0 To PopSize - 1 Step 2
    If Rnd < CrossoverProbability Then
      FirstCuttingPoint = Floor(Rnd * 10) + 1 // Da 1 a 10
      SecondCuttingPoint = Floor(Rnd * 10) + 1

      Maximum = Max(FirstCuttingPoint, SecondCuttingPoint)
      FirstCuttingPoint = Min(FirstCuttingPoint, SecondCuttingPoint)
      SecondCuttingPoint = Maximum

      // Crossover del primo sul secondo
      NewGenes = ""
```

```
        GenomeCopy = Genome(i+1)

        For k=FirstCuttingPoint To SecondCuttingPoint
          GenomeCopy = ReplaceAll(GenomeCopy, Mid(Genome(i),k,1), "")
          NewGenes = NewGenes + Mid(Genome(i),k,1)
        Next

        Temp=Mid(GenomeCopy, FirstCuttingPoint)
        GenomeCopy = ReplaceAll(GenomeCopy, Temp, "")
        NewGenome1 = GenomeCopy + NewGenes + Temp

        // Crossover del secondo sul primo
        NewGenes = ""
        GenomeCopy = Genome(i)

        For k=FirstCuttingPoint To SecondCuttingPoint
          GenomeCopy = ReplaceAll(GenomeCopy, Mid(Genome(i+1),k,1), "")
          NewGenes = NewGenes + Mid(Genome(i+1),k,1)
        Next

        Temp = Mid(GenomeCopy, FirstCuttingPoint)
        GenomeCopy = ReplaceAll(GenomeCopy, Temp, "")
        NewGenome2 = GenomeCopy + NewGenes + Temp

        Genome(i) = NewGenome1
        Genome(i+1) = NewGenome2
      End if
    Next
  End Sub
```

Listato 5: Codice *BASIC* della funzione per applicare l'operatore di incrocio

La funzione *DoCrossOver* possiede un ciclo particolare. Questo fa scorrere i vari cromosomi ma incrementa la variabile *i* di 2 al posto di 1. In questo modo, all'interno del ciclo, si può lavorare direttamente con la coppia dei cromosomi *Genome(i)* e *Genome(i+1)*. L'istruzione successiva è un *if* che coinvolge la costante *CrossoverProbability*. Essa è stata definita all'inizio del codice e rappresenta la probabilità con la quale due cromosomi, possano commettere l'errore di scambiarsi del materiale genetico. Per questo motivo viene verificato se un numero casuale (compreso tra 0 e 1) è minore della costante in questione che, in questo caso, vale 0.80. Se dovesse essere così viene eseguito il codice interno.

Prima di tutto si scelgono i punti che racchiudono la sotto sequenza. È da notare che il primo punto deve essere sempre minore, o uguale, del secondo (perché altrimenti il ciclo *For* successivo non terminerebbe mai). Per questo se dovessero avere valori illegittimi essi vengono scambiati. La parte successiva di codice è divisa in due blocchi: quella che genera il cromosoma estraendo la sotto sequenza del primo e quella che lo fa estraendola dal secondo. Nel primo blocco il ciclo cancella tutti i geni ridondanti del secondo cromosoma e mette nella variabile *NewGenes* i geni da trasferire. Infine ricompone il nuovo genoma come è stato illustrato in figura 3.7. Nel secondo blocco la procedura è analoga.

3.3.6 La mutazione

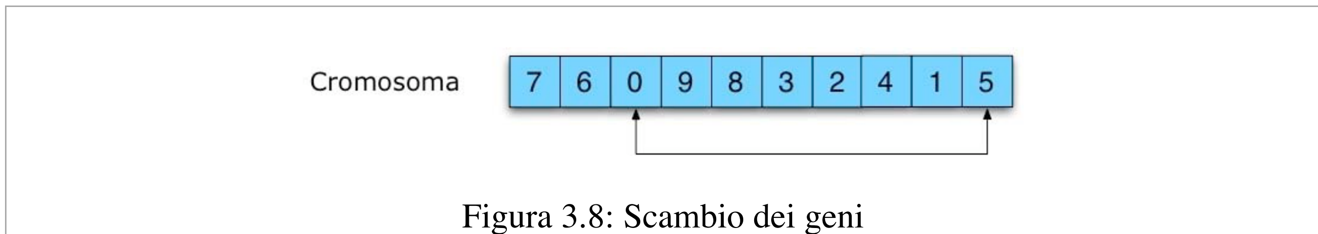


Figura 3.8: Scambio dei geni

Anche per la mutazione non può venir eseguita in modo classico. Perciò al posto che cambiare in modo casuale un gene, ne vengono selezionati due e poi scambiati di posto (figura 3.8).

```
Sub DoMutation()
  Dim i, FirstMutationPoint, SecondMutationPoint As Integer
  Dim Gene1, Gene2 As String

  For i=0 To PopSize
    If Rnd < MutationProbability Then
      FirstMutationPoint = Floor(Rnd * 10) + 1 // Da 1 a 10
      SecondMutationPoint = Floor(Rnd * 10) + 1

      Gene1 = Mid(Genome(i), FirstMutationPoint, 1)
      Gene2 = Mid(Genome(i), SecondMutationPoint, 1)

      Genome(i) = ReplaceAll(Genome(i), Gene1, " ")
      Genome(i) = ReplaceAll(Genome(i), Gene2, Gene1)
      Genome(i) = ReplaceAll(Genome(i), " ", Gene2)
    End if
  Next
End Sub
```

Listato 6: Codice *BASIC* della funzione per applicare l'operatore di mutazione

Come avviene per il *crossover* anche qui, all'interno del ciclo, si applica la mutazione con una certa probabilità che in questo caso è di 0.05. Il programma seleziona due geni a caso e mediante la manipolazione delle stringhe li scambia.

3.3.7 Risultati

Il codice sopra illustrato è stato scritto in *RealBasic* (http://www.realbasic.com), un ottimo *IDE* di programmazione. Ho scelto questo ambiente di sviluppo per tutti i programmi di questo lavoro poiché mi è sembrato molto semplice ed intuitivo (soprattutto per la gestione della grafica e la comunicazione con periferiche tramite porta seriale) ed al tempo stesso molto potente e compatibile. Infatti esiste la possibilità di compilare il codice sorgente per 3 diverse piattaforme: Windows, Mac OS X e Linux. Nel CD-Rom allegato è presente il programma con il codice sorgente in formato *RealBasic*.

Per l'esame dei risultati è stata aggiunta una porzione di codice per gestire l'interfaccia grafica e per esportare i dati che vengono registrati durante una simulazione. Il programma si compone di una finestra nella quale è presente una regione grafica che mostra il miglior percorso della popolazione corrente, e di due pulsanti. Il primo fa partire l'algoritmo genetico, il secondo esporta i dati del *fitness* dopo un'esecuzione. Esiste anche la possibilità di velocizzare nettamente la ricerca del percorso migliore selezionando l'opzione *Fast Mode* che non visualizza lo stato corrente del miglior genoma nell'area grafica.

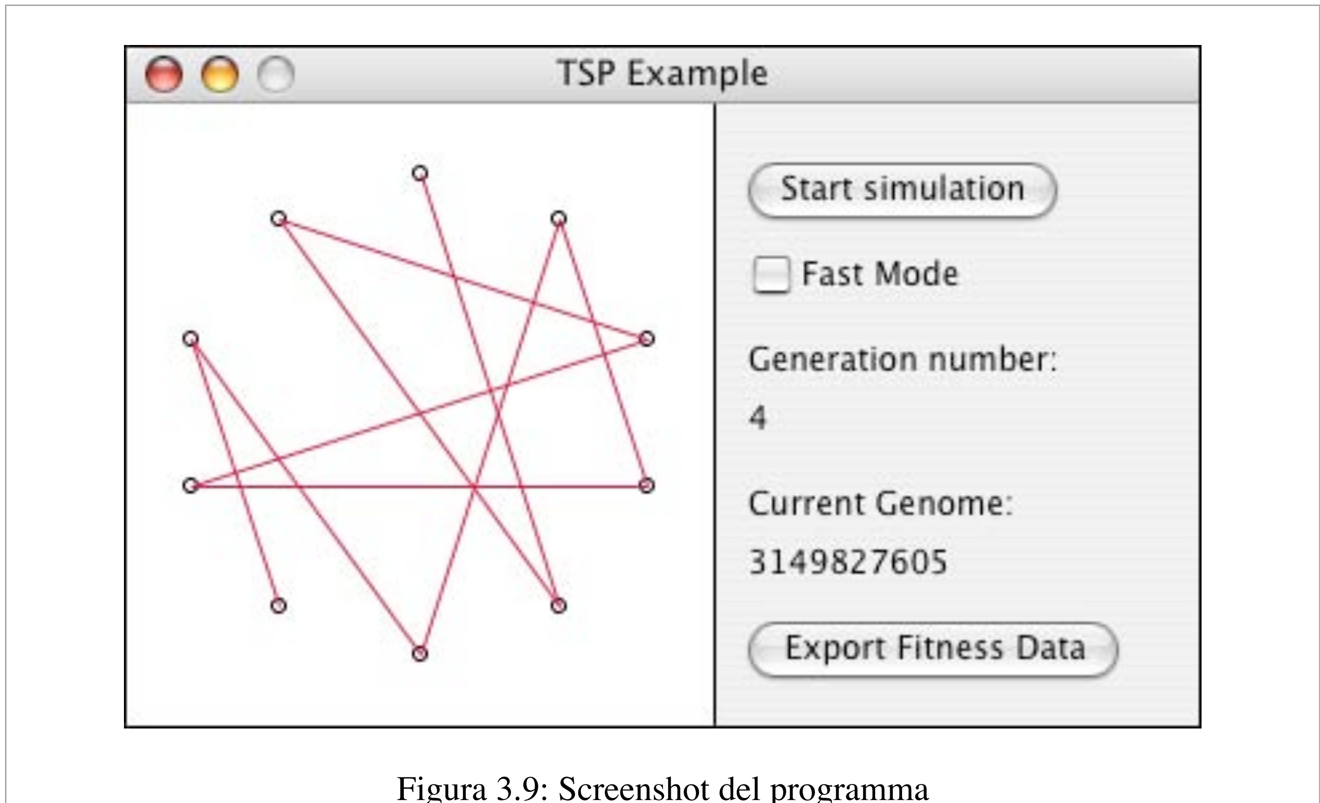


Figura 3.9: Screenshot del programma

Si può notare chiaramente che durante le prime generazioni il genoma è ancora pressochè casuale e possiede un fitness molto basso dato che il percorso è molto lungo. Nella figura 3.9 possiamo osservare la finestra del programma che rappresenta il percorso del miglior individuo alla quarta generazione. Generalmente il programma trova una soluzione dopo al massimo 20 secondi e dopo 5 secondi con la *Fast Mode* inserita. Mentre la simulazione è in corso si può osservare come all'inizio vengano subito eliminati i percorsi più inefficienti. Durante la simulazione ci si avvicina sempre di più alla soluzione, verso la fine può capitare che tutti i punti sono connessi tra di loro in modo corretto tranne due. Per qualche istante quel percorso rimane inalterato ma poi, grazie alla mutazione, esso viene "perturbato" e quindi ricondotto alla soluzione. La figura 3.10 mostra il programma ad esecuzione terminata. Per trovare la soluzione sono state impiegate 58 generazioni, in media non vengono mai superate le 200. Il genoma migliore è stato "5432109876". È interessante notare che non per forza la combinazione migliore doveva essere in ordine crescente ed iniziare da 0.

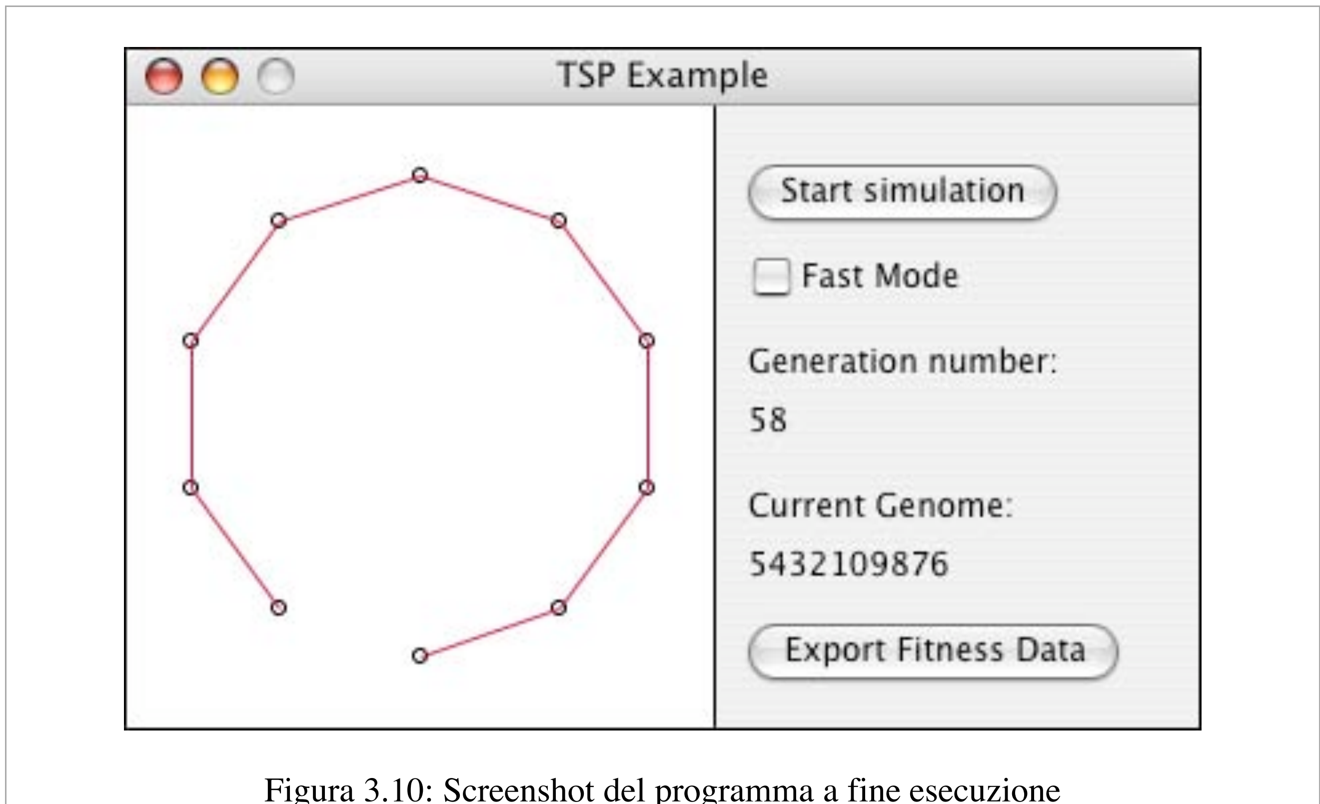


Figura 3.10: Screenshot del programma a fine esecuzione

Dopo la fine di un'esecuzione particolare in cui si sono lasciati evolvere gli individui anche dopo aver trovato la soluzione, sono stati esportati i dati della simulazione. Questi vengono salvati in formato di testo semplice che è leggibile dalla maggior parte di applicazioni di fogli elettronici. Le informazioni contenute sono tutti i valori dei *fitness* degli individui accanto ai quali viene indicato il numero di individuo e generazione. Inoltre vengono forniti anche dei dati statistici come la media del *fitness* di ogni generazione e il valore del rispettivo *fitness* migliore. Dopo aver importato questi dati in *Excel* è interessante rappresentare queste due misurazioni in funzione delle generazioni (figura 3.11). Come si vede chiaramente dal grafico il *fitness* migliore è, ovviamente, sempre al di sopra della media. Esso ha un andamento pressochè logaritmico: all'inizio tende a salire in modo abbastanza marcato (anche se con diverse oscillazioni) per poi stabilirsi quando ha raggiunto il massimo. Invece la media del *fitness* di ogni generazione tende sempre a migliorare in modo abbastanza lineare, infatti anche dopo la centesima generazione nella quale si è raggiunta la soluzione, la media delle prestazioni della popolazione continuano a salire. Questo approccio è molto utile per studiare l'evoluzione di problemi anche più complessi e per cercare di definire la funzione di *fitness* migliore e i coefficienti di mutazione e *crossover*.

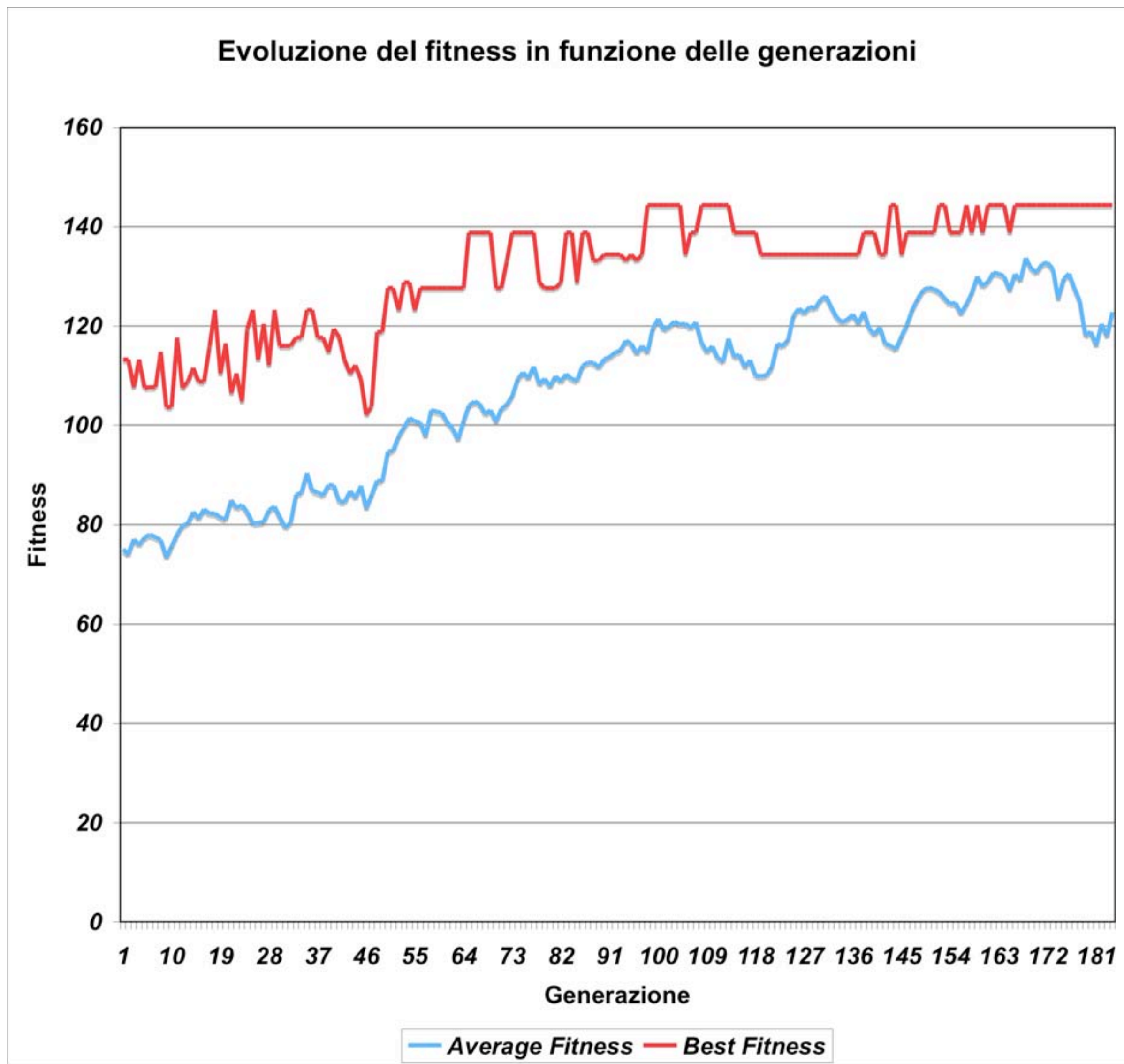


Figura 3.11

4. Gli organismi artificiali

4.1 Vita Artificiale

Per svolgere l'esperimento del capitolo 5 sono stati utilizzati degli organismi artificiali. Essi sono degli elementi costituenti di una scienza nata negli ultimi decenni chiamata Vita Artificiale (spesso chiamata anche *Alife* o AL per brevità). La Vita Artificiale ha come scopo la creazione di esseri appunto artificiali, ossia creati dall'uomo, che esprimano comportamenti simili a quelli degli organismi che conosciamo oggi noi in natura. Questo studio del comportamento potrebbe portare ad una migliore comprensione dei meccanismi che regolano la vita.

La differenza principale con la biologia è che questa studia il fenomeno della vita utilizzando un approccio *top-down*. Ossia osservando dall'esterno gli organismi e cercando di scomporli determinando le funzioni degli elementi del livello inferiore (già qui sorge il problema di non perdere di vista le varie correlazioni con gli elementi costituenti). Mentre l'approccio utilizzato dalla Vita Artificiale è quello inverso, *down-top*. Individuare, o ipotizzare, alcuni semplici meccanismi che regolano le interazioni tra gli elementi che costuiscono l'organismo e da cui derivano i complessi comportamenti globali, per poi cercare di verificare l'ipotesi simulando il fenomeno in esame. Ciò, chiaramente, è stato possibile solo da pochi decenni, da quando cioè è stato possibile effettuare queste simulazioni al computer.

Tuttavia le simulazioni di organismi artificiali effettuate mediante un computer spesso hanno degli svantaggi. Per questo si è ricorso a dei robot reali. Essi presentano alcuni vantaggi: come gli organismi biologici che troviamo in natura possiedono un corpo con determinate caratteristiche fisiche e geometriche che interagiscono con un ambiente con altrettante caratteristiche fisiche (luce, rumore di fondo, attriti, inerzia, ecc.). Queste proprietà sono spesso assenti o molto ridotte nelle simulazioni su computer perché è il programmatore dell'ambiente artificiale a decidere il grado di complessità dell'ambiente. Il risultato sarà il grado di realismo dell'esperimento più o meno alto.

4.2 Evoluzione di organismi artificiali

4.2.1 Combinazione reti neurali e algoritmi genetici

Esistono varie implementazioni della Vita Artificiale. Tuttavia la scelta migliore è, come visto nel capitolo introduttivo, quella di ricercare ed utilizzare sistemi autonomi che si evolvono. Ultimamente si tende quindi ad utilizzare un modello di organismo artificiale che soddisfi queste caratteristiche.

La combinazione di reti neurali e algoritmi genetici ha avuti molti successi in questo campo. Infatti rispecchiano la maggior parte di requisiti che anche un organismo biologico possiede.

Le reti neurali rappresentano il suo sistema nervoso, al quale giungono gli stimoli dagli organi di senso e da quale partono gli impulsi che controllano il corpo. Gli algoritmi genetici rappresentano la natura che costringe gli organismi ad evolversi.

Il vantaggio di questa combinazione è che non è necessario speficicare una risposta corretta per ogni attivazione della rete neurale mentre l'organismo è in vita. La funzione di *fitness* può essere più o meno dettagliata. Ad esempio, nel caso di un organismo artificiale che debba sopravvivere in un mondo con vari oggetti, il criterio di *fitness* potrebbe essere semplicemente la durata della sua vita o la quantità di cibo mangiato. Ovviamente per poter massimizzare il valore delle prestazioni, il sistema nervoso dell'organismo artificiale deve imparare a fare una serie di azioni: muoversi in modo corretto, riconoscere il cibo, raccoglierne la maggior quantità nella sua vita, evitare gli ostacoli; in generale deve imparare ad interpretare correttamente l'informazione sensoriale e coordinare le proprie azioni. Tutte queste abilità emergono spontaneamente a causa della pressione evolutiva, piuttosto che essere esplicitamente inserite o insegnate al sistema.

4.2.2 Implementazione

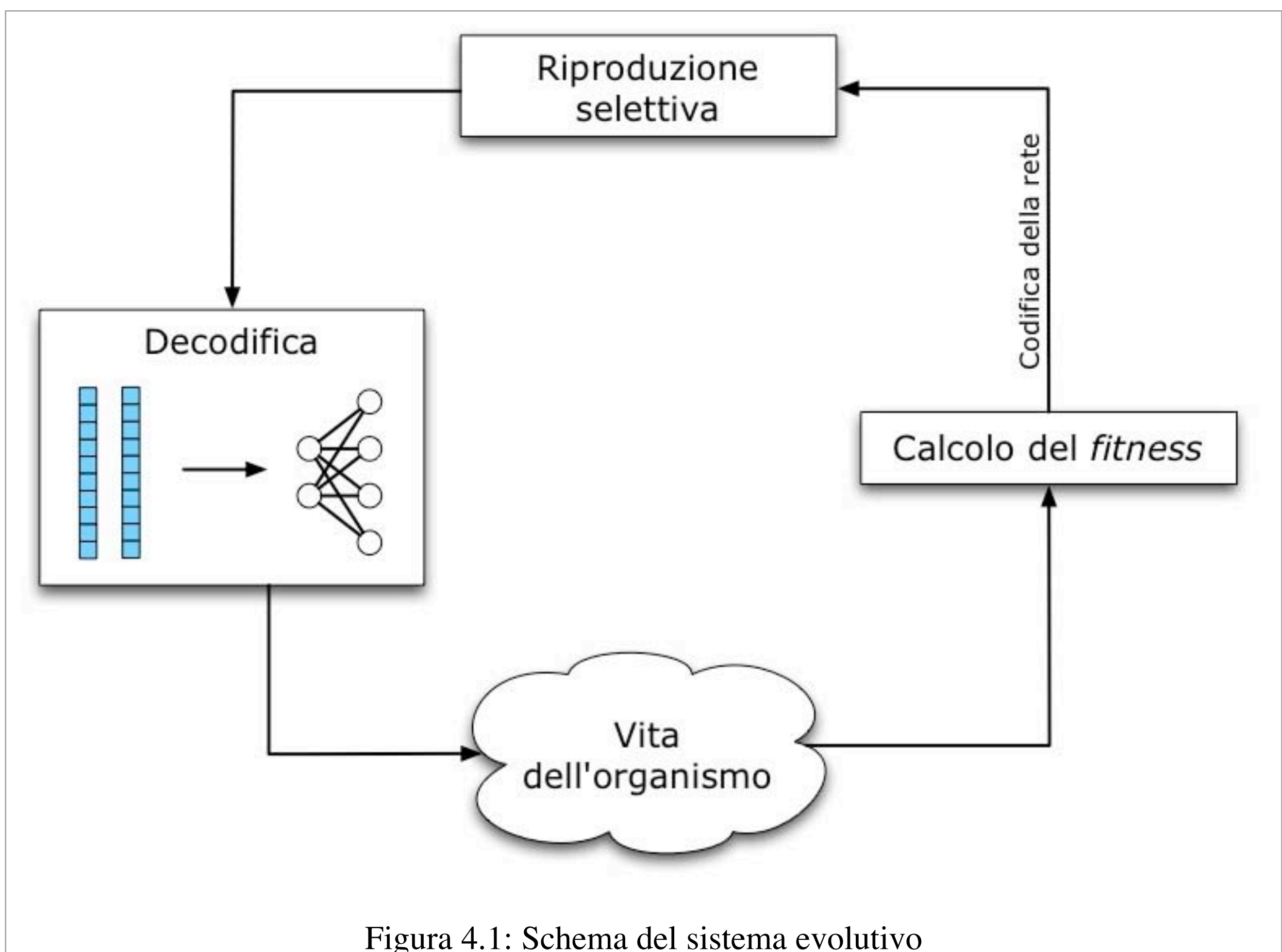


Figura 4.1: Schema del sistema evolutivo

Per poter evolvere il sistema nervoso occorre trovare un modo per codificare le caratteristiche della rete neurale. Nella maggior parte dei casi, il cromosoma di ogni organismo contiene l'insieme dei valori sinaptici della rete neurale. Ma potrebbe ad esempio codificare anche il tipo di architettura della rete, come la presenza di sinapsi e neuroni della

rete in determinati punti o le caratteristiche elettrochimiche delle sinapsi. Ogni gene non è un numero binario ma bensì un numero reale, che coincide con il valore della sinapsi. Il *crossover* viene effettuato scambiando sottosequenze di numeri reali, mentre la mutazione avviene aggiungendo al valore attuale del gene un valore reale casuale, solitamente compreso entro un certo intervallo. La popolazione iniziale viene generata creando una popolazione di cromosomi casuali (ossia le sinapsi vengono inizializzate con valori casuali). In seguito viene decodificato ciascun cromosoma creando il sistema nervoso corrispondente (figura 4.1). I neuroni di ingresso vengono collegati ai recettori sensoriali dell'organismo mentre quelli di uscita con l'apparato motorio dell'organismo in modo da lasciarlo libero di muoversi all'interno dell'ambiente. Durante questo tempo le sue prestazioni vengono registrate e valutate secondo la funzione di *fitness*. Alla fine della vita dell'organismo avviene la riproduzione selettiva e il ciclo ricomincia da capo.

5. Sviluppo della coordinazione sensomotoria in agenti robot

5.1 Introduzione

In questo capitolo descriverò un esperimento che ho cercato di riprodurre. Lo scopo è di sviluppare la coordinazione sensomotoria di un robot mobile che apprende in modo autonomo interagendo con l'arena in cui si muove. Più precisamente il robot in seguito descritto sarà inserito in un labirinto. Esso dovrà, grazie ai suoi sensori, sviluppare la capacità di muoversi evitando gli ostacoli.

Com'è già stato osservato nel capitolo 1, la maggior parte dei robot del giorno d'oggi sembrano solamente a prima vista autonomi e intelligenti. In realtà questo è dovuto alla programmazione in dettaglio del robot in modo che esprima i comportamenti voluti. Questo approccio determina la passività della macchina, la quale si limita ad eseguire le istruzioni predefinite. Al contrario, in questo esperimento, la macchina passiva si trasforma in un agente attivo che apprende grazie all'interazione con l'ambiente.

5.2 Simulazioni e robot reali

Prima di iniziare un esperimento occorre chiedersi qual è la strada migliore da percorre. Quella di una simulazione mediante un computer o un esperimento mediante un robot reale? Su questo tema esistono vari dibattiti, c'è chi sostiene che la simulazione sia uno strumento sufficiente per poter trarre delle conclusioni e chi invece sostiene il contrario.

I vantaggi di una simulazione sono che esse sono molto rapide a causa delle elevate prestazioni dei computer. In poco tempo si può riprodurre, mediante gli algoritmi genetici, la nascita di una popolazione fino alla centesima generazione, cosa che richiederebbe molto più tempo se fosse effettuata con un robot reale. Questo perché non si può evolvere in contemporanea robot reali (si potrebbe ma ciò comporterebbe una spesa molto elevata), quindi occorre farli evolvere uno ad uno isolati dagli altri. Occorre notare però che le simulazioni sono più veloci quando il grado di complessità dell'ambiente da simulare non è

molto elevato. Infatti per simulare un ambiente fisico occorre introdurre moltissime regole che rallentano molto la simulazione. Programmi recenti che simulano spazi soggetti a leggi fisiche che rassomigliano la realtà necessitano di una grande potenza di calcolo e per ogni passo impiegano diverso tempo. Anche la simulazione di un fenomeno fisico semplice come lo scontro con un muro risulta complesso.

Il secondo vantaggio delle simulazioni sembra essere l'economicità, ma in realtà se si vuole veramente ottenere un risultato plausibile anche per l'ambiente fisico reale occorre alzare il grado di realismo. Questo comporta l'impiego di programmatori specializzati, il che spesso è più costoso che lavorare con un robot reale.

Concludendo vorrei citare anche la "legge di Murphy", che afferma che un sistema biologico o fisico è soggetto a delle azioni di deterioramento, come il malfunzionamento, il danneggiamento di componenti e la graduale consumazione di parti meccaniche o biologiche. Queste proprietà non sono presenti in una simulazione.

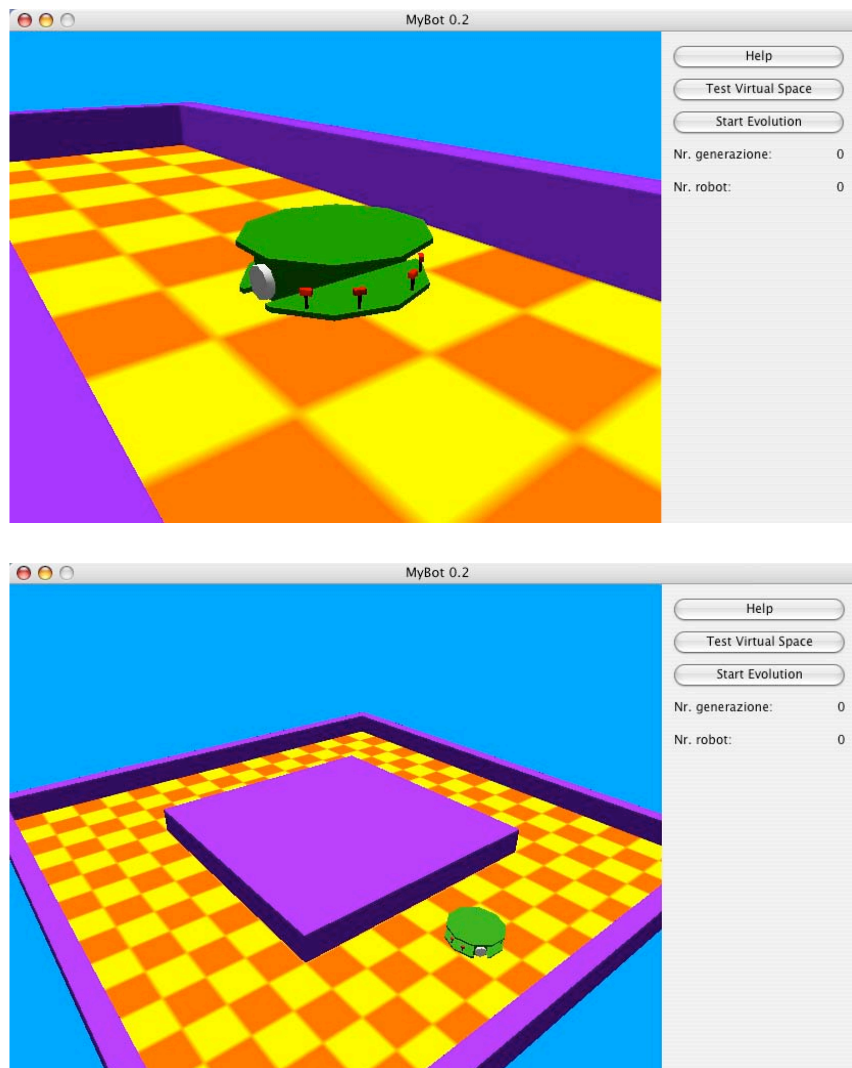


Figura 5.1: Due *screenshots* del programma di simulazione.

A testimonianza di ciò che è stato scritto, riporto la descrizione della prima versione dell'esperimento in questione effettuato mediante una simulazione. L'idea era quella di riprodurre un mondo soggetto alle leggi fisiche di base che il robot avrebbe incontrato spostandosi in tale ambiente. Per la programmazione è stato utilizzato il compilatore *RealBasic*. Nella figura 5.1 si può notare che il mondo virtuale in cui vive il robot viene visualizzato nella finestra dell'applicazione. Questa operazione viene fatta di continuo. In questo modo è possibile tenere sotto controllo l'evoluzione ed osservare i mutamenti comportamentali. Per visualizzare l'ambiente a tre dimensioni è stata utilizzata la libreria di grafica 3D *Quesa*[2]. Il robot è stato disegnato con un programma di CAD 3D chiamato *Carrara* e successivamente importato nell'applicazione. Esso rassomiglia al robot *Khepera* sviluppato dal *K-Team*. Ha una forma cilindrica, dei sensori di distanza davanti e dietro e si muove grazie a due ruote a velocità differenti.

Lo sviluppo del programma ha richiesto moltissimo tempo, sia nella pianificazione, sia nello sviluppo vero e proprio del codice, soprattutto a causa del lavoro svolto in un ambiente a 3 dimensioni e della simulazione delle leggi cinematiche a cui è sottoposto robot.

Prima di riportare le porzioni di codice e spiegarle è opportuno studiare la struttura del programma. Esso è stato realizzato utilizzando la programmazione ad oggetti, ed utilizzando anche le classi. Non essendo lo scopo di questo lavoro la spiegazione di concetti di programmazione, si da per scontato che essi siano noti.

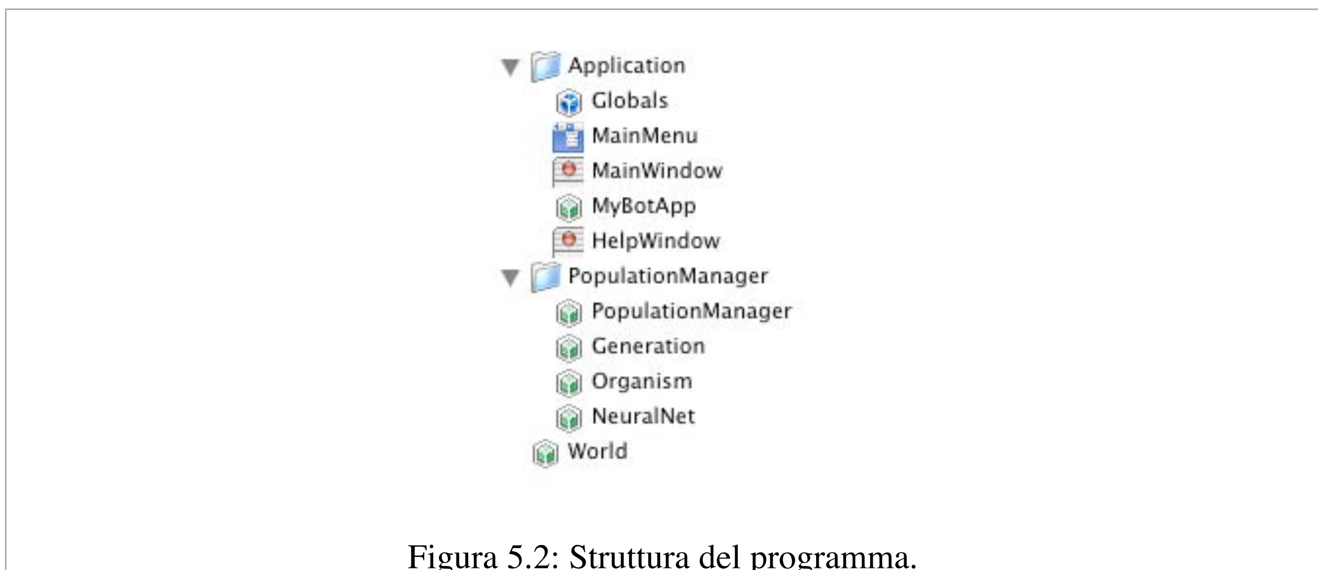


Figura 5.2: Struttura del programma.

Aprendo il file del codice sorgente mediante *RealBasic* si può subito comprendere l'organizzazione del programma (figura 5.2). Nella cartella *Application* vi sono diversi oggetti che servono al funzionamento del programma, quali alcune costanti (*globals*) ed alcune finestre per l'interfaccia grafica (*MainWindow*, è la finestra dove viene rappresentato il robot; *HelpWindow* è la finestra che compare premendo il pulsante *Help*). La cartella *PopulationManager* raggruppa invece tutte le classi necessarie alla gestione della popolazione. La classe *PopulationManager* possiede tutte le funzioni per la riproduzione selettiva e per creare quindi una nuova generazione. A sua volta, la classe *Generation*

[2] http://quesa.designcommunity.com/quesa.html

contiene un insieme di oggetti *Organism* (che rappresentano il singolo robot) e le funzioni per aggiungere un nuovo organismo. La classe *Organism* possiede diverse funzioni, come quella del calcolo del *fitness*, e un genoma in cui sono codificate le sinapsi della rete neurale, che è un oggetto della classe *NeuralNet*. Essa contiene le caratteristiche della rete e le funzioni per il calcolo degli output. La classe dedicata invece alla simulazione del mondo fisico è chiamata *World* ed è una sottoclasse della classe *Rb3Dspace*, un controllo dedicato al lavoro con ambienti 3D. La classe *World* è responsabile dunque della cinematica del robot, del calcolo dell'attivazione dei sensori di prossimità, del calcolo degli scontri con i muri e della rappresentazione visiva di tutto ciò. Essa è la classe più importante, e quella che ha richiesto più tempo.

Cominciamo ad esaminare il codice della classe *World*.

```
World.Open:
Sub Open()
  // Creo il mondo
  CreateWorld
End Sub

World.CreateWorld:
Sub CreateWorld()
  CreatePlane // Creo il piano
  CreateWalls // Creo i muri
  CreateBounds // Crea i limiti dei muri
  CreateBot // Creo il robot
  InitCamera // Inizializzo la posizione e la rotazione della camera

  me.update // Renderizzo la scena
End Sub
```

Listato 1

Appena il programma si apre è chiamato l'evento *Open*, il quale richiama la funzione *CreateWorld*. Essa ha il compito di creare e posizionare gli oggetti che costituiscono l'ambiente. Vale la pena fare un'osservazione alla funzione *CreateBounds*. È abbastanza complicato lavorare direttamente con dei modelli importati da un programma di CAD, questo perché essi sono definiti come un insieme di triangoli nello spazio che formano la superficie desiderata (*mesh*). Per calcolare ad esempio la distanza di un sensore di prossimità del robot o il suo contatto con lo stesso, si dovrebbero effettuare calcoli complessi riferiti a ciascun triangolo. Per questo motivo, esprimere la *mesh* come una semplice superficie geometrica (ne esistono di tre tipi: parallelepipedo, sfera e cilindro) che rappresenta la superficie di inviluppo che meglio approssima quella dell'oggetto (chiamata appunto *bound*), semplificherebbe molto i calcoli. Per i confini perimetrali, esterni ed interni, in cui opera il robot sono stati utilizzati dei parallelepipedi, simulanti i muri di contorno. Le altre funzioni si limitano o ad importare dei modelli 3D esterni, o ad impostarne la posizione e la rotazione. Di seguito sono riportate le 5 funzioni.

```
World.CreatePlane:
Sub CreatePlane()
  Dim PlanePict as Picture
  Dim xi,yi,begin as integer

  // Creo il piano 3D
  PlanePict=newpicture(160,160,16)
  PlanePict.graphics.foreColor=rgb(255,100,0)
  PlanePict.graphics.Fillrect 0,0,PlanePict.width,PlanePict.Height
  PlanePict.graphics.foreColor=rgb(255,255,0)
  // Disegno i quadrati colorati
  For yi=0 to 160 step 10
    For xi=begin to 160 step 20
      PlanePict.graphics.FillRect xi,yi,10,10
    Next
    if begin=0 then
      begin=10
    else
      begin=0
    end if
  Next

  // Aggiungo il piano nello spazio
  Plane=new object3D
  Plane.AddShapePicture PlanePict,0.5
  Plane.RenderBackFaces=true
  me.objects.append Plane

  // Aggiusto la sua posizione nello spazio
  Plane.pitch -pi/2
End Sub

World.CreateWalls:
Sub CreateWalls()
  Dim f as FolderItem
  Dim obj as Object3D

  // Carico il modello 3D dei muri
  f=getFolderItem("")
  f=f.Child("3DModels").Child("Walls.3DMF")
  obj=new Object3D
  obj.AddShapeFromFile f
  Walls=obj
  Walls.renderBackFaces=true
  me.objects.Append Walls

  // Lo posiziono nello spazio
  Walls.pitch -pi/2
  Walls.position.y=2
End Sub
```

```
World.CreateBounds:
Sub CreateBounds()
  Dim v1,v2 as Vector3D

  v1=new Vector3D
  v2=new Vector3D

  v1.x=40
  v1.z=-40
  v1.y=0

  v2.x=45
  v2.z=40
  v2.y=4

  Bounds(0)=new Bounds3D(v1,v2) // sinistra

  v1.x=-45
  v1.z=-40
  v1.y=0

  v2.x=-40
  v2.z=40
  v2.y=4

  Bounds(1)=new Bounds3D(v1,v2) // destra

  v1.x=-40
  v1.z=40
  v1.y=0

  v2.x=40
  v2.z=45
  v2.y=4

  Bounds(2)=new Bounds3D(v1,v2) // su

  v1.x=-40
  v1.z=-45
  v1.y=0

  v2.x=40
  v2.z=-40
  v2.y=4

  Bounds(3)=new Bounds3D(v1,v2) // giu'

  v1.x=-20
  v1.z=-20
  v1.y=0

  v2.x=20
  v2.z=20
```

```
  v2.y=4

  Bounds(4)=new Bounds3D(v1,v2) // centro
End Sub

World.CreateBot:
Sub CreateBot()
  Dim f as FolderItem
  Dim obj as Object3D

  // Carico il modello 3D del robot
  f=getFolderItem("")
  f=f.Child("3DModels").Child("MyBot.3DMF")
  obj=new Object3D
  obj.AddShapeFromFile f
  Bot=obj
  Bot.renderBackFaces=false
  me.objects.Append Bot

  // Lo posiziono nello spazio
  Bot.pitch -pi/2
  rot=pi/2 // Imposto la rotazione iniziale del robot a 90°
  Bot.position.x=30
  Bot.position.y=0.95

  //Imposto la velocita' iniziale a 0
  WR=0.5
  WL=0.5
End Sub

World.InitCamera:
Sub InitCamera()
  // Imposto la latitudine, la longitudine e il raggio della posizione della
camera sull'orbita sferica
  Latit=pi/4
  Longi=pi/4
  Radius=75
  wview=true

  ArrangeCamera // Aggiorno la posizione e l'orientamento della camera
End Sub
```

Listato 2

Passiamo ora al cuore della classe *World*. Si tratta della funzione *NextFrame*. Essa è responsabile di calcolare e visualizzare la "prossima immagine" del robot e del suo ambiente. La funzione - oltre a calcolare il vettore di spostamento del robot, a controllare eventuali collisioni con i muri e a calcolare il valore dei sensori di prossimità virtuali - deve tener conto anche di possibili input da tastiera. Infatti, durante la simulazione, esiste la possibilità di ruotare l'ipotetica telecamera, che sta filmando il robot, su una orbita sferica mediante la pressione dei tasti direzionali e numerici. Nel listato 3 è riportata la funzione *NextFrame* e le funzioni correlate.

```
World.NextFrame:
Sub NextFrame()
  ArrangeCamera
  BotNextPosition

  if mode=0 then // "Test Virtual Space" mode
    me.update
  elseif mode=1 then // "Evolution" mode
    if keyboard.asynckeyDown(&h31) then // Aggiorno se viene premuto il tasto
[spazio]
      me.update
    end if
  end if
End Sub

World.ArrangeCamera:
Sub ArrangeCamera()
  GetCameraKeys // Prendo gli input dai tasti per la posizione dalla tastiera

  // Calcolo la posizione della camera
  if rview then
    me.camera.position=OrbitSphere(Bot.position,radius,Longi,Latit)
    PointObjectAt me.camera,Bot.Position
  else
    me.camera.position=OrbitSphere(Walls.position,radius,Longi,Latit)
    PointObjectAt me.camera,Walls.Position
  end if
End Sub

World.GetCameraKeys:
Sub GetCameraKeys()
  if keyboard.asyncKeyDown(&h7B) then // sinistra
    Longi=Longi-0.05
  end if
  if keyboard.asyncKeyDown(&h7C) then // destra
    Longi=Longi+0.05
  end if
  if keyboard.asyncKeyDown(&h7D) then // giu'
    Latit=Latit-0.05
    if latit<=0.2 then
      latit=0.2
    end if
  end if
  if keyboard.asyncKeyDown(&h7E) then // su
    Latit=Latit+0.05
    if latit>=pi/2-0.2 then
      latit=pi/2-0.2
    end if
  end if
  if keyboard.asyncKeyDown(&h45) then // +
    radius=radius-1
    if radius<=20 then
```

```
      radius=20
    end if
  end if
  if keyboard.asyncKeyDown(&h4E) then // -
    radius=radius+1
    if radius>=120 then
      radius=120
    end if
  end if

  if keyboard.asyncKeyDown(&h12) then // 1
    rview=false
    wview=true
  end if
  if keyboard.asyncKeyDown(&h13) then // 2
    rview=true
    wview=false
  end if
End Sub

World.OrbitSphere:
Function OrbitSphere(POI as Vector3D, radius as Double, longitude as double,
latitude as Double) As Vector3D
  Dim v As Vector3D
  Dim q,q2 As Quaternion

  q = New Quaternion
  q.SetRotateAboutAxis 0,1,0, longitude
  q2 = New Quaternion
  q2.SetRotateAboutAxis 0,0,1, latitude
  q2.MultiplyBy q

  v = New Vector3D
  v.x = radius

  return q2.Transform(v).Plus(POI)
End Function

World.PointObjectAt:
Sub PointObjectAt(obj As Object3D, POI As Vector3D)
  Dim yawAngle As Double
  Dim pitchAngle As Double
  Dim v1, v2 As Vector3D

  yawAngle = Atan2(POI.x-obj.position.x, POI.z-obj.position.z)

  obj.orientation.SetRotateAboutAxis 0,1,0, yawAngle

  v1 = New Vector3D
  v1.z = 1.0
  v1 = obj.orientation.Transform(v1)
  v2 = New Vector3D
  v2.x = POI.x - obj.position.x
```

```
  v2.y = POI.y - obj.position.y
  v2.z = POI.z - obj.position.z
  v2.Normalize
  pitchAngle = Acos( Min(Max(v1.Dot(v2),-1.0),1.0) )
  If v2.y > 0 then
    pitchAngle = -pitchAngle
  end if

  obj.Pitch pitchAngle
End Sub

World.BotNextPosition:
Sub BotNextPosition()
  Dim Contact as boolean

  // Calcolo le componenti del vettore velocita' avendo
  // il raggio della ruota (0.625) e la distanza tra esse (7)
  // e lo aggiungo al vettore della posizione del robot
  Bot.Position.Add(ComputeVectorVelocity(0.625,7))

  if mode=0 then // "Test Virtual Space" mode
    Contact=checkContact(3.5)
  end if
End Sub

World.ComputeVectorVelocity:
Function ComputeVectorVelocity(WheelRadius as double, WheelsDistance as double)
As Vector3D
  // Velocity=vettore velocita' del robot, WR=velocita' angolare right,
WL=velocita' angolare left,
  // VR=velocita' lineare right, VL=velocita' lineare left, R=raggio di
curvatura,
  // D=spazio percorso, curv=angolo di curvatura, rot=rotazione del robot,
  Dim Velocity as Vector3D
  Dim VR,VL,R,D,curv as double

  Velocity=new Vector3D

  if mode=0 then
    // Controllo manuale
    if keyboard.asynckeydown(&h56) then
      WL=WL+0.01
    end if
    if keyboard.asynckeydown(&h53) then
      WL=WL-0.01
    end if
    if keyboard.asynckeydown(&h57) then
      WR=WR+0.01
    end if
    if keyboard.asynckeydown(&h54) then
      WR=WR-0.01
    end if
  end if
```

```
  // Calcolo le due velocita' lineari moltiplicando il raggio della ruota per la
velocita' angolare
  VR=WheelRadius*(WR-0.5)*10
  VL=WheelRadius*(WL-0.5)*10

  // Calcolo il raggio di curvatura
  R=WheelsDistance/2*( (VR+VL)/(VR-VL) )

  // Tre casi:
  // 1) La velocita' right e' l'opposto della velocita' left
  // --> fai girare il robot su se' stesso
  // 2) Le due velocita' sono uguali
  // --> fai procedere il robot in avanti senza rotazione
  // 3) Tutti gli altri casi
  // --> calcola la rotazione del robot e calcola le componenti del vettore
velocita'
  if VR=-VL then
    Velocity.X=0
    Velocity.Z=0
    rot=rot+(VR-VL)/WheelsDistance
    Bot.roll (VR-VL)/WheelsDistance
  elseif (VR-VL)=0 then
    Velocity.X=-VR*cos(rot) // metto "-" davanti alla componente dato il sist.
di rif.
    Velocity.Z=VR*sin(rot)
  else
    D=(VL+VR)/2
    curv=D/R
    rot=rot+curv
    Bot.roll curv
    Velocity.X=-1/2*(VL+VR)*cos(rot) // metto "-" davanti alla componente dato
il sist. di rif.
    Velocity.Z=1/2*(VL+VR)*sin(rot)
  end if

  Return Velocity
End Function

World.CheckContact:
Function CheckContact(BotRadius as double) As boolean
  Dim Contact as boolean

  // Gestisce il contatto con i muri esterni
  If bot.position.Z>40-BotRadius then
    bot.position.Z=40-BotRadius
    Contact=true
  end if

  If bot.position.Z<-(40-BotRadius) then
    bot.position.Z=-(40-BotRadius)
    Contact=true
  end if
```

```
  If bot.position.X>40-BotRadius then
    bot.position.X=40-BotRadius
    Contact=true
  end if

  If bot.position.X<-(40-BotRadius) then
    bot.position.X=-(40-BotRadius)
    Contact=true
  end if

  // Gestisce il contatto con i muri interni
  if Bot.position.Z<(20+BotRadius) and Bot.position.Z/abs(Bot.position.X)>1 then
    Bot.position.Z=20+BotRadius
    Contact=true
  end if

  if Bot.position.Z>-(20+BotRadius) and Bot.position.Z/abs(Bot.position.X)<-1
then
    Bot.position.Z=-(20+BotRadius)
    Contact=true
  end if

  if Bot.position.X<(20+BotRadius) and Bot.position.X/abs(Bot.position.Z)>1 then
    Bot.position.X=20+BotRadius
    Contact=true
  end if

  if Bot.position.X>-(20+BotRadius) and Bot.position.X/abs(Bot.position.Z)<-1
then
    Bot.position.X=-(20+BotRadius)
    Contact=true
  end if

  return Contact
End Function

World.GetSensorValue:
Function GetSensorValue(SensorAngle as double) As double
  // Restituisce l'intensita' del sensore a partire dall'angolo in gradi
  Dim intensity as double
  dim i as integer
  Dim v as Vector3D

  v=new Vector3D

  for intensity=3.5 to 18.5
    v.x=-intensity*cos(rot+SensorAngle*pi/180)
    v.z=intensity*sin(rot+SensorAngle*pi/180)
    v.Add(Bot.position)
    v.y=2

    for i=0 to 4
```

```
      if Bounds(i).ContainsPoint(v)=true then
        return 1-(intensity-3.5)/15
      end if
    next
  next
End Function
```

Listato 3

Durante lo sviluppo della porzione di codice appena riportata è sorto un problema riguardante la cinematica del robot. Esso, come sarà descritto nel capitolo 5.3 più peculiarmente, possiede una forma circolare e per muoversi è dotato di due ruote laterali. La particolarità sta nel fatto che la velocità delle due ruote non è identica, ma è indipendente l'una dall'altra. In questo modo può curvare molto agilmente ed addirittura girare su sé stesso. Il problema è dunque questo: date le velocità angolari delle ruote, il raggio e la distanza fra di esse, calcolare lo spostamento *dx* e *dy* del robot all'istante successivo. A questo punto, richiedendo questo studio del movimento conoscenze di matematica che non mi erano ancora del tutto note, ho consultato un testo sullo studio dei movimenti dei robot del politecnico di Milano[3] intitolato "Struttura dei robot a ruote".

Nel listato 4 troviamo invece le funzioni delle altre classi. Esse traducono in linguaggio di programmazione la teoria di evoluzione di un organismo artificiale.

```
PopulationManager.GenerateFirstGeneration:
Sub GenerateFirstGeneration()
  // Creo la prima generazione di 80 robot con codice genetico casuale
  Dim i as integer

  // Incremento il contatore di generazioni
  GenerationsCount=GenerationsCount+1
  MainWindow.GenerazioneNR.text=str(GenerationsCount)
  MainWindow.refresh

  CurrentGeneration.reset
  for i=0 to 79
    CurrentGeneration.AddRndRobot
  next
End Sub

PopulationManager.StartEvolution:
Sub StartEvolution()
  // Disabilito i task in background per rendere l'esecuzione
  // del programma piu' veloce
  #pragma disableBackgroundTasks
  #pragma disableBoundsChecking

  // Inizializzo le due variabili
  CurrentGeneration= new Generation
  NextGeneration=new Generation
```

[3] http://www.elet.polimi.it/upload/gini/rob/ruote.pdf

```
  // Imposto le variabili di probabilita' di crossover e mutation
  CrossoverProbability=0.8
  MutationProbability=0.1

  // Creo la prima generazione
  GenerateFirstGeneration

  // Avvio il loop principale
  do
    RunCurrentGeneration
    GenerateNextGeneration
  loop
End Sub

PopulationManager.RunCurrentGeneration:
Sub RunCurrentGeneration()
  dim i as integer

  for i=0 to 79
    // Lascio il robot libero di muoversi nell'arena
    RunRobot(i)

    // Incremento il contatore dei robot
    MainWindow.RobotNR.text=str(i+1)
    MainWindow.GenerazioneNR.refresh
    MainWindow.RobotNR.refresh
    MainWindow.Space.refresh
    MainWindow.updateNow
  next
End Sub

PopulationManager.GenerateNextGeneration:
Sub GenerateNextGeneration()
  Dim i as integer

  // Incremento il contatore di generazioni
  GenerationsCount=GenerationsCount+1
  MainWindow.GenerazioneNR.text=str(GenerationsCount)

  // Codifico il genoma di tutti i nuovi robot
  for i=0 to 79
    CurrentGeneration.robot(i).CodifyGenome
  next

  // Inizializzo la prossima generazione
  NextGeneration.Reset

  // Creo la ruota della fortuna per la probabilita' di riproduzione degli
individui migliori
  CreateFortuneWheel

  // Giro la ruota (80 volte) per selezionare gli individui della prossima
```

```
generazione
  for i=0 to 79
    RotateWheel
  next

  NextGeneration.Robot.shuffle

  // Eseguo il crossover e la mutazione sui robot della nuova generazione
  DoCrossover
  DoMutation

  // Metto i nuovi robot nella generazione corrente
  PutNewRobotsInCurrentGeneration

  // Decodifico il loro genoma
  for i=0 to 79
    CurrentGeneration.robot(i).DecodeGenome
  next
End Sub

PopulationManager.RunRobot:
Sub RunRobot(index as integer)
  dim i as integer

  // Azzero il fitness
  CurrentGeneration.robot(index).Fitness=0

  for i=0 to 250
    // Assegno l'intensita' dei sensori e le velocita' dei due motori agli input
della rete neurale del robot

CurrentGeneration.robot(index).NNet.input(0)=MainWindow.Space.getSensorValue(20)

CurrentGeneration.robot(index).NNet.input(1)=MainWindow.Space.getSensorValue(50)

CurrentGeneration.robot(index).NNet.input(2)=MainWindow.Space.getSensorValue(160
)

CurrentGeneration.robot(index).NNet.input(3)=MainWindow.Space.getSensorValue(-
20)

CurrentGeneration.robot(index).NNet.input(4)=MainWindow.Space.getSensorValue(-
50)

CurrentGeneration.robot(index).NNet.input(5)=MainWindow.Space.getSensorValue(-
160)
    CurrentGeneration.robot(index).NNet.input(6)=MainWindow.Space.WR
    CurrentGeneration.robot(index).NNet.input(7)=MainWindow.Space.WL

    // Calcolo gli output della rete neurale
    CurrentGeneration.robot(index).NNet.ComputeOutputs

    // Assegno gli output della rete neurale alle velocita' dei due motori del
```

```
robot
    MainWindow.Space.WR=CurrentGeneration.robot(index).NNet.output(0)
    MainWindow.Space.WL=CurrentGeneration.robot(index).NNet.output(1)

    // Aggiorno la posizione del robot
    MainWindow.Space.NextFrame

    // Se il robot va a sbattere imposto la proprieta' WallContact. Nota: 3.5 e'
il raggio del robot.

CurrentGeneration.robot(index).WallContact=MainWindow.Space.CheckContact(3.5)

    // Calcolo il fitness
    CurrentGeneration.robot(index).ComputeFitness
  next
End Sub

PopulationManager.CreateFortuneWheel:
Sub CreateFortuneWheel()
  Dim i as integer

  FortuneWheel=0
  For i=0 to 79
    FortuneWheel=FortuneWheel+CurrentGeneration.robot(i).Fitness
  next
End Sub

PopulationManager.RotateWheel:
Sub RotateWheel()
  Dim i as integer
  Dim ScoreValue, PreviousValues as double

  ScoreValue=rnd*FortuneWheel

  For i=0 to 79
    if ScoreValue<=(CurrentGeneration.robot(i).Fitness+PreviousValues) then
      NextGeneration.AddRobot CurrentGeneration.robot(i)
      return
    end if
    PreviousValues=PreviousValues+CurrentGeneration.robot(i).Fitness
  next
End Sub

PopulationManager.DoCrossover:
Sub DoCrossover()
  Dim i,k,CutPoint as integer
  Dim TempGenome(45) as double

  // Faccio passare tutte le 40 coppie di nuovi robot per un potenziale
crossover
  for i=0 to 78 step 2
    If rnd<=CrossoverProbability then // Se c'e' la probabilita' del crossover
lo eseguo
```

```
      CutPoint=round(rnd*45) // Trovo un punto casuale dove tagliare il genoma

      // Scambio il genoma a partire dal punto di taglio
      for k=CutPoint to 45
        TempGenome(k)=NextGeneration.robot(i).Genome(k)
      next
      for k=CutPoint to 45
        NextGeneration.robot(i).Genome(k)=NextGeneration.robot(i+1).Genome(k)
      next
      for k=CutPoint to 45
        NextGeneration.robot(i+1).Genome(k)=TempGenome(k)
      next
    end if
  next
End Sub

PopulationManager.DoMutation:
Sub DoMutation()
  Dim i,MutationPoint as integer

  // Faccio passare tutti i 79 nuovi robot per una potenziale mutazione
  For i=0 to 79
    if rnd<=MutationProbability then // Se c'e' la probabilita' della mutazione
la eseguo

      MutationPoint=round(rnd*45) // Trovo un punto casuale dove effettuare la
mutazione

      // Effettuo la mutazione

NextGeneration.robot(i).Genome(MutationPoint)=NextGeneration.robot(i).Genome(Mut
ationPoint)+(rnd-0.5)
    end if
  next
End Sub

PopulationManager.PutNewRobotsInCurrentGeneration:
Sub PutNewRobotsInCurrentGeneration()
  Dim i as integer

  // Inizializzo la generazione corrente per aggiungerci i nuovi robot
  CurrentGeneration.reset

  for i=0 to 79
    CurrentGeneration.AddRobot NextGeneration.robot(i)
  next
End Sub

MainWindow.Close:
Sub Close()
  Quit
End Sub
```

```
MainWindow.PushButton1.Action:
Sub Action()
  // Faccio partire l'evoluzione del robot
  Dim PopManager as PopulationManager

  // Imposto la modalita' "Evolution" ed avvio l'evoluzione
  Space.mode=1
  PopManager= new PopulationManager
  PopManager.StartEvolution
End Sub

Generation.AddRobot:
Sub AddRobot(NewRobot as Organism)
  // Aggiungo un nuovo robot alla poplazione
  robot(OrganismsCount)=NewRobot
  OrganismsCount=OrganismsCount+1
End Sub

Generation.Reset:
Sub Reset()
  // Elimino tutti gli organismi della popolazione
  OrganismsCount=0
End Sub

Generation.AddRndRobot:
Sub AddRndRobot()
  // Aggiungo un robot con codice genetico casuale
  Dim RndRobot as Organism

  RndRobot=new Organism
  RndRobot.NNet=new NeuralNet

  RndRobot.NNet.RndSynapsesAndThresholds

  AddRobot RndRobot
End Sub

Organism.CodifyGenome:
Sub CodifyGenome()
  Dim i,j,count as integer

  // Codifica la sinapsi dello strato input-hidden
  for j=0 to 3
    for i=0 to 7
      Genome(count)=NNet.L1Synapse(i,j)
      count=count+1
    next
  next

  // Codifica la sinapsi dello strato hidden-output
  for j=0 to 1
    for i=0 to 3
```

```
      Genome(count)=NNet.L2Synapse(i,j)
      count=count+1
    next
  next

  // Codifica le soglie di attivazione dei neuroni dello strato hidden e output
  For i=0 to 5
    Genome(count)=NNet.NeuronActivationThreshold(i)
    count=count+1
  next
End Sub

Organism.DecodeGenome:
Sub DecodeGenome()
  Dim i,j,count as integer

  // Decodifica la sinapsi dello strato input-hidden
  for j=0 to 3
    for i=0 to 7
      NNet.L1Synapse(i,j)=Genome(count)
      count=count+1
    next
  next

  // Decodifica la sinapsi dello strato hidden-output
  for j=0 to 1
    for i=0 to 3
      NNet.L2Synapse(i,j)=Genome(count)
      count=count+1
    next
  next

  // Decodifica le soglie di attivazione dei neuroni dello strato hidden e
output
  For i=0 to 5
    NNet.NeuronActivationThreshold(i)=Genome(count)
    count=count+1
  next
End Sub

Organism.ComputeFitness:
Sub ComputeFitness()
  // Per calcolare il fitness dell'organismo prende i dati
  // dai neuroni di input della rete neuronale

  Dim RWR,RWL as double
  Dim i(5),k as integer

  for k=0 to 5
    i(k)=NNet.input(k)
  next

  i.sort
```

```
  if not(WallContact) then
    RWR=(NNet.output(0)-0.5)
    RWL=(NNet.output(1)-0.5)
  else
    RWR=0
    RWL=0
  end if

  Fitness=Fitness + abs( ( RWR +RWL ) / 2 ) * ( 1 - sqrt (abs( ( NNet.output(0)
- 0.5 ) - ( NNet.output(1) - 0.5 ) ) ) ) * ( 1- i(5) )
End Sub

NeuralNet.RndSynapsesAndThresholds:
Sub RndSynapsesAndThresholds()
  // Assegno un valore casuale alle sinapsi da -0.5 a 0.5

  dim i,j as integer

  // Sinapsi del layer input-hidden
  for j=0 to 3
    for i=0 to 7
      L1Synapse(i,j)=rnd-0.5
    next
  next

  // Sinapsi del layer hidden-output
  for j=0 to 1
    for i=0 to 3
      L2Synapse(i,j)=rnd-0.5
    next
  next

  // Assegno un valore casuale alle soglie di attivazione dei neuroni da 0.50 a
1.50
  for i=0 to 5
    NeuronActivationThreshold(i)=1+(rnd-0.5)
  next
End Sub

NeuralNet.ComputeOutputs:
Sub ComputeOutputs()
  // Calcola gli output della rete neuronale
  Dim i,j as integer
  Dim TotalSum as double

  // Output dei neuroni dello strato hidden
  for j=0 to 3
    for i=0 to 7
      TotalSum=TotalSum+input(i)*L1Synapse(i,j)
    next
    hidden(j)=Sigmoid(TotalSum,j)
    TotalSum=0
```

```
  next

  // Output dei neuroni dello strato output
  for j=0 to 1
    for i=0 to 3
      TotalSum=TotalSum+hidden(i)*L2Synapse(i,j)
    next
    output(j)=Sigmoid(TotalSum,j)
    TotalSum=0
  next
End Sub

NeuralNet.Sigmoid:
Function Sigmoid(number as double, neuron as integer) As double
  return 1/(1+exp(-NeuronActivationThreshold(neuron)*number))
End Function

MainWindow.PushButton2.Action:
Sub Action()
  Space.mode=0

  do
    Space.NextFrame
  loop until keyboard.asyncKeyDown(&h35) // Esco alla pressione del tasto [esc]
End Sub

MainWindow.PushButton3.Action:
Sub Action()
  HelpWindow.show
End Sub

World.Open:
Sub Open()
  // Creo il mondo
  CreateWorld
End Sub
```

Listato 4

Dopo molti tentativi, i risultati della simulazione non sono stati accettabili, questo perché il livello di interazione con l'ambiente era troppo, oserei dire, banale. L'altro aspetto negativo è stato il grande tempo di programmazione impiegato per la simulazione dell'ambiente virtuale (cosa che non rientrava nel fine ultimo dell'esperimento) e la richiesta di buone conoscenze di programmazione. Si può quindi concludere che non sempre ci si devono aspettare risultati migliori da una simulazione.

5.3 Evoluzione di un robot reale

5.3.1 Materiale

Il robot che è stato utilizzato in questo esperimento è *Khepera* (figura 5.3). Esso ha una forma cilindrica di diamentro 55 mm, un'altezza di 30 mm e un peso di 70 g. La sua forma cilindrica gli permette di muoversi sopra su una sperficie senza incastrarsi. Questo è una caratteristica fondamentale per la riuscita dell'esperimento poiché esso dovrà essere lasciato evolvere per diversi giorni e quindi non è assicurata una supervisione da parte dell'operatore. Per muoversi utilizza due ruote laterali. Esse sono controllate da due motori DC ad impulso (in particolare 10 impulsi corrispondono al movimento di 1 millimetro del robot) che possono girare nelle due direzioni a 127 velocità diverse. Inoltre per aumentare la sua stabilità sulla superficie vi sono 2 piccoli spessori sotto la sua base. *Khepera* è dotato di 8 sensori di prossimità a raggi infrarossi, 6 di essi sono posizionati davanti, e 2 di essi dietro (benchè non esista proprio un davanti ed un dietro, dato la sua simmetricità). Questi sensori presentano la capacità di interpretare l'intensità del segnale infrarossi ricevuto con una sfumatura di 1024 valori. Il chip controllore è un Motorola 68331 con 256 kBytes di RAM e 512 kBytes di ROM. Esso si preoccupa di gestire tutti i segnali di input-output, e di comunicare attraverso la porta seriale con un computer.

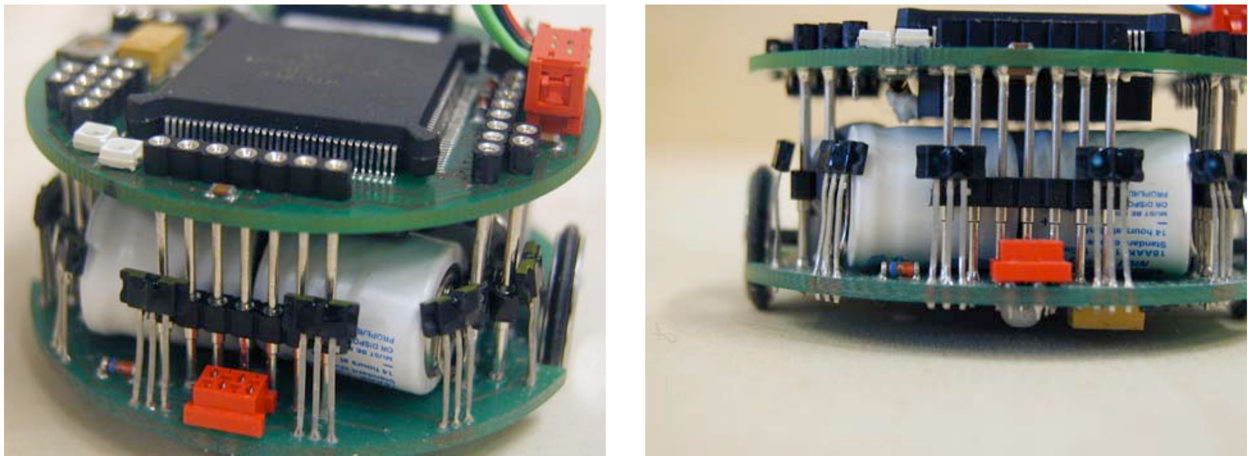


Figura 5.3: *Khepera* visto da vicino.

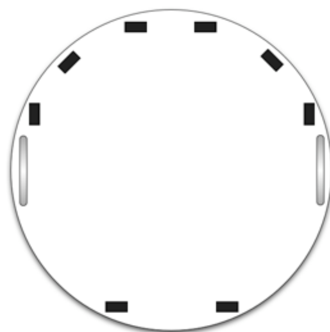


Figura 5.4: Schematizzazione di *Khepera*.

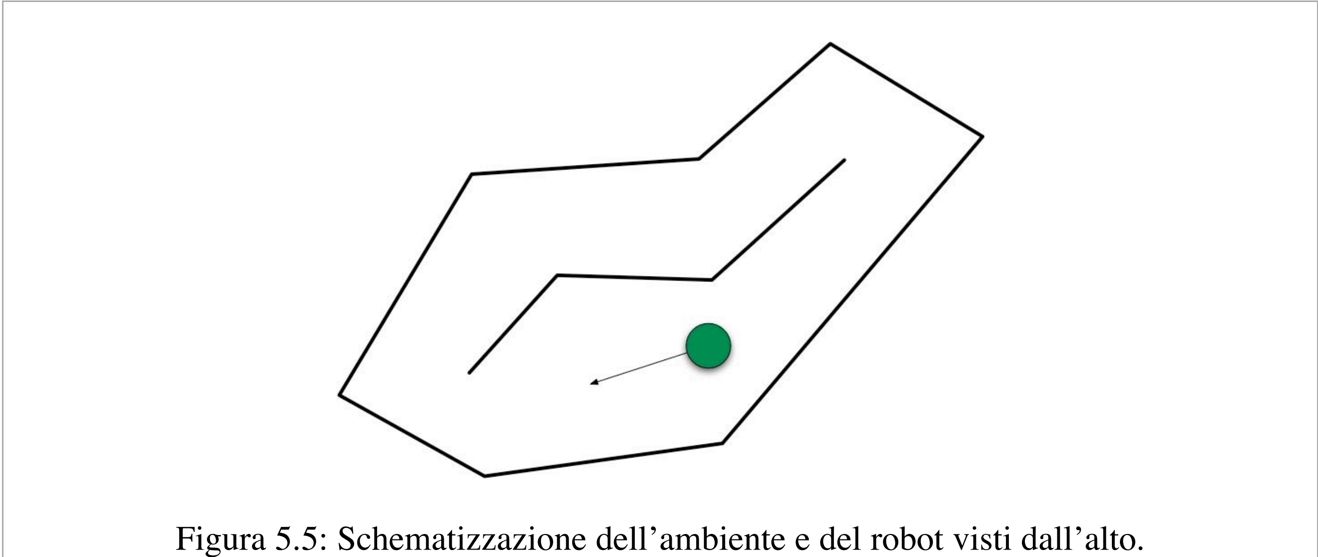


Figura 5.5: Schematizzazione dell'ambiente e del robot visti dall'alto.

La maggior parte degli altri robot è controllabile solamente scaricando un programma nel loro chip controllore. Questo presenta diversi svantaggi. Per prima cosa la potenza di calcolo è molto bassa, basti pensare che spesso questi processori non superano i 16 Mhz, inoltre anche lo spazio di memoria operativa e di quella su disco sono assai ridotte. Secondariamente ci si deve adattare al linguaggio di programmazione del controllore, che non presenta la potenza dei linguaggi orientati agli oggetti. *Khepera*, invece, presenta la possibilità di essere controllato direttamente ad un qualsiasi computer provvisto di porta seriale, grazie ad un leggero cavetto. In questo modo si può approfittare a pieno delle potenze di calcolo moderne e di poter registrare i dati dell'evoluzione in tempo reale sul disco rigido del computer. Un altro vantaggio di *Khepera* sta nella possibilità di essere alimentato sempre dallo stesso cavetto di connessione al computer, aumentando così la sua autonomia.

L'ambiente in cui *Khepera* si dovrebbe muovere è una superficie piana delimitata da dei muri che formano un *looping-maze* (labirinto circolare). Esso è riportato in figura 5.5.

5.3.2 La funzione di *fitness*

La scelta di quali parametri tenere in considerazione è molto importante e discriminante per la buona riuscita dell'esperimento. Soprattutto bisogna cercare di non dettare esplicitamente al robot in che modo deve muoversi, lasciandogli una buona autonomia per spaziare nel campo delle possibili soluzioni e quindi fornendogli la "libertà" di evolversi nel modo più naturale. La funzione di *fitness* Φ cerca di rispettare queste condizioni.

$$\Phi = V \cdot (1 - \sqrt{\Delta v}) \cdot (1 - i)$$

Prima di spiegare i termini di questa funzione, occorre precisare che le velocità delle ruote del robot vengono prima normalizzate tra -0.5 e 0.5. Una velocità negativa corrisponde ad una rotazione in un senso, una velocità positiva corrisponde ad una rotazione nel senso opposto (nel caso che sia 0 le ruote sono ferme). Anche per i sensori infrarossi viene normalizzato il loro valore entro l'intervallo 0-1. Un valore alto corrisponde ad un oggetto

molto vicino al sensore, un valore basso corrisponde a nessun oggetto nelle vicinanze. Detto questo si può procedere con la descrizione della funzione.

Il termine V rappresenta la velocità media delle due ruote, esso viene calcolato sommando i valori assoluti delle due velocità; Δv è la differenza tra le velocità delle due ruote in valore assoluto e i è il valore di attivazione del sensore di prossimità più alto. Queste tre componenti determinano rispettivamente la selezione degli individui con: la velocità massima di rotazione delle ruote (per evitare che si sviluppino organismi che stiano semplicemente fermi sul posto), traiettorie rettilinee (per evitare che si sviluppino organismi che girino su sé stessi) e la maggior abilità nell'evitamento degli ostacoli. Al fine di tenere in considerazione questi tre requisiti essi vengono moltiplicati tra di loro. Vale la pena fare ancora un commento sul termine Δv: la radice quadrata serve per non rendere troppo "aggressiva" la sola selezione degli organismi che si muovono con una traiettoria lineare, dato che la capacità di curvare è comunque necessaria per evitare gli ostacoli.

È ovvio che la funzione di *fitness* raggiungerà il valore massimo, ossia 1, solamente se il robot si trovasse su una superficie piana completamente priva di muri. Un'altra osservazione importante è che, data la forma circolare del robot e la possibilità di esso di muoversi in una direzione o nell'altra in modo analogo, la funzione Φ ha una certa simmetricità: non detta esplicitamente al robot in quale direzione muoversi. Nella figura 5.6 si può notare infatti come essa possieda due massimi.

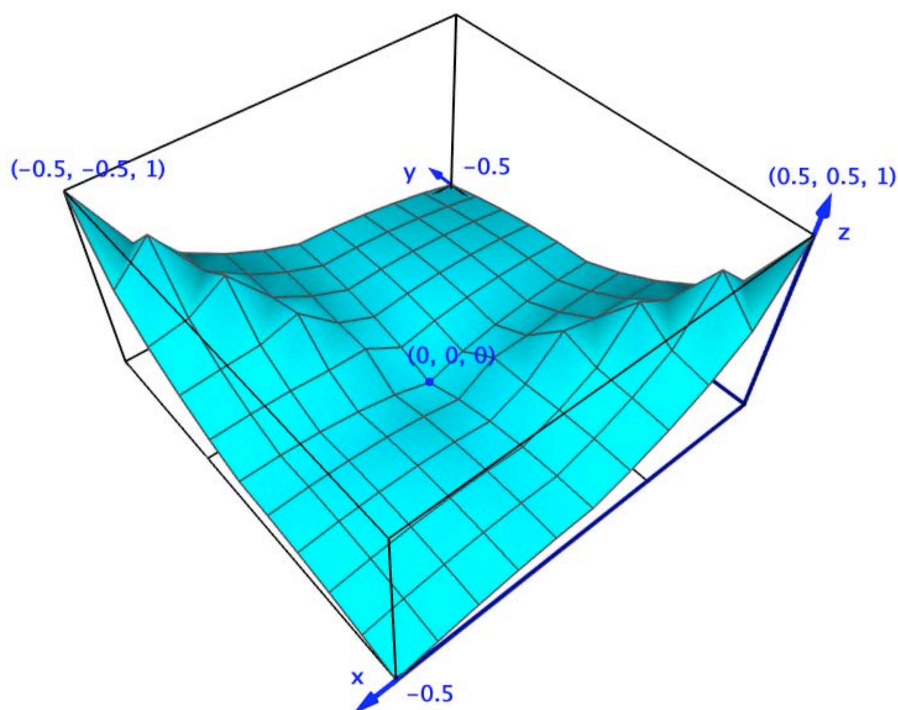


Figura 5.6: Funzione Φ in funzione della velocità della ruota sinistra (asse x) e destra (asse y). È stato scelto il parametro i come costante del valore 0.

5.3.3 L’architettura dell’organismo artificiale

Come già accennato prima, per risolvere il problema dell’evitamento degli ostacoli, permettendo al robot di evolversi, viene usato il modello degli organismi artificiali descritto nel capitolo 4. La figura 5.7 sintetizza bene l’architettura neurale del robot.

L’organismo è provvisto di un sistema neurale che ha 8 unità di input, a cui sono collegati i sensori infrarossi del robot (i cui valori sono stati normalizzati tra 0 e 1 come descritto prima). Le due unità di output della rete neurale sono ad attivazione sigmoide e determinano invece la velocità delle ruote del robot. I valori di output possibili vanno da 0 a 1 (dato l’output della funzione sigmoide), l’attivazione massima corrisponde alla velocità massima delle ruote in un senso, l’attivazione nulla corrisponde alla velocità minima delle ruote nell’altro senso, attivazione 0.5 corrisponde a velocità 0. Le unità di output inoltre possedono delle connessioni di *feedback* di sé stesse, questo per dare una memoria al sistema nervoso della velocità precedente delle ruote. Il cromosoma dell’organismo è un’array che contiene i valori dei pesi sinaptici (come numeri reali).

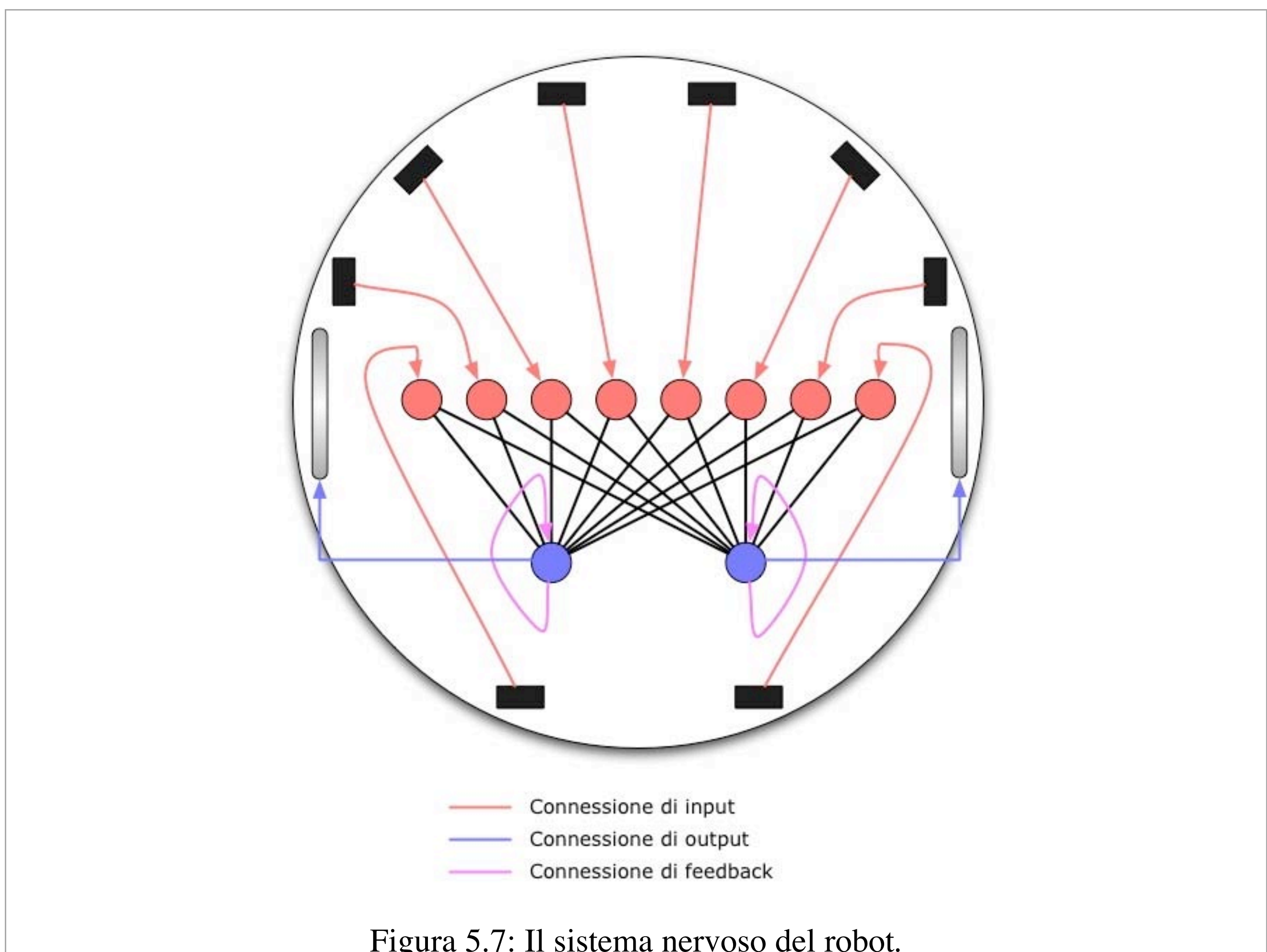


Figura 5.7: Il sistema nervoso del robot.

Il *crossover* è effettuato scambiando due elementi dell cromosoma e la probabilità che questo succeda è 0.1. La mutazione avviene aggiungendo ±0.5 al gene in questione e la probabilità è 0.2. All’inizio i geni dei cromosomi sono generati casualmente entro l’intervallo [-0.5; 0.5].

5.3.4 Implementazione e risultati

Per svolgere l'esperimento è stato scritta un'applicazione in *RealBasic* alla quale viene affidato il compito di coordinare il processo evolutivo. L'applicazione, unitamente al codice sorgente in formato *RealBasic* o testo semplice, può essere trovata nel CD-Rom allegato. L'interfaccia grafica dell'applicazione è molto semplice, si compone di un pulsante "Start Evolution" e di due etichette per vedere lo stato corrente dell'evoluzione. Prima di far partire l'evoluzione è necessarrio impostare il robot *Khepera* come scritto nell manuale del utente e collegarlo alla porta seriale numero 1 del computer.

Partito il processo, l'applicazione comincerà a comunicare con il robot secondo il protocollo descritto nell'appendice A del manuale di *Khepera*. I valori delle velocità delle ruote vengono aggiornate ogni 300 ms. La vita di un individuo dura 60 passi, ossia circa 20 s.

La struttura di questo programma risulta più semplice di quello di simulazione. Troviamo analogamente le tre classi *Population* (si occupa di effettuare la riproduzione selettiva, mutazione, *crossover*), *Organism* (rappresenta l'organismo, la sua rete neurale, il suo genoma) e *NeuralNet*.

Figura 5.8: La struttura del programma.

Anche se a prima vista la struttura può sembrare più semplice, non significa che il tempo ed il lavoro sono stati meno. Infatti, gran parte del codice risiede nell'oggetto *Window1* (la finestra dell'interfaccia grafica). Si tratta del controllo *Serial* e delle funzioni per interagire con esso. Per comunicare con il robot tramite porta seriale, si fa utilizzo di un controllo che permette la trasmissione e ricezione di dati tramite la suddetta porta. La forma dei dati da inviare ed il modo per comprendere quelli ricevuti è definito nel protocollo di comunicazione (appendice A del manuale *Khepera*). In particolare riporto le descrizioni dei due comandi di cui ho fatto uso.

D Set speed

Format of the command: D, speed_motor_left, speed_motor_right∏

Format of the response: d¶

Effect: Set the speed of the two motors. The unit is the pulse/10 ms that corresponds to 8 millimetres per second. The maximum speed is 127 pulses/ 10ms that correspond to 1m/s.

Figura 5.9: L'istruzione per impostare la velocità delle due ruote.

N **Read proximity sensors**

Format of the command: N∏

Format of the response: n,val_sens_left_90°,val_sens_left_45°,val_sens_left_10°, val_sens_right_10°,val_sens_right_45°,val_sens_right_90° ,val_sens_back_right,val_sens_back_left¶

Effect: Read the 10 bit values of the 8 proximity sensors (section 2.1.6.2), from the front sensor situated at the left of the robot, turning clockwise to the back-left sensor.

Figura 5.10: L'istruzione per richiedere il valore degli 8 sensori di prossimità.

Di seguito è riportato il codice sorgente del programma per l'evoluzione di un robot reale.

```
Window1.OpenSerialConnection:
Sub OpenSerialConnection()
  If Serial1.Open then
    CreateNewRandomPop()

    DecodeGenomes()
    TestOrganism(0)
  Else
    MsgBox "The serial port could not be opened."
  End if
End Sub

Window1.CreateNewRandomPop:
Sub CreateNewRandomPop()
  Dim TempOrganism As Organism
  Dim i As Integer

  MyPop = New Population

  For i=0 To 79
    TempOrganism = New Organism
    TempOrganism.initialize
    MyPop.Robot(i) = TempOrganism
  Next
End Sub

Window1.DecodeGenomes:
Sub DecodeGenomes()
  Dim i, k, m As Integer

  For i=0 To 79

    For k=0 To 9
      MyPop.Robot(i).MyNeuralNet.Weight(k,0) = MyPop.Robot(i).Genome(k)
    Next

    For k=0 To 9
      MyPop.Robot(i).MyNeuralNet.Weight(k,1) = MyPop.Robot(i).Genome(k+10)
    Next
```

```
    MyPop.Robot(i).MyNeuralNet.PreviousLeftWheelSpeed = 0
    MyPop.Robot(i).MyNeuralNet.PreviousRightWheelSpeed = 0
  Next
End Sub

Window1.TestOrganism:
Sub TestOrganism(Index as Integer)
  CurrentOrganismIndex = Index
  StepsDone=0
  ReadProximitySensors()
End Sub

Window1.ReadProximitySensors:
Sub ReadProximitySensors()
  Serial1.Write "N"+chr(13)+chr(10)
End Sub

Window1.SetMotorsSpeed:
Sub SetMotorsSpeed(SensorLeft90 As Double, SensorLeft45 As Double, SensorLeft10
As Double, SensorRight10 As Double, SensorRight45 As Double, SensorRight90 As
Double, SensorBackRight As Double, SensorBackLeft As Double)
  Dim ms As Double

  MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Input(0)=SensorLeft90/1024
  MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Input(1)=SensorLeft45/1024
  MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Input(2)=SensorLeft10/1024
  MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Input(3)=SensorRight10/1024
  MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Input(4)=SensorRight45/1024
  MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Input(5)=SensorRight90/1024
  MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Input(6)=SensorBackRight/1024
  MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Input(7)=SensorBackLeft/1024

MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Input(8)=MyPop.Robot(CurrentOrgani
smIndex).MyNeuralNet.PreviousLeftWheelSpeed

MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Input(9)=MyPop.Robot(CurrentOrgani
smIndex).MyNeuralNet.PreviousRightWheelSpeed

  MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.ComputeOutputs()

  MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.PreviousLeftWheelSpeed =
MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Output(0)
  MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.PreviousRightWheelSpeed =
MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Output(1)

  MyPop.Robot(CurrentOrganismIndex).ComputeFitness()

  // 300 ms delay
  ms = Microseconds * 1000
  While True
    If (Microseconds * 1000) - ms >= 300 Then
      Exit
```

```
    End if
  Wend

  Serial1.write
"D,"+str(round(MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Output(0)*254-
127))+","+str(round(MyPop.Robot(CurrentOrganismIndex).MyNeuralNet.Output(1)*254-
127))+chr(13)+chr(10)
End Sub

Window1.PushButton1.Action:
Sub Action()
  OpenSerialConnection()
End Sub

Window1.Serial1.Error:
Sub Error()
  MsgBox "Error detected."
End Sub

Window1.Serial1.DataAvailable:
Sub DataAvailable()
  Dim SensorLeft90, SensorLeft45, SensorLeft10 As Double
  Dim SensorRight90, SensorRight45, SensorRight10 As Double
  Dim SensorBackLeft, SensorBackRight As Double

  Dim Answer as String

  Answer = Serial1.ReadAll

  If left(Answer,1)="n" Then
    Answer = NthField(Answer,chr(13),1)

    SensorLeft90 = Val(NthField(Answer,",",2))
    SensorLeft45 = Val(NthField(Answer,",",3))
    SensorLeft10 = Val(NthField(Answer,",",4))
    SensorRight10 = Val(NthField(Answer,",",5))
    SensorRight45 = Val(NthField(Answer,",",6))
    SensorRight90 = Val(NthField(Answer,",",7))
    SensorBackRight = Val(NthField(Answer,",",8))
    SensorBackLeft = Val(NthField(Answer,",",9))

    SetMotorsSpeed(SensorLeft90, SensorLeft45, SensorLeft10, SensorRight10,
SensorRight45, SensorRight90, SensorBackRight, SensorBackLeft)
  elseIf left(Answer,1)="d" Then
    StepsDone = StepsDone + 1

    If StepsDone=60 Then
      If CurrentOrganismIndex=79 Then
        CurrOlbl.Text = "0"
        CurrPoplbl.Text = str(val(CurrPoplbl.Text)+1)

        MyPop.SelectiveReproduction()
        MyPop.DoCrossOver()
```

```
        MyPop.DoMutation()

        DecodeGenomes()
        TestOrganism(0)
      else
        CurrOlbl.Text = str(CurrentOrganismIndex+1)
        TestOrganism(CurrentOrganismIndex+1)
      End if
    End if
  End if
End Sub

Population.SelectiveReproduction:
Sub SelectiveReproduction()
  Dim i,k As Integer
  Dim FitnessSum, RndValue, Temp As Double
  Dim NewRobot(79) As Organism

  For i=0 To 79
    NewRobot(i) = New Organism
  Next

  For i=0 To 79
    FitnessSum = FitnessSum + Robot(i).Fitness
  Next

  For i=0 To 79
    RndValue = Rnd * FitnessSum

    Temp = 0
    For k=0 To 79
      Temp = Temp + Robot(k).Fitness
      If RndValue < Temp Then
        NewRobot(i) = Robot(i)
        Exit
      End if
    Next

  Next

  For i=0 To 79
    Robot(i) = NewRobot(i)
  Next
End Sub

Population.DoCrossOver:
Sub DoCrossOver()
  Dim i,k as integer
  Dim CutPoint as integer
  Dim TempGenome(19) as double

  For i=0 To 78 step 2
    If Rnd < 0.1 Then
```

```
      CutPoint = floor(Rnd * 20)
      For k=CutPoint To 19
        TempGenome(k) = Robot(i).Genome(k)
        Robot(i).Genome(k) = Robot(i+1).Genome(k)
        Robot(i+1).Genome(k) = TempGenome(k)
      Next
    End If
  Next
End Sub

Population.DoMutation:
Sub DoMutation()
  Dim i As integer
  Dim RndValue As Double

  For i=0 To 79
    If Rnd < 0.2 Then
      RndValue = floor(rnd*20)
      Robot(i).Genome(RndValue) = Robot(i).Genome(RndValue) + (Rnd - 0.5)
    End if
  Next
End Sub

Organism.Initialize:
Sub Initialize()
  Dim i As Integer

  MyNeuralNet = New NeuralNet

  For i=0 To 19
    Genome(i) = Rnd-0.5
  Next

  Fitness = 0
End Sub

Organism.ComputeFitness:
Sub ComputeFitness()
  Dim i As Integer
  Dim s() As Double

  For i=0 To 7
    s.append MyNeuralNet.Input(i)
  Next

  s.sort

  Fitness = Fitness + ((abs(MyNeuralNet.Output(0)-
0.5)+abs(MyNeuralNet.Output(1)-0.5))*(1-sqrt(abs(MyNeuralNet.Output(0)-
MyNeuralNet.Output(1))))*(1-s(7)))/60
End Sub

NeuralNet.ComputeOutputs:
```

```
Sub ComputeOutputs()
  Dim i, k As Integer
  Dim TotalSum As Double

  For i=0 To 1
    TotalSum = 0
    For k=0 To 9
      TotalSum = TotalSum + Input(k) * Weight(k,i)
    Next
    Output(i) = Sigmoid(TotalSum)
  Next
End Sub

NeuralNet.Sigmoid:
Function Sigmoid(x as double) As Double
  return 1/(1+exp(-x))
End Function
```

Listato 5

La prima prova di evoluzione è stata effettuata posizionando *Khepera* su una superficie delimitata da 4 muri in modo da formare un rettangolo. Purtroppo ci si è accorti sin dall'inizio che i primi organismi possiedono il "vizio" di girare su sé stessi. Questo fatto è da imputare alla scoordinazione del loro sistema senso-motorio, infatti la prima generazione viene generata con dei valori sinaptici casuali. Il cavo che collega *Khepera* al computer ha quindi cominciato ad attorcigliarsi su sé stesso e a generare una certa resistenza già dopo circa 10 s di vita dei primi organismi. Per questo motivo si è cercato di risolvere il problema facendo svolgere il cavo dopo la vita di ogni organismo. Ma questa soluzione si è rilevata troppo complessa per essere implementata poiché, a causa di diverse forze come l'attrito e l'inerzia, non era possibile tenere in considerazione in modo affidabile di quanti radianti si fosse avvolto su sé stesso il robot. Inoltre è stata scartata in partenza anche la possibilità di un assistenza da parte dell'operatore, poiché il tempo di svolgimento dell'esperimento diventerebbe troppo lungo (dato che da supposizioni il robot dovrebbe imparare a muoversi evitando gli ostacoli dopo circa 100 generazioni corrispondenti a circa 2 giorni).

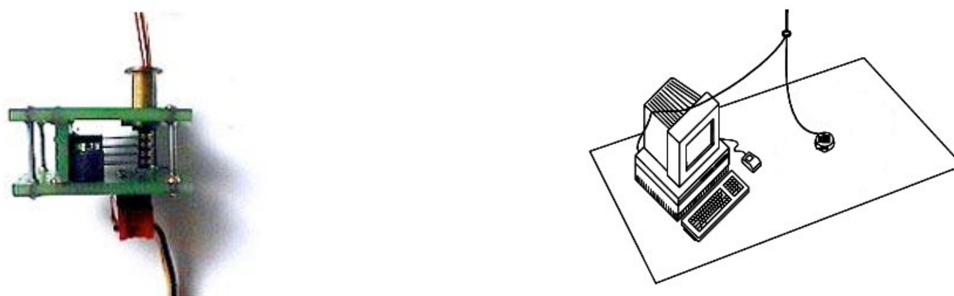


Figura 5.11: Dispositivo a contatti rotanti per impedire l'attorciliamento del cavo.

Infine sarebbe esistita la possibilità di utilizzare un dispositivo a contatti rotanti (figura 5.11) per permettere al robot di non attorcigliarsi su sé stesso. Questa soluzione, senz'altro

possibile dal punto di vista tecnico, non è stata sperimentata per il suo aspetto economico. Al momento non è stato dunque possibile avere un riscontro sperimentale della teoria svolta.

6. Indice del contenuto del CD-Rom

- Esempio TSP: programma d'esempio del *Traveling Salesman Problem*
- Manuali *Khepera*: manuale d'uso di *Khepera* e Appendice A per il protocollo di comunicazione
- Copia del testo corrente in formato *pdf*
- Robot Simulation: simulazione dell'evoluzione del robot
- EvoRobot: programma per l'evoluzione del robot reale

7. Riferimenti bibliografici

- Dario Floreano, 2002, "Evolutionary Robotics: A Gentle Introduction", http://asl.epfl.ch/research/projects/EvolutionaryRobotics/er.php
- Dario Floreano, Francesco Mondada, 2003, "Evolution of Adaptation Rules", http://asl.epfl.ch/research/projects/EvolutionOfAdaptationRules/index.php
- Tullio Tinti, "Introduzione al Connessionismo", http://www.neuroingegneria.com/art/Introduzione%20al%20Connessionismo/106.php
- Università Della Calabria, http://galileo.cincom.unical.it/corsi/psico/corsi/cognitiva/materiali/scienza%20cognitiva.pdf
- Istituto Nazionale Fisica Nucleare Sezione di Genova, http://www.ge.infn.it/web/masulli.html#UNO
- Università Di Bologna, 2004, "Le reti neurali", http://www.scienzagiovane.unibo.it/intartificiale/intart-retineuro.html
- S. de Bellis, 2001, "Reti Neurali Artificiali, Teoria delle Reti Neurali", http://www.irccsdebellis.it/html/Reti_Neurali/Teoria_delle_reti_neurali2.htm
- Marek Obitko, 1998, "Genetic Algorithms", http://cs.felk.cvut.cz/~xobitko/ga/