Computational Physics Laboratory

Final Report

Genetic Algorithms

Giorgio Ruffa

March 4, 2017

Contents

1 Introduction to Genetic Algorithms

Genetic Algorithms (GA) are a particular kind of algorithms used to solve problems where the solution space is particularly wide.

The basic idea behind GA is to consider a program as a biological entity. The building block of any biological individual is its genome. In the case of executable programs their genome is nothing else that a set of ordered(although nested) instructions. Exactly as our DNA, the program genome can be split and mixed with the one of another compatible individual, resulting in one or more new executable entities. This process of cutting and exchanging is called *Crossover*. Crossover can be very complicated, but in its simplest form, the one we used, the two "parents" individual are split in just two parts and then exchanged to generate two new programs.

Another fundamental concept is the one of *Fitness*. Moving on with the biological resemblance between GA and biology, we can attribute to every individual an index of it's *"ability to perform well"* in a specific environment. The function that associates a scalar value to any given individual is indeed called *Fitness*. The main goal of the fitness value is to rank individual for mating. Programs with high fitness will be select for crossover between each-others, granting in this way a faster convergence towards a better individual. Also the frequency of crossover could be increased for well performing individuals and, on the other hand, programs with low fitness score can be excluded from crossing over, effectively suppressing them.

We can now summarize the key steps of any GA:

1. random creation of the first generation of individuals;
2. estimate of the fitness for every individual;
3. couples formation;
4. reproduction of individuals trough crossover;
5. eventual mutation;

2 The Problem

We aim to solve an apparently simple problem: let's imagine having a set of squared wood bricks. On every one of them a letter is engraved. The whole set of bricks can be piled up in a single stack composing different words, or just left on the table. There are only eight bricks: l-a-s-r-e-v-i-n-u, so no spare or duplicated letters are provided. Given a precise starting condition (an already formed stack), we aim to find the best possible program that can form the word "u-n-i-v-e-r-s-a-l".

Every program can be encoded as a set of allowed actions divided in three categories: reading, moving and logical actions.

reading	
CS[ ]	returns the letter that sits on top of the stack
TB[ ]	returns the last correct brick on the stack
NN[ ]	returns the next letter needed to from the target word
moving	
MTT[spam]	if *spam* is the last letter on the stack, move it to the table
MTS[spam]	take the letter *spam* from the table and put it on top of the stack
logical	
DU[work,test]	execute *work* as long as *test* becomes true
EQ[expr1,expr2]	execute *expr1* and *expr2* and return true if the results are equal
NOT[expr]	if *expr* is *LispNil* return true, otherwise false

Table 1: List of all the allowed actions

3 Development and Modeling environment

3.1 Wolfram Mathematica 8

Given the kind of problem we are facing, it appears fundamental to use a programming language able to easily manipulate complex sets of operations in a flexible and intuitive way. The choice has fallen on *Wolfram Mathematica 8* [1], a lisp-like, very abstract and flexible programming language.

Let's see the reasons behind this choice.

3.1.1 Every program is an expression

Mathematica treats every statement as an expression, and every expression has a well defined structure. Taking as a reference the last example program of table 2, we can see that *Mathematica* organizes the expression in a tree-like way (see figure 1).

The tree is divided in levels, everyone of them hosts other sub-expressions. In our example the first level contains the sub-expressions MTT[CS[ ]] and MTS[NN[ ]], the second level contains CS[] and NN[]. To obtain the sub-expressions contained in every level up to the n^{th} we can use the *Level[expr,{n}]* command. In our case the command *Level[individual,2]* returns {CS[], MTT[CS[]], NN[], MTS[NN[]]}, while *Level*{*individual,Infinity*} returns all the sub-expressions contained in the whole individual.

Every expression (or sub-expression) has a so-called *Head*, which, in our case is the name of the function (arguments are discarded). In our case the head of the individual is EQ and we can obtain it with the command *Head[individual]*.

Every sub-expression has a well defined position that can be used to access and modify the whole expression. The command *Position[expr,subexpr]* gives you the possibility to *"walk"* trough the tree to reach the desired sub-expression.

Keeping our example under consideration, we will have that *Position[individual,CS[]]* returns {2,1}. The first number tells us that CS[]'s parent is the second argument of the first level, while the second number means that CS[] is the first argument of its parent.

In the case a sub-expression is contained in multiple parts of the tree, the command *Position* will return a list of positions.

We can use the obtained position to extract (command *Extract*) and modify (command *ReplacePart*) the sub-expression, while leaving the rest of the three intact.

3.1.2 Variables, Functions and Parameters

Mathematica is not statically typed, meaning that variables can take any value that can be changed in any moment. Moreover the language prefer to avoid explicit evaluations of expressions as much as possible. This feature comes in handy when one must introduce new parameters for the fitness calculations or when creating list of

MTT[CS[ ]]	sposta l'ultima lettera della stack sul tavolo
DU[MTS[NN[ ]],NOT[NN[ ]]]	sposta la prossima lettera sulla stack finche' non hai terminato la parola cercata.
EQ[MTT[CS[ ]] ,MTS[NN[ ]]	prima muovi l'ultima lettera della stack sul tavolo e poi prendi la prossima lettera necessaria del tavolo e mettila sulla stack

Table 2: esempi di individui

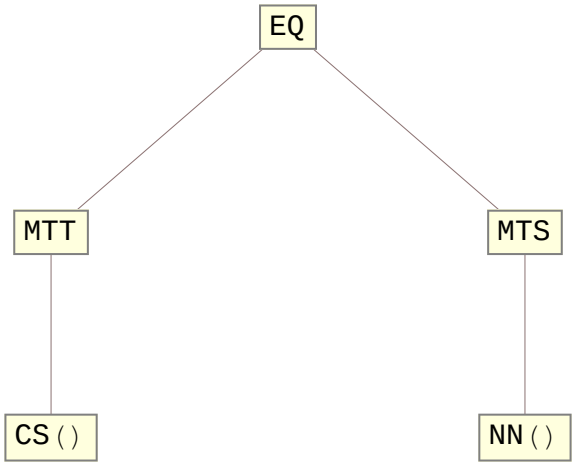


Figure 1: three-like structure of an expression (command *TreeForm*)

records about evolutionary trends of the population. The generation of visualization and profiling tools is also very easy within this paradigm.

A statically typed language, even if object oriented, would have requested a great effort to introduce these feature, not to mentions the increase of iterations needed to make the whole design sound.

In this framework it is convenient to introduce the variables used in the package:

the stack and the table (Stack and Table from now on) are two lists that can be modified only by the aforementioned functions MTT and MTS.

In order to simplify the implementation the stack is read from top to bottom, hence the target word (TargetWord) is equal to {l,a,s,e,r,v,i,n,u};

The fitness function will evaluate the individual and compare the resulting stack with the target word. It will eventually take under consideration also other aspects of the individual, for instance his efficiency as number of executed operations.

4 Modules

Since we cleared out how *Mathematica* models its expressions we can now proceed to evaluate the modules the package is made of. Five modules have been developed, each with its own purpose and functions. In this section you will find a brief attempt to documenting them. [1].

4.1 Main functions

The functions and variables mentioned below are contained in the file *moves.m*

[1]Comments are still in italian

4.1.1 Variables

TargetWord: {l,a,s,r,e,v,i,n,u} the word we are attempting to obtain

MyStack: the stack, list.

MyTable: the table, list.

maxdustep: maximum number of executions of *work* after which DU evaluations is aborted. When this happens the DU is considered *Broken*.

BrokenDU: number of broken DUs.

StepCount: number of steps for individual. A step occurs when a letter is moved.

LispNil: to be consistent with the literature [2] some functions can return this variable, which, now, is just a string.

Listing 1: variables in moves.m

```
TargetWord = {l,a,s,r,e,v,i,n,u};
MyStack = {};
MyTable = {};

LispNil = "lisp nil";
maxdustep = 2 * Length[TargetWord] ;
BrokenDU = 0;
StepCount = 0;
MAXDU = 5;
```

4.1.2 Functions

function:	CS[ ]
complete name:	current stack
behavior:	returns the top letter on the stack, if the stack is empty returns ListNil
return:	a letter or LispNil
function:	TB[ ]
complete name:	top correct block
behavior:	the last correct letter on the stack or LispNil if none is correct.
esempio:	MyStack = {l,a,n,i} ; In:TB[ ] ; Out: a
return:	a letter or LispNil
functions:	NN[ ]
complete name:	next needed
behavior:	return the next letter needed to compose the TargetWord. If the stack is already the equal to the TargetWord LispNil will be returned
return:	a letter or LispNil

functions:	MTS[*spam*]
complete name:	move to the stack
behavior:	move the letter *spam* from the table to the top of the stack (i.e. appended to MyStack) and return the moved letter. If spam *spam* is not on the table, just returns it.
return:	always a letter
functions:	MTT[*spam*]
complete name:	move to the table
behavior:	if the letter *spam* in on top of the stack (i.e. the last element of MyStack) move it to the table and return *spam*. if *spam* is not on top of the stack, do nothing and return *spam*
return:	always a letter
functions:	NOT[*expr*]
complete name:	not
behavior:	if expr is LispNil returns True, otherwise False
return:	True or False
functions:	EQ[*expr1 , expr2*]
complete name:	equal
behavior:	evaluate expr1 and then expr2, then returns True if the two results are the same, otherwise false.
return:	True o False
functions:	DU[*work , test*]
complete name:	do untill
behavior:	evaluate *work* until *test* does not return True. Return LispNil if *work* is evaluated more then maxdustep, otherwise Null.
return:	Null or LispNil

Table 3: Allowed functions

Listing 2: main functions implementation

```
TB[] := Module[{i=0, c , tab , first , splitted} ,
        tab =   Table[ MyStack[[i]] === TargetWord[[i]] ,
                        {i,Length[MyStack] }
                ];
        splitted = Split[tab];
        If[ Length[splitted ] == 0,
            Return[LispNil]      ,
            first = splitted[[1]];
        ];
        c = If[MemberQ[first,True] ,
```

```
                Length[first],
                0
            ];
        If[ c == 0,
            Return[LispNil] ,
            Return[MyStack[[c]]]
        ];
];

MTT[x_] := Module[{},
    StepCount ++;
    If[ Length[MyStack] == 0 , Return[x] , Null];

    If[ Last[MyStack] == x ,
        (* togli l'ultimo e solo l'ultimo *)
        MyStack = Delete[MyStack , Length[MyStack]];
            MyTable = Append[MyTable , x] ;
            Return[x]    ,
        (* se non e' l'ultimo della stack non fare niente*)
        Return[x] ;
    ];
];

DU[work_, test_] := Module[{tmp,counter = 0},
    If[BrokenDU >= MAXDU,
        Return[Null],
        Null
    ];
    While[  test ==False   ,
        (*forza valutazione con assegnazione*)
        tmp = work ;
        counter ++ ;
        If[counter -1 == maxdustep ,
            BrokenDU ++;
            Return[LispNil] ,
            Null
        ];
    ];
];

SetAttributes[DU,HoldAll];
```

4.1.3 Note Tecniche

As you can see in the listing, the DU functions is a mere wrapper around a *While* loop with the *HoldAll* attribute set. This attribute grants that the expression *test* is not evaluated before DU invocation, instead *test* is passed as an expression (not a value), hence re-computed for every loop iteration. Without the *HoldAll* attribute the loop will most likely run indefinitely, as long as *test* is false before applying *work* for the first time. [2]

4.2 Creation of the Individual

questa sezione analizza il contenuto del file *firstgen.m* che contiene il primo step di ogni algoritmo genetico: la generazione di una prima popolazione di individui.

[2] The *HoldAll* attribute forced me to understand more deeply how *Mathematica* evaluates and manages functions and expressions.

4.2.1 correttezza sintattica

Normalmente la generazione viene fatta casualmente ma in questo particolare caso e' stato scelto di formare individui che siano sempre sintatticamente corretti. ovvero costrutti del tipo CS[MTT[ ]] che non hanno alcun senso dal punto di vista sintattico non devono essere presenti.

Il modo in cui ho affrontato il problema e' il seguente: presa una delle 8 funzioni (vedi tabella 3 a pag 7) essa potra' avere solo un numero limitato di argomenti possibili. Per esempio MTT potra' avere come argomento solo quelle funzioni che restituiscono una lettera, altrimenti la funzione non e' definita e non verra' eseguita da Mathematica;

per poter guadagnare in flessibilita' ho scelto di utilizzare delle liste di argomenti permessi. Ad ogni funzione e' associata una lista dalla quale verra' estratta una funzione corretta. In questo modo posso cambiare in qualsiasi momento le regole sintattiche e questo e' un vantaggio da non sottovalutare.

in seguito e' allegata l'implementazione:

Listing 3: liste compatibilita'

```
EQList = {  fEQ[e1,e2] , fNOT[n] , fDU[duw,dut] , fMTT[mt] , fMTS[ms] ,
            fCS[] , fTB[] , fNN[] };

NOTList = { fCS[] , fTB[] , fNN[] };

DUWList = { fEQ[e1,e2] , fNOT[n] , fDU[duw,dut] , fMTT[mt] , fMTS[ms] ,
            fCS[] , fTB[] , fNN[] };

DUTList = { fEQ[e1,e2] , fNOT[n] };

MTTList = {  fCS[] , fTB[] , fNN[] };

MTSList = {  fCS[] , fTB[] , fNN[] };

CSList = NNList = TBList = {};
```

come si puo' notare per il DU appaioni due liste: DUW e DUT , che stanno rispettivamente per DUWork e DUTest. essenzialmente il primo e il secondo argomento di DU differiscono per regole sintattiche, il *test* dovra' essere una funzione che restituisce o True o False mentre il *Work* puo' essere qualsiasi funzione.

la presenza di variabili in caratteri minuscoli (come *e1,e2,n, ...*), viene spiegata nella sezione 4.2.3.

4.2.2 valutazione ritardata

come si vede nel listato 3 le funzioni appiono con una *f* come prima lettera. Se non ci fosse questa lettera le funzioni verrebbero valutate subito ad ogni loro ricorrenza nel codice, metre l'obbiettivo e' di valutarle tramite la chiamata di funzioni apposite che effettueranno la sostituzione con le funzioni vere e proprie. Per quanto possa sembrare artificiosi in realta' si sta' semplicemento dando un nome diverso alle funzioni, un nome che non ha alcuna definizione implementata. [3]

[3]per maggiori informazioni si veda la sezione 4.3 a pagina 11

4.2.3 generazione ricorsiva

Per sfruttare al meglio la struttura ad albero esposta nella sezione 3.1.1 ho deciso di generare gli individui con un algoritmo ricorsivo che percorra tutti i "rami" dell'albero. In sostanza l'algoritmo consiste nel generare una prima Head a caso dalla lista EQList[4] e poi chiamare una funzione che data la head del "padre" e' in grado di scegliere un argomento a caso dalla lista di compatibilita' del padre e richiamare se stessa sugli argomenti "figli".

La funzione che implementa questo algoritmo e' *HAC* [5] (si veda listato 4).

La prima definizione viene usata la prima volta. Chiamando il comando *HAC[]* la funzione sceglie una Head random che sara' per esempio *fEQ[e1,e2]*, a questo punto appare ovvia l'importanza delle variabili *e1 e2* menzionate in 4.2.1: tramite la lista di sostituzioni chiamata *sostituzioni* mi permettono di chiamare la funzione *HAC[fathhead_]* con la Head del "padre" corretta. Dopo la prima chiamata di HAC abbiamo un espressione del tipo fEQ[HAC[fEQ1] , HAC[fEQ2]].

A questo punto viene eseguita la seconda funzione del listato (per esempio HAC[fEQ1]), che sceglie a sua volta un argomento che puo' stare a primo membro di fEQ e richiama se stessa sugli argomenti ottenuti in modo analogo a come descritto prima. la scelta dell'argomento viene effettuata con la funzione ChooseArg (il codile e' nella sezione 9).

Come in ogni algoritmo ricorsivo sono necessarie delle condizioni di uscita senza delle quali la generazione si arresterebbe nel caso in cui tutte le chiamate residue di *HAC[fathhead_]* resituissero NN[] CS[] o TB[]. Questo potrebbe portare ad una generazione iniziale di individui troppo complessi per i nostri scopi, di conseguenza viene messo un limite al numero di volte che HAC viene chiamato per singolo individuo e al numero di foglie che l'individuo possiede [6]. E' la funzione *RigthInd[]* che genera un individuo sempre corretto premurandosi di controllare anche che il numero di foglie non superi un certo limite. La definizione di questi limiti a livello di implementazione e' trattata nella sezione seguente.

4.2.4 variabili

all'interno del file firstgen.m sono presenti le seguenti variabili:

Nind : numero di individui che compongono la prima generazione (utilizzata per testare il singolo file);

MaxHacCall : numero massimo di chiamate di HAC oltre le quali la generazione dell'individuo si arresta;

CurrHacCall : attuale numero di chiamate di HAC (utilizzata per testare il singolo file);

MinLeaf : numero minimo di foglie che un individuo deve avere;

ManLeaf : numero massimo di foglie che un individuo deve avere;

[4] EQList contiene tutte le funzioni

[5] Head A Caso

[6] si veda comando LeafCount

Listing 4: la funzione HAC per la generazione ricorsiva degli individui

```
sostituzioni := {
    e1  -> xHAC[fEQ1]    ,
    e2  -> xHAC[fEQ2]    ,
    n   -> xHAC[fNOT]    ,
    duw -> xHAC[fDUW]    ,
    dut -> xHAC[fDUT]    ,
    mt  -> xHAC[fMTT]    ,
    ms  -> xHAC[fMTS]
};

HAC[]:=Module[{randhead , sostitutiprova},
    randhead = RandomChoice[EQList];
    (*per esempio randhead = fEQ[e1,e2]*)

    sostituitiprova = randhead /. sostituzioni ;
    (*sostituti prova diventa fEQ[xHAC[fEQ1],xHAC[fEQ2]]*)

    (*a questo punto xHAC viene sostituita con HAC e *)
    (*la funzione viene effettivamente chiamata*)
    (*l'espressione ritornata sara' del tipo *)
    (*fEQ[HAC[EQ1],HAC[EQ2]]*)

    Return[ sostituitiprova /. { xHAC -> HAC} ];
];

HAC[fathhead_] := Module [{args , sostituiti , sostitutiprova} ,

    (*controlla il numero di chiamate totale*)
    If[ CurrHacCall >= MaxHacCall,
        Print["max hac calls raggiunto!"];
        Return[Null],
        Null
    ];

    CurrHacCall ++;

    (*scegli un argomento compatibile con la head del padre*)
    args = ChooseArg[fathhead];
    sostituitiprova = args /. sostituzioni ;
    Return[ sostituitiprova /. { xHAC -> HAC} ];
];
```

4.3 Meccanismo di Valutazione

La valutazione di un individuo e' una parte fondamentale in quanto ricorre Nind volte per generazione. Non deve solo valutare l'individuo ma generare dei dati che verranno passati alla fitness per valutare quanto l'individuo si sia comportato correttamente. Infine deve essere molto semplice da utilizzare, deve quindi avere un'interfaccia pulita. Con Mathematica tutto questo puo' essere fatto in una trentina di righe di codice in una singola funzione, come possiamo vedere dal listato 6.

La funzione *EvalInd[ind_]* valuta un individuo e restituisce la Stack ottenuta alla fine della valutazione, il numero di BrokenDU occorsi, il numero di Step eseguiti [7] , e la Richness dell'individuo. [8]

La funzione *Block* forza le variabili contenute nella lista ad avere il valore che gli viene assegnato localmente, in questo modo per ogni individuo la Stack e la Table vengono riportate alla configurazione iniziale e poi restituite dalla funzione

[7] per il significato di queste variabili si veda 4.1.1

[8] la richness e' spiegata nella sezione 4.5

per essere analizzate dalla fitness.

Listing 5: funzione che valuta un individuo

```
finalsost = {   fEQ      :>  EQ   ,
                fNOT     :>  NOT  ,
                fDU      :>  DU   ,
                fMTT     :>  MTT  ,
                fMTS     :>  MTS  ,
                fNN      :>  NN   ,
                fCS      :>  CS   ,
                fTB      :>  TB
};

DefaultStack = {u,n,i,v} ;
DefaultTable = {e,r,s,a,l};

EvalInd[ind_] := Block[{     TargetWord = {l,a,s,r,e,v,i,n,u},
                             MyStack = DefaultStack,
                             MyTable = DefaultTable,
                             BrokenDU = 0 ,
                             StepCount = 0,
                             MAXDU = 5,
                             newind,
                             rich = 0
                      } ,

    newind = ind /. finalsost;
    rich = Richness[ind];

    Return[{MyStack , MyTable , BrokenDU, StepCount , rich}];
];
```

4.4 Crossover

il crossover e' la parte che piu' caratterizza un algoritmo genertico e in questo Package e' una delle fasi tecnicamente piu' complesse. esso deve mantenere la correttezza sintattica degli individui e garanetire che lo scambio venga eseguito sempre (qualora la correttezza sintattica lo permettesse).

questa sezione analizza la funzione *CrossOver* presente nel file *crossover.m*.

4.4.1 garantire la correttezza sintattica

In sommi capi il CrossOver deve tagliare due individui in due parti e scambiarle. Se in generale questa e' un operazione semplice, nel nostro caso il fatto di dover garantire la correttezza sintattica complica le cose.

Chiariamo cosa si intenda per "punto di taglio": ogni sottoespressione[9] presente in un individuo sara' un argomento di una delle otto funzioni fondamentali e, se l'individuo e' stato generato correttamente, sara' nella sua lista di compatibilita'. [10]

un punto di taglio sara' quindi una coppia costituita da una sottoespressione e dalla Head della funzione di cui la sottoespressione e' argomento.

per comodita' la sottoespressione del primo individuo sara' chiamata "Ramo1" e la funzione di cui Ramo1 e' argomento la chiameromo HeadPadre1. Ramo2 sara'

[9] si veda sezione 3.1.1

[10] si veda listato 3 pagina 9

analogamente la sottoespressione del secondo individuo , e HeadPadre2 la funzione di cui Ramo2 e' argomento. [11]

Per garantire la correttezza sintattica dovranno essere presenti contemporaneamente due condizioni: Ramo2 deve essere compatibile con HeadPadre1 e Ramo1 dovra' essere compatibile con HeadPadre2. bisognera' quindi scegliere una coppia di punti di taglio che permetta il CrossOver.

4.4.2 garantire lo scambio

Ad un primo tentativo il crossover puo' non trovare una coppia di punti di taglio corretti. Nelle prime generazioni questo fenomeno non e' trascurabile e porta a delle modificazioni sensibili della probabilita' di CrossOver. Per ovviare a questa situazione l'algoritmo dovra' cercare di considerare tutti i punti di taglio possibili finche' non trovera' una coppia adatta.

4.4.3 l'algoritmo

La funzione CrossOver[] opera i seguenti passaggi:

1. genera una lista di tutte le sottoespressioni dellindividuo 1 e la mescola in maniera random;
2. partendo dalla prima genera una lista di Ramo2 compatibili con HeadPadre1, che chiameremo "Candidati".
3. riordina Candidati casualmente
4. passa in sequenza i membri di Candidati finche' HeadPadre2 non e' compatibile con Ramo1
5. se hai trovato una buona coppia di punti di taglio effettua lo scambio, altrimenti restituisci gli individui intatti ed incrementa un contatore.

questa procedura ha permesso di minimizzare i casi di fallimento del crossover nelle prime generazioni da circa 10/100 a circa 1/100.[12]

4.4.4 variabili

le due variabili presenti nel file sono PartialCrossFail , TotalCrossFail. la prima viene utilizzata per conteggiare quanti crossover falliscono perche' l'individuo 1 non ha candidati compatibili per il taglio, mentre il secondo conteggia i crossover falliti perche' nessuna HeadPadre2 dei candidati e' compatibile con nessun Ramo1.

[11] TODO: ESEMPI CON FIGURE

[12] test effettuati su popolazioni di 200 individui alla prima generazione

4.4.5 note tecnice

data la lunghezza l'algoritmo e' riportato bella sezione 9. Le funzioni di *Mathematica* fondamentali in questo algoritmo sono:

- il comando Positon[ind,subexpr] gia' menzionato nella sezione 3.1.1;
- il comando Extract[ind,position] che restituisce la sottoespressione che si trova nella posizione *position* dell'individuo *ind*
- il comando ReplacePart[ind,position -> expr] che permette di sostituire all'interno dell'individuo la sottoespressione che si trova in *position* con l'espressione *expr*
- il comando RandomSample[list] che riordina casualmente una lista *list*
- il comando RandomChoice[list] che restituice un elemento random della lista *list*

si consiglia la retturna della reference per le funzioni sopracitate.

4.4.6 considerazioni

il CrossOver in questo caso non conserva la lunghezza degli individui che tenderanno ad aumentare per profondita' e complessita' con l'andare delle generazioni. [13]

4.5 Fitness

La Fitness e' sicuramente la parte decisiva nel succeso di un algoritmo genetico e modellizzando quello che e' l'ambiente per gli esseri viventi.

La scelta fatta nello scrivere la Fitness e' stata di garantire una certa modularita'.

La modellizzazione e' la seguente: ogni caratteristica di un induviudo viene valutata con un punteggio. ogni punteggio ha un peso e la somma dei punteggi pesati costituisce il voto finale dell'individuo.

le caratteristiche valutate sono:

RigthLetters	un punto per ogni lettera implata correttamente sulla stack
DidSome	un punto per ogni lettera che differisce dalla stack iniziale piu' un punto per ogni differenza di dimensione tra stack iniziale e finale
Richness	un punto per ogni ognuna delle otto funzioni base presente nell'individuo
BrokenDU	un punto per ogni BrokenDU
Step	un punto per ogni step fatto

Table 4: aspetti valutati dalla fitness

[13] si veda 6

I punteggi ci danno una sorta di fotografia comportamentale dell'individuo, ma la vera selettivita' consiste nel peso che viene affidato ad ogni singolo punteggio. Possiamo decidere di premiare individui particolarmente attivi o ricchi di funzioni base, oppure possiamo premiare individui che fanno un numero di step prestabilito o un numero di BrokenDu piccolo.

Per esempi e risposta della popolazione a particolari tipi di selezione si veda la sezione6

4.5.1 implementazione

La fitness e' una funzione che si applica al singolo individuo e che per le sopracitate scelte di modularita' e' stata divisa in step diversi.

1. la valutazione degli individui produce una lista di risultati descritta in 4.3 a pagina 11

2. il risultato della valutazione viene passato alla funzione FitnessParameters che si occupa di applicare le funzione di valutazione in tabella 4 e di resituire i punteggi in modo ordinato. [14]

3. infine i punteggi calcolati da FitnessParameters vengo dati in pasto alla funzione Fitness che li elabora secondo i pesi scelti ed eventali altri aspetti

il vantaggio di questa procedura consiste nel poter raccogliere piu' facilmente le valutazioni degli individui per costruire strumenti diagnosctici con estrema flessibilita'. [15]

4.5.2 considerazioni

E' stato scelto di non dare punteggi negativi ma premiare di meno individui che non hanno certe caratteristiche. In questo modo e' possibile registrare un trend della fitness media ed avere quindi un punto di riferimento sulla convergenza della popolazione.

Ho scelto di non valutare caratteristiche "fisiche" degli individui, come per esempio il numero di foglie o la profondita'[16], ma di valutare il solo comportamento degli individui, questo non ha comunque precluso la possibilita' di trovare risultati significativi.

4.6 Tools

la parte finale dell'implementazione e' stata dedicata a costrure degli strumenti che permettessero di analizzare velocemente e in maniera semplice le caratteristiche di una generazione e del suo sviluppo al passare delle iterazioni dell'algoritmo.

[14] i BrokenDu e gli Step vengono calcolati nel momento della valutazione dell'individuo, nessuna funzione viene appliacata di FitnessParameters ma vengono semplicementi copiati all'interno della lista dei risultati

[15] si veda sezione 4.6

[16] si vedano comandi LeafCount e Depth

Anche in questo caso *Mathematica* ha offerto un insieme di strumenti che hanno reso snella l'implementazione.

Per permettere questo genere di analisi le caratteristiche, come per esempio le singole stack degli individui alla fine della valutazione o le singole fitness, vengono salvate in apposite liste che vengono riempite dalla funzione Profiler che si trova nel file *fitness.m*, per poi essere lette dalle funzioni presenti in *tools.m*.

Ecco l'elenco delle funzioni e il loro risultato:

GenGraphs[n_]: stampa una griglia di quattro grafici rappresentanti la fitness, la distribuzione della fitness, il numero di foglie di ogni individuo e la profondita' di ogni individuo;

PlotTrends[]: stampa una griglia di quattro grafici rappresentanti i valori che le seguenti grandezze hanno assunto nelle generazioni precedenti: fitness media, devianzione standard della fitness , foglie medie e profondita' media.

popgiusti[]: resituisce una lista di coppie {numero generazione , numero individui giusti}, dove per individui giusti si intende quelli che formano la TargetWord a prescindere dalle prestazioni.

perfetti[]: resituisce una lista di coppie {numero generazione , numero individui perfetti}, dove per individui perfetti si intendono quegli individui uguali all'individuo perfetto presente in letteratura; [17]

PR[]: fai un profilo della simulazione effettuata, stampando l'output di popgiust[], i paramentri usati nella simulazione e il numero di generazioni.

ProfileRun[filename_]: stampa su file l'output di popgiusti[] , perfetti[], PlotTrends[], i parametri usati e il numero di generazioni.

salvaindividuo[ind_,filename_]: salva l'individuo su un file.

F2D[list_]: data una lista restituisce una lista di coppie {valore, numero di ricorrenze di valore in list}

ProfileGeneration[n_]: data la generazione n salva gli individui che formano la TargetWord nella variabile *giusti* e il prodotto delle loro valutazioni in *valgiusti*

un uso approfondito di questi strumenti e' stato fatto nella sezione 6

5 Utilizzo del Package

in questa sezione troverete come utilizare il Package per generare una popolazione, farla evolvere e analizzare il risultato.

come prima cosa occorre generare una popolazione, settiamo quindi il numero di individui e generiamo la prima popolazione:

[17] si veda 6

Listing 6: funzione che valuta un individuo

```
PM = 0.001;
PC = 0.7;
Nind = 100;
pop = FirstGeneration[];
```

ora abbiamo la nostra popolazione di individui formata. Per analizzarla preliminarmente bastera' dare il comando

```
ListPlot[ F2D[ Map[LeafCount,pop] ] ];
```

ottenedo come output qualcosa di simile alla figura 2

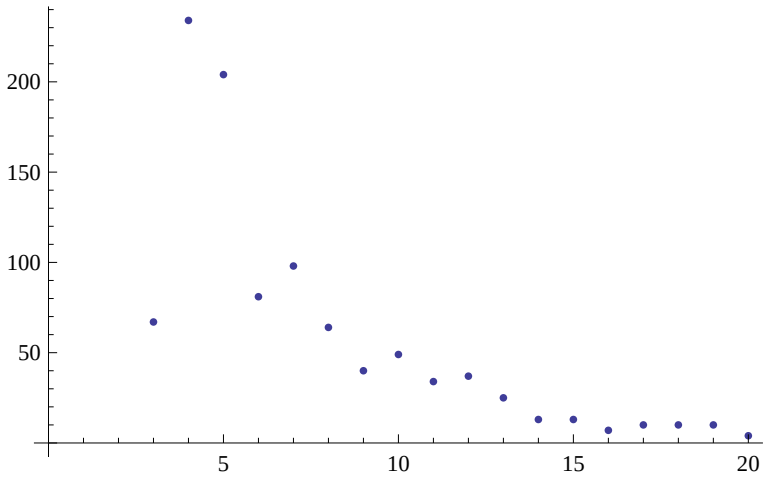


Figure 2: numero di individui con x foglie

ora possiamo fare evolvere la nostra popolazione di uno step con il comando

```
pop2 = StepGen[pop]
```

la funzione StepGen[popolazione] esegue tutti i passi di un algoritmo genetico esposti a pag 2, meno che la mutazione e restituisce la popolazione elaborata.

volendo far eseguire piu' generazioni una routine di questo tipo permettera' anche di salvare ogni generazione:

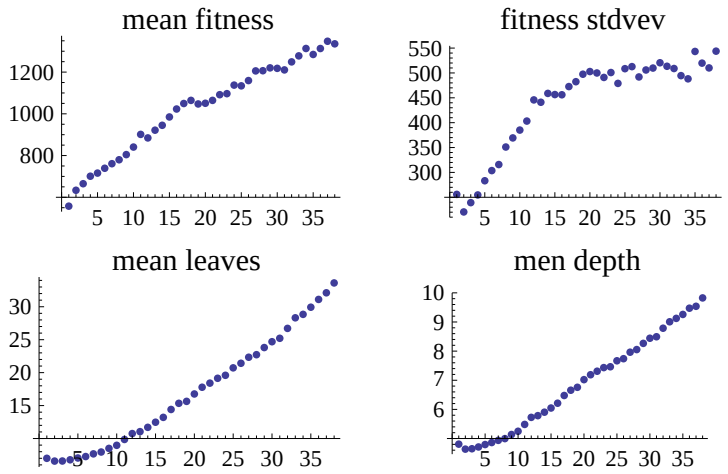
```
gennumebr = 20;
pops = {pop,pop2}
prova[]:= For[incr=3,incr<gennumber,incr++,
                Print["GEN: " , incr];
                gens =  Append[ pops,
                                StepGen[ pops [[incr-1]]
                                ]
                        ]
];
```

infine potremo avere un analisi di insieme della nostra popolazione con il comando PR[] e salvare i risultati con il comando ProfileRun[filenale_]:

```
PR[]
ProfileRun["rundiprova.m"]
```

se siete nel notebook otterrete un output del tipo (figura 5):

```
popolazione totale:
1000
parametri:
{rlw,dsw,riw,bdw,stw,msl,dsl,mdl,ddl}
{250, 1, 5, 500, 1000, 0, 2, 13, 5}
popolazione di elementi perfetti:
{}
popolazione di elementi giusti:
{{5, 1}, {18, 1}, {27, 1}, {28, 1}, {29, 2}, {30, 1}, {31, 2}, {32, 1},
 {35, 1}}
numero runs:
40
MAXDU:
5
MAXSTEP:
30
```



per conoscere meglio come utilizzare il Package si consiglia una lettura della sezione seguente.

6 Testing e Applicazioni

questa sezione vuole rappresentare una scpecie di “quaderno di laboratorio” che ha portato ad individuare l’effetto di determinati set di paramentri e regole di selezione sulla qualita’ degli individui. Per idee su possibili sviluppi e miglioramente si veda la sezione 7

6.1 l’individuo perfetto

come viene indicato in letteratura [2], esiste la soluzione perfetta al nostro problema, cioe’ un individuo che ricostrutisce la parala cercata nel minor numero possibile di mosse.

EQ[DU[MTT[CS[]] ,NOT[CS[]]] ,DU[MTS[NN[]] , NOT[NN[]]]]

Table 5: l'individuo "perfetto"

come possiamo vedere il primo argomento di EQ e' un DU che svuota completamente la stack, mentre il secondo la riempie con le lettere in ordine corretto.

Le caratteristiche di questo individuo sono le seguenti:

- un numero di foglie pari a 11
- una profondita' pari a 5
- un numero di step pari a 13
- nessun broken du

La ricerca della giusta combinazione di parametri che permettessero a questo individuo di apparire e di riprodursi con successo e' stata la parte finale dell'esperienza di laboratorio. Questo ha portato allo sviluppo di molti dei tools esposti nella sezione 4.6

6.2 Condizioni Iniziali

Per condizioni iniziale si intende la composizione iniziale di Stack e Table. Non solo e' essenziale che tutti gli individui partano dalla stessa configurazione, ma anche che sia identica per tutte le simulazioni. in questo modo il successo di determinati set di parametri non e' riconduciblie a configuarazione particolarmente fortunate della Stack.

Di conseguenza la funzione di valutazione (si veda 4.3) utilizza due variabili, DefaultStack e DefaultTable, che rappresentano rispettivamnte il valore iniziale della Stack e del Tavolo.

Il valore scelto per queste simulazioni e' stato:

```
DefaultStack = {u,n,i,v} ;
DefaultTable = {e,r,s,a,l};
```

si ricordi che la stack corretta non e' {u,n,i,v,e,r,s,a,l}, ma {l,a,s,e,r,v,i,n,u};

6.3 Parametri e funzione Lol

per parametri si intendono i pesi che vengono dati ai punteggi indicaticati nella sezione 4.5 , che sono presenti nella funzione *Fitness*. Si veda il listato sottostante.

```
    rlw =   1000;   (*right letter weight*)
    dsw =   1;      (*do something weight*)
    riw =   5;      (*richness weight*)
    bdw =   10;     (*broken DU weight*)
    stw =   10;     (*step weight*)
```

Inoltre e' stata sviluppata anche una funzione chiamata *Lol[media_ , sigma_ , x_]* (presente in tools.m) , che data una gaussiana centrata su *media* e sigma uguale a *sigma* il cui valore massimo vale 1 restituisce il valore della funzione in *x*.

essa e' stata utilizzata per premiare quegli individui che avevano un valore di Step piu' o meno vicino ad un valore medio, e un valore di BrokenDU vicino a zero. infatti, qualora venga fatto l'utilizzo di *Lol* nella funzione *fitness*, troverete anche le seguenti variabili:

```
msl = 13;    (*mean step lol*)
dsl = 5;     (*delta step lol*)
mdl = 0;     (*mean broken du lol*)
ddl = 2;     (*delta broken du lol*)
```

la funzione *lol* restituisce sempre un valore tra 0 e 1, quindi modula il punteggio dato dai pesi bdw e stw.

6.4 Primi test sui parametri

la parte iniziale dell'esprerienza e' stata dedicata a primi test indicativi sull'effetto dei parametri.

6.4.1 individui giusti

Chiaramente l'aspetto che piu' mi premeva testare era se l'algoritmo fosse stato in grado di produrre una popolazione di individidui che formassero la TargetWord. Per fare cio' ho scelto una fitness che premiasse soprattutto gli individui in grado di formare sequenze di lettere in ordine corretto. Di conseguenza ho usato un valore di *rlw* particolarmente alto e non ho utilizzato la funzione *Lol*.

Ecco i risultati della simulazione cosi' come vengono riportati dall funzione *ProfileRun[filename_]* (4.6 a pag 16).

```
popolazione totale:
300
parametri:
{rlw,dsw,riw,bdw,stw}
{1000, 1, 5, 10, 10 }
popolazione di elementi perfetti:
{}
popolazione di elementi giusti:
{{2, 1}, {3, 13}, {4, 72}, {5, 151}, {6, 211}, {7, 201}, {8, 221}, {9, 228},
 {10, 229}, {11, 239}, {12, 249}, {13, 237}, {14, 233}, {15, 248}, {16, 264},
 {17, 265}, {18, 264}, {19, 261}, {20, 261}, {21, 274}, {22, 267}, {23, 268},
 {24, 268}, {25, 271}, {26, 260}, {27, 256}, {28, 254}}
numero runs:
30
```

con questa configurazione un individui in grado di formare la TargetWord prende 9000 punti di fitness. infatti dai dati raccolti notiamo che la popolazione di individui giusti cresce fortemente dopo la generazione numero 3. per vedere meglio questo trend bastera' dare il comando:

```
ListPlot[popgiusti[] ]
```

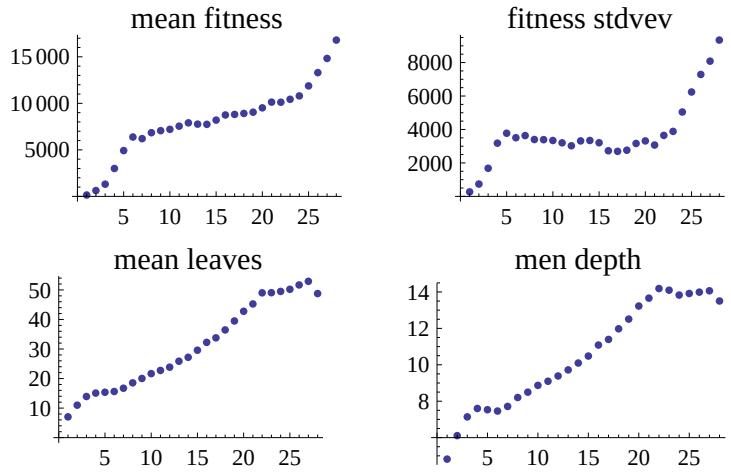


Figure 3: output del comando TrendPlot eseguito da PR[]

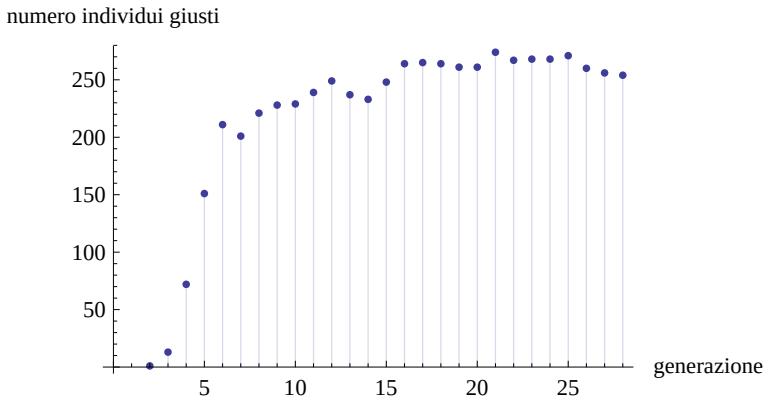


Figure 4: output del comando ListPlot[popgiusti[]], la popolazione di individui giusti cresce rapidamente dopo la generazione 4

Considerato che la popolazione totale e' di trecento individui, i parametri della fitness considerati sono pienamente soddisfacenti nell'ottica di trovare individui in grado di formare la TargetWord.

Possiamo pero' osservare dalla figura 3 che la complessita' di questi individui cresce molto rapidamente sia a livello di foglie che di profondita', inoltre gli individui delle generazioni superiore alla 8 hanno tutti un numero di BrokenDu almeno pari a 6 e un numero di step superiore al 60. [18]

Concludiamo quindi che il package funziona, ma con gli attuali parametri gli individui giusti sono complessi e antieconomici rispetto ai parametri indicati in 6.1

[18]per orrervare questi aspetti si lanci *ProfileGeneration[8]* e si osservino i valori della variabile *valgiusti*. Per approfondire si torni alla sezione 4.6 o si veda il file *tools.m*

6.4.2 DidSome

per cercare di ottenere individui piu' semplici ma che ottenessero comunque buoni risultati ho scelto di diminuire fortemente il parametro rlw e aumentare dsw. In questo modo vengono premiati gli individui che alterano la stack.

Inolte per cercare di premiare la semplcita' degli individui e' stata utilizzata la funzione Lol per modulare il punteggio dato da Step e da BrokenDU modificando la fitness nel seguente modo:

```
Fitness[params_]:=Module[{toret},
    (*pesi dei singoli punteggi : *)
    (*i pesi dei punteggi rappresentano l'"ambiente"*)
    rlw =   100;
    dsw =   200;
    riw =   5;
    bdw =   300;
    stw =   300;

    mdl = 0;     (*mean step lol*)
    ddl = 1;     (*delta step lol*)
    msl = 13;    (*mean broken du lol*)
    dsl = 5;     (*delta broken du lol*)

    toret=  params[[1]] rlw +
            params[[2]] dsw +
            params[[3]] riw +
            Lol[mdl,ddl,params[[4]]] * 1.0  bdw +
            Lol[msl,dsl,params[[5]]]  * 1.0 stw ;
    Return[toret];
];
```

cosi' facendo gli individui con BrokenDu pari a zero prenderanno 300 punti e via a scalare con la crescita dei BrokenDU, analogamente piu' gli individui avranno un numero di step pari a 13 piu' potranno ricevere un punteggio aggiuntivo di 300 in base agli step fatti.

Dalle simulazioni fatte, osservando le stack elaborate dagli individui [19], la tendenza generale e' di svuotare le stack e non di riempirle. Inoltre non sono stati trovati individui in grado di ricostruire la Stack, questo vuol dire che il premio dato da didsome e' troppo alto per gli scopi richiesti. Per quanto riguarda la semplicita' degli individui sono stati individuati dei trend di crescita delle foglie e della profondita' simili a figura 3.

Un'altra aspetto che e' venuto a galla e' che con l'aumentare delle dimensioni degli individui il tempo di valutazione riservato ad una generazione cresceva a ritmi impressionanti, creando simulazioni che avrebbero impiegato ore a terminare(consideriamo che la popolazione e' di 300 individui, un numero non grande). Per risolvere questo problema e' stata introdotta la variabile MAXDU che termina la valutazione di un individuo quando i suoi BrokenDU superano il valore di MAXDU.

settando MAXDU a 5 il tempo di runtime e' stato notevolmente ridotto, dopotutto individui con un elevato numero di BrokenDU non sono quelli che stiamo cercando.

[19] la variabile *stackdisth* contiene per ogni nerazione il tipo di stack formata e il numero di stack formate

6.5 Mimare l'individuo perfetto

il Package si e' rivelato efficiente nella creazione di individui in grado di creare la TargetWord, ma non di crearne di abbastanza semplici. Si e' quindi scelto di cercare i parametri in grado di mimare le caratteristiche elencate a pag 19.

Quello che e' emerso e' che modificare i pesi di bdw e stw non si e' rivelato sufficienete. Infatti sia con parametri come:

```
rlw =   50;
dsw =   1;
riw =   5;
bdw =   500;
stw =   500;
```

che con parametri come

```
rlw =   50;
dsw =   1;
riw =   5;
bdw =   500;
stw =   1000;
```

non si riusciva ad arrestare la crescita delle foglie e della profondita' degli individui per almeno un numero non trascurabile di generazioni.

I parametri che si sono stati decisivi invece sono quelli della funzione Lol. La funzione Lol che modulava il premio dato agli step ha dato i migliori risultati con una media di 13 step (quelli dell'individuo perfetto) e una sigma di 5. Provare a stringere la gaussiana a 2 non ha dato risultati soddisfacenti. Il maggiore impatto l'ha avuto la funzione Lol che modulava il premio dei BrokenDU.

Centrando la gaussiana su 0 e settando la sigma a 3 produceva risultati disastrosi con crescita di complessita' superiore alle precedenti, mentre inpostandola a 2 sono apparsi i primi risultati sperati:

la media delle foglie degli individui restava pari a 9per una decina di generazioni, mentre la profondita' a 5. Dei valori fortemente vicini a quelli cercati.

Listing 7: parametri usati per mimare le caratteristiche dell'individuo perfetto

```
rlw =   50;
dsw =   1;
riw =   5;
bdw =   500;
stw =   1000;

mdl = 0;    (*mean step lol*)
ddl = 2;    (*delta step lol*)
msl = 13;   (*mean broken du lol*)
dsl = 5;    (*delta broken du lol*)
```

Ripetendo le simulazioni con i medesimi parametri i risultati non erano costanti. Su cinque simulazioni effettuate solo due producevano i risultati cercati. Credendo che questo effetto fosse dovuto ad una mancanza di varieta' del genoma della popolazione ho aumentato il numero degli individui da 300 a 1000 riuscendo finalmente a stabilizzare i risultati. Ognuna delle 5 simulazioni ha dato risultati simili al seguente.

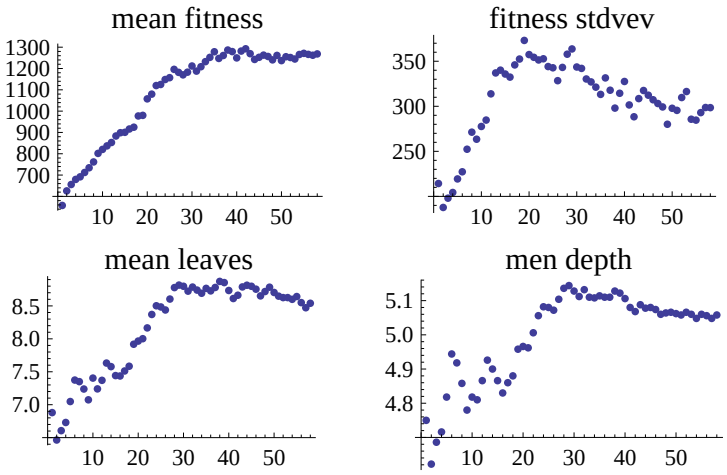


Figure 5: risultati ottenuti cercando di mimare l'individuo perfetto con una popolazione di 300 individui. Foglie medie circa 9 e profondita' media circa 5

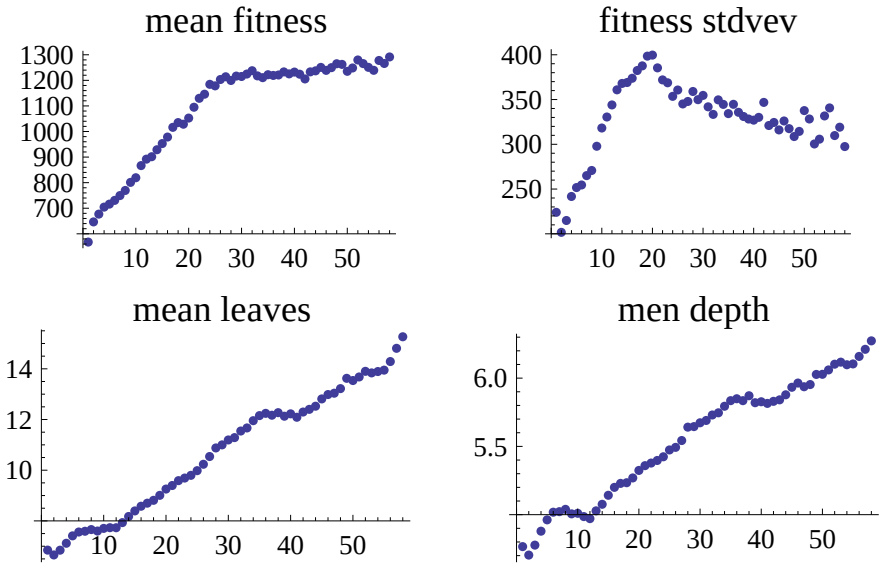


Figure 6: risultati ottenuti con una popolazione di 1000 individui e stessi parametri del listato 7

nonostante la crescita sia maggiore rispetto a quella in figura 5 possiamo vedere che le foglie medie sono costanti tra la generazione 35 e la generazione 43. Questo accade quando la fitness si stabilizza e contemporaneamente la sua deviazione standard diminuisce sensibilmente. Cioe' sempre piu' individui hanno una fitness simile.

Superata la generazione 40 questo la complessita' aumenta velocemente. Chiaramente questa tendenza non puo' essere evitata perche' la complessita' degli individui tendera' a crescere sempre di piu' con l'aumentare delle generazioni perche' il crossover non conserva la lunghezza degli individui. Si ricordi che la fitness non analizza nessuna caratteristica morfologica degli individui ma solo caratteristiche comportamentali. A mano a mano che i crossover aumentano nascono individui

con strutture ridondanti che non implicano ne spostamenti di lettere (aumento degli step), ne BrokenDU frequenti (proliferazione di funzioni logiche e di lettura). Per questi motivi ritengo soddisfacente che la media delle foglie si stabilizzi anche solo per un numero di generazioni limitato. Soprattutto il fatto che questo avvenga quando la std della fitness diminuisce, sta ad indicare che nonostante la fitness sia gia' stabile da qualche generazione non ha ancora uniformato la popolazione su una caratteristica preponderante, quando questa uniformazione avviene il numero di foglie si stabilizza.

6.6 ricerca della perfezione

A questo punto non resta che trovare l'individuo perfetto. La linea guida da seguire e' di non perdere i caratteri morfologici ottenuti nella sezione precedente e di aumentare la capacita' di creare delle stack corrette.

6.6.1 primi tentativi, primi fallimenti

la prima cosa fatta' e' stata chiaramente la piu' intuitiva: lasciare tutti i parametri invariati tranne rlw, facendola aumentare. intuitivamente pero' rlw non deve amumentare troppo se no la selezione morfologica va a perdersi.

Come primo valore rlw viene settato a 100, questo comporta un punteggio di 900 punti per una stack corretta. Dai risultati appare che le caratteristiche morfologiche si conservano ma nessun individuo giusto si palesa.

Cercando di movimentare un po' l'azione degli individui amumento il parametro dsw da 1 a 10. Questo aumenta di gran lunga il numero di individui giusti, ma appaiono essere tremendamente antiestetici e poco funzionali (minimo 6 BrokenDU e 30 step).

Aumentando ancora dsw da 10 a 20 le stack si ritrovano tutte vuote e nessun individuo giusto appare. lo stesso vale per valori di dsw pari a 50. insomma nonostante il premio per ogni lettera giusta sia di 250 le stack continuano a essere svuotate e non ririempite.

Decido quindi di portare dsw a 1 e di tornare alla configurazione del listato 7 , incrementando rlw fino a 250. La popolazione di individui giusti tende ad aumentare ma comunque a discapito della semplicita', abbassare rlw a 200 invece non produceva alcun individuo giusto. Insomma data la configurazione del listato 7 rlw= 250 e' una sorta di soglia di ativazione per la produzione di individui giusti ma tende ad non conservare le caratteristiche di semplicita' degli individui.

6.6.2 selezione aggressiva

a questo punto, non potendo piu' agire sui parametri senza perdere in semplicita' ho adottato una tecnica piu' selettiva: assegnare fitness nulla ad individui con step troppo alti e BrokenDu maggiori di MAXDU. [20]

[20]la variabile MAXDU e' gia' stata assegnata per ridurre i tempi di valutazione ma nessn parametro della fitness dipendeva direttamente da lei

E' stata quindi introdotta la variabile MAXSTEP che segna il mitite di step massimi, sopra questo limite l'individuo viene soppresso, a prescindera delle altre votazioni. La fitness e' stata quindi modificata come segue:

```
Fitness[params_]:=Module[{toret},

    (*pesi dei singoli punteggi : *)
    (*i pesi dei punteggi rappresentano l'"ambiente"*)
    rlw =   250;
    dsw =   1;
    riw =   5;
    bdw =   500;
    stw =   1000;

    mdl = 0;     (*mean step lol*)
    ddl = 2;     (*delta step lol*)
    msl = 13;    (*mean broken du lol*)
    dsl = 5;     (*delta broken du lol*)

    MAXSTEP = 30;
    (*penalizza fortemente i programmi troppo lunghi*)
    If[ params[[4]] > MAXDU || params[[5]] >= MAXSTEP,
        toret=  0    ,
        toret=  params[[1]] rlw +
                params[[2]] dsw +
                params[[3]] riw +
                Lol[mdl,ddl,params[[4]]] * 1.0  bdw +
                Lol[msl,dsl,params[[5]]]  * 1.0 stw ;
    ];
    Return[toret];
];
```

quello che mi aspettavo originariamente con questo approccio era di ottenere individui piu' semplici per piu' generazioni possibili, in modo da contrastare la complessita' crescente, dando la possibilita' di emergere agli individui particolarmente attivi nelle prime fasi.

Infatti settando MAXSTEP=20 sia le foglie che la profondita' resta uguale a quella iniziale per le prime 20 generazioni(si veda figura 10), ma gli individui in grado di formare la TargetWord non sono piu' di due per generazione.

Settando MAXSTEP=40 la crescita della complessita' comincia quasi subito.

Molto aristotelicamente assegnando a MAXSTEP il valore di trenta al primo tentativo non solo appaiono molti individui giusti, ma addirittura una popolazione non trascurabile di individui perfetti.

I sisultati della simulazione sono i seguenti:

```
popolazione totale:
1000
popolazione di individui perfetti:
{{20, 1}, {21, 1}, {22, 2}, {23, 3}, {24, 5}, {25, 7}, {26, 8}, {27, 7},
 {28, 5}, {29, 7}, {30, 9}, {31, 5}, {32, 4}, {33, 4}, {34, 5}, {35, 2},
 {36, 4}, {37, 3}, {38, 4}, {39, 5}, {40, 2}, {42, 2}, {43, 4}, {44, 3},
 {45, 4}, {46, 4}, {47, 6}, {48, 5}, {49, 4}, {50, 11}, {51, 6}, {52, 6},
 {53, 3}, {54, 4}, {55, 3}, {56, 1}, {57, 1}, {58, 1}, {59, 1}}
popolazione di elementi giusti:
{{2, 1}, {10, 1}, {11, 1}, {12, 3}, {13, 9}, {14, 11}, {15, 19}, {16, 35},
 {17, 67}, {18, 115}, {19, 179}, {20, 237}, {21, 293}, {22, 413}, {23, 493},
 {24, 566}, {25, 572}, {26, 607}, {27, 590}, {28, 616}, {29, 644}, {30, 664},
 {31, 651}, {32, 667}, {33, 647}, {34, 662}, {35, 648}, {36, 701}, {37, 710},
 {38, 700}, {39, 707}, {40, 700}, {41, 673}, {42, 689}, {43, 721}, {44, 719},
 {45, 747}, {46, 715}, {47, 715}, {48, 732}, {49, 774}, {50, 766}, {51, 763},
```

```
 {52, 754}, {53, 776}, {54, 783}, {55, 766}, {56, 796}, {57, 823}, {58, 808}}
parametri:
{rlw,dsw,riw,bdw,stw,msl,dsl,mdl,ddl}
{250, 1, 5, 500, 1000, 0, 2, 13, 5}
numero runs:
60
MAXDU:
5
MAXSTEP:
30
```

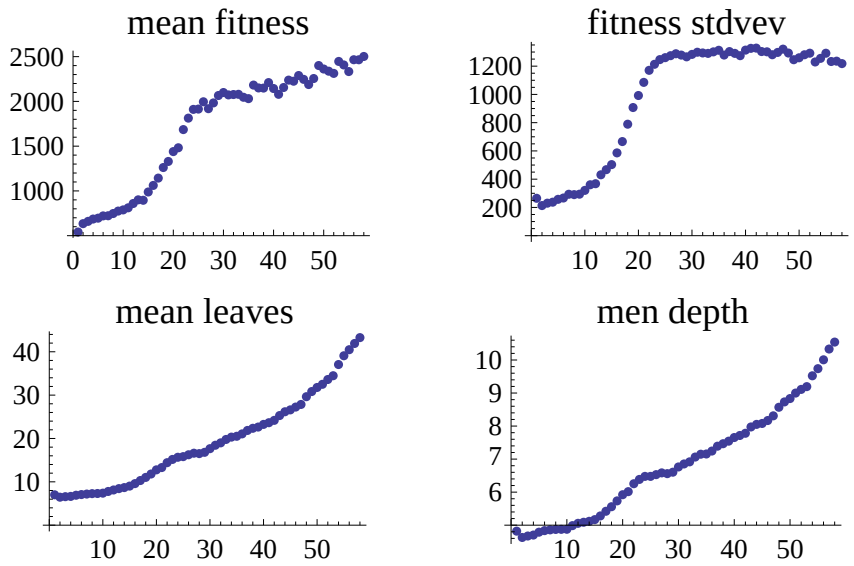


Figure 7: trend della simulazione in cui sono apparsi degli individui perfetti.

con il seguente comando possiamo plottare la distribuzione degli individui perfetti per generazione

```
ListPlot[perfetti[]]
```

infine per comparare le popolazioni di individui perfetti con quella di individui giusti si puo' dare semplicemente il comando:

```
ListPlot[{popgiusti[],perfetti[]}]
```

quello che appare lampante dalla figura 9 e' che quando esplode la popolazione di individui giusti appaiono anche degli individui perfetti ma che non sono predominanti. E' difficile anche capire che relazione ci sia tra popolazione di individui giusti e popolazione di individui perfetti guardando la figura 8, ma il solo fatto che la popolazione di individui perfetti non si estingua e' una soddisfazione notevole.

Possiamo dare una prima interpretazione pensando che quando gli individui perfetti appaiono vengono premiati e tramite CrossOver vengono incapsulati in altri individui piu' complessi, ma che guadagnano in efficienza, facendo esplodere la popolazione di individui giusti.

Purtroppo le simulazioni con questi parametri non danno sempre i risultati sperati, su 11 simulazioni solo 3 hanno avuto risultati simili a questa. In tutti i

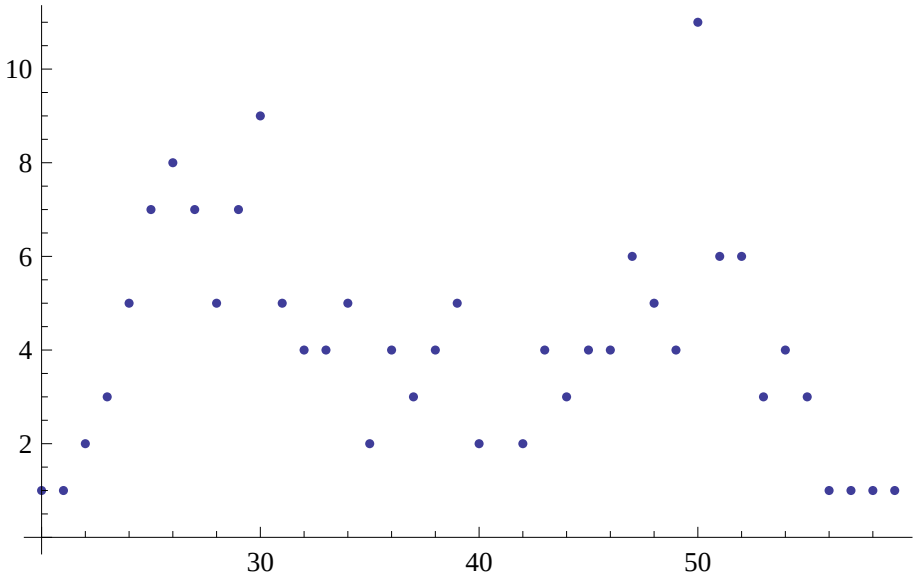


Figure 8: andamento del numero di individui perfetti in funzione della generazione

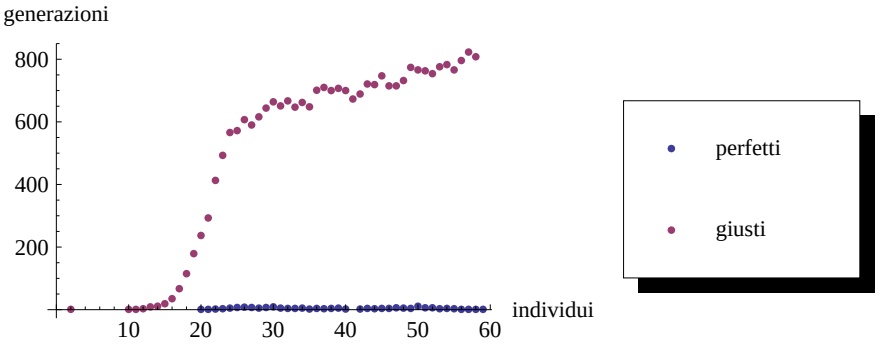


Figure 9: comparazione del numero di individui perfetti vs numero di individui giusti per generazione

casi rimanenti la popolazione di individui giusti non esplode affatto. Questo porta dei punti a favore alla spiegazione precedente, senza individui perfetti non c'e' un avanzamento collettivo della popolazione.

7 Ottimizzazione e Possibili Sviluppi

Per quanto riguarda l'ottimizzazione questo programma e' molto e adatto alla parallelizzazione. La parte computazionalmente piu' impegnativa e' la valutazione degli individui che e' un operazione indipendente e relativamente facile da implementare, tutti ingredienti adatti per la parallelizzazione magari su gpu(memoria permettendo). Lo stesso ragionamento vale per la fitness, il CrossOver, il Profiling e la Mutazione. Sono tutte operazioni indipendenti e relativamente facili.

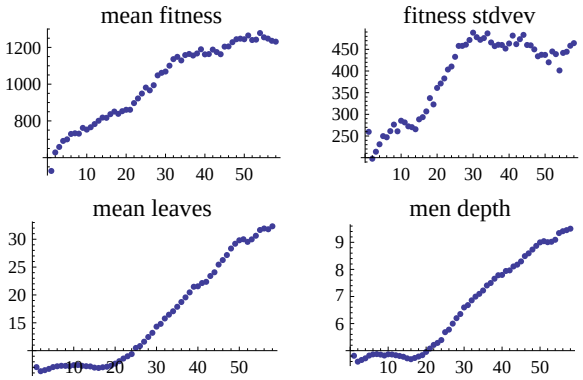


Figure 10: con MAXSTEP=20 la complessita' degli individui resta costante per le prime 20 generazioni

Per quanto riguarda i miglioramenti del mio codice bisogna pensare a scrivere una funzione di Mutazione che garantisca la correttezza sintattica, cosa che non ho fatto per mancanza di tempo. Inolte bisognerebbe implementare una versione di HAC che risolva il problema DUT DUW sulla posizione del ramo e non passando una Head ad hoc come ho fatto (non avevo ancora acquisito buone conoscenze del comando Position quando ho implementato *firstgen.m*).

Infine si potrebbe dedicare piu' tempo alla ricerca della giusta combinazione di parametri e condizioni per creare una popolazione di elementi perfetti nelle prime generazioni con costanza. Confido pero' nella struttura del mio Package e ritengo sia primariamente una questione di tempo.

Infine sarebbe bello testare l'algoritmo in casi in cui l'individuo perfetto scritto nella sezione 6, non sia il migliore. Per esempio nel caso in cui la DefaultStack presentasse gia' delle lettere in posizioni corrette, per le quali sarebbe inutile riposizionarle a terra.

8 Conclusioni personali

Per quanto questo esercizio sia astratto mi ha forzato a comprendere il cuore del linguaggio che utilizza *Mathematica*: la struttura delle espressioni. Ritengo che avere una buona conoscenza di base di un linguaggio cosi' astratto sia decisamente una cosa costruttiva ed educativa.

Essendo abituato al massimo a livelli procedurali od ObjectOriented ho sicuramente fatto fatica ad abituarmi all'inizio, ma dopo le difficolta' iniziali l'apprendimento si e' rivelato veloce e appagante. Il maggior sforzo e' stato nell'acquisire una mentalita' molto piu' flessibile e non piu' incentrata sui tipi di dato ma sulle espressioni ed i pattern. L'aver potuto scrivere una funzione concettualmente complessa come HAC in tre righe di codice rende onore a questo linguaggio e alla sua praticita'.

Dal punto di vista delle prestazioni sono rimasto piacevolmente colpito dall'uso modico dell memoria che e' stato fatto durante le simulazioni, anche con popolazioni di 5000 individui. E' chiaro che questa flessibilita' vada a discapito di un utilizzo della CPU molto elevato. Dati i tempi di runtime di questo Package bisogna seriamente

tenere in considerazione che il tempo risparmiato a scrivere il codice e' pienamente sufficiente per compensare il tempo di computazione.

Un altro risparmio di tempo notevole e' dovuto alla flessibilita' del linguaggio rispetto al C++ per esempio. Con i linguaggi OO a basso livello sarebbe stato necessario un lungo periodo di design e la scrittura "a braccio" del codice andrebbe ridotta al minimo. Con *Mathematica* problemi complessi possono essere risolti in una giornata e la produttivita' dello sviluppatore e' decisamente superiore alla media.

Sinceramente sarei curioso di vedere come si comporta *Mathematica* con progetti che hanno una mole molto maggiore sia a livello di moduli che di codice.

Una cosa a cui non sono pero' riuscito ad abituarmi e' la mancanza di un bebugger che permetta di eseguire riga per riga il codice a livello di Kernell, secondo me una qualita' insostituibile che deve avere un linguaggio, per quanto sia scritto per ridurre al minimo il Debug.

9 Sorgenti

9.1 moves.m

```
TargetWord = {l,a,s,r,e,v,i,n,u};
MyStack = {};
MyTable = {};

LispNil = "lisp nil";
maxdustep = 2 * Length[TargetWord] ;
BrokenDU = 0;
StepCount = 0;
MAXDU = 5;

RandomInit[] := Module[{cutpos , permutations , tocut},
    cutpos = Random[Integer, {1, Length[TargetWord]}];
    (*get random permutation*)
    permutations = Permutations[TargetWord];
    tocut = permutations[[
                Random[Integer, {0,Length[permutations]} ]
            ]];
    MyStack = Take[tocut,cutpos];
    MyTable = Take[tocut,-(Length[tocut]-cutpos)];

    Print["starting stack and table:" , {MyStack , MyTable}];

];

CS[] := If[Length[MyStack] == 0,Return[LispNil] , Return[Last[MyStack]]];

TB[] := Module[{i=0, c , tab , first , splitted} ,

        tab =   Table[ MyStack[[i]] == TargetWord[[i]] ,
                        {i,Length[MyStack] }
                ];

        splitted = Split[tab];
        If[ Length[splitted ] == 0,
            Return[LispNil]     ,
            first = splitted[[1]];
        ];

        c = If[MemberQ[first,True] ,
                Length[first],
```

```
                0
            ];

        If[ c === 0,
            Return[LispNil] ,
            Return[MyStack[[c]]]
        ];
];

NN[] := Module[{i=0 , toret , c , tab ,first , splitted},

        tab =   Table[ MyStack[[i]] === TargetWord[[i]] ,
                        {i,Length[MyStack] }
                ];

        splitted = Split[tab];
        If[ Length[splitted ] === 0,
            Return[TargetWord[[1]]]      ,
            first = splitted[[1]];
        ];

        c = If[MemberQ[first,True] ,
                Length[first],
                0
            ];

        If[ c===0,
            Return[TargetWord[[1]]],
            Null
        ];

        If[ c===Length[TargetWord]   ,
            Return[LispNil],
            Return[TargetWord[[c+1]]]
        ];
];


(*************************************************************)
(*************************************************************)
(*************************************************************)


MTS[x_] := Module[{},
    StepCount ++;
    If[ MemberQ[MyTable,x]   ,
        (* potrebbe non funzionare con le lettere doppie*)
        (*Print["stack table" , {MyStack , MyTable}];*)
        MyTable = DeleteCases[MyTable , x];
            MyStack = Append[MyStack , x] ;
            Return[x]    ,
        (* se non c'e' sul tavolo non fare niente*)
        Return[x]
    ];
];


MTT[x_] := Module[{},
    StepCount ++;
    If[ Length[MyStack] === 0 , Return[x] , Null];

    If[ Last[MyStack] === x ,
        (* togli l'ultimo e solo l'ultimo *)
        (*Print["stack table" , {MyStack , MyTable}];*)
        MyStack = Delete[MyStack , Length[MyStack]];
            MyTable = Append[MyTable , x] ;
            Return[x]    ,

        (* se non e' l'ultimo della stack non fare niente*)
        Return[x] ;
```

```
    ];
];


(*************************************************************)
(*************************************************************)
(*************************************************************)


NOT[expr_] := If[expr === LispNil , True , False];

EQ[expr1_ , expr2_] := Module [ {} ,Return[expr1===expr2] ];



DU[work_, test_] := Module[{tmp,counter = 0},
    If[BrokenDU >= MAXDU,
        Return[Null],
        Null
    ];
    While[  test ===False   ,
        (*forza valutazione con assegnazione*)
        tmp = work ;
        counter ++ ;
        If[counter -1 === maxdustep ,
            BrokenDU ++;
            Return[LispNil] ,
            Null
        ];
    ];
];

SetAttributes[DU,HoldAll];
```

9.2 firstgen.m

```
Nind = 1000;
MaxHacCall = 20;
CurrHacCall = 0;
MinLeaf = 3;
MaxLeaf = 20;

Dummies = {e , n , duw , dut , mt , ms};

sostituzioni := {
    e1  -> xHAC[fEQ1]    ,
    e2  -> xHAC[fEQ2]    ,
    n   -> xHAC[fNOT]    ,
    duw -> xHAC[fDUW]    ,
    dut -> xHAC[fDUT]    ,
    mt  -> xHAC[fMTT]    ,
    ms  -> xHAC[fMTS]
};

finalsost = {   fEQ     :>  EQ   ,
                fNOT    :>  NOT  ,
                fDU     :>  DU   ,
                fMTT    :>  MTT  ,
                fMTS    :>  MTS  ,
                fNN     :>  NN   ,
                fCS     :>  CS   ,
                fTB     :>  TB
};

EQList = {  fEQ[e1,e2] , fNOT[n] , fDU[duw,dut] , fMTT[mt] , fMTS[ms] ,
            fCS[] , fTB[] , fNN[] };
```

```
NOTList = { fCS[] , fTB[] , fNN[] };

DUWList = { fEQ[e1,e2] , fNOT[n] , fDU[duw,dut] , fMTT[mt] , fMTS[ms] ,
            fCS[] , fTB[] , fNN[] };

DUTList = { fEQ[e1,e2] , fNOT[n] };

MTTList = {  fCS[] , fTB[] , fNN[] };

MTSList = {  fCS[] , fTB[] , fNN[] };

CSList = NNList = TBList = {};


ChooseArg[head_] := Module[{toret},

    Which[  head === fEQ1,
            toret = RandomChoice[EQList],

            head === fEQ2,
            toret = RandomChoice[EQList],

            head === fNOT,
            toret = RandomChoice[NOTList],

            head === fDUT,
            toret = RandomChoice[DUTList],

            head === fDUW,
            toret = RandomChoice[DUWList],

            head === fMTT,
            toret = RandomChoice[MTTList],

            head === fMTS,
            toret = RandomChoice[MTSList],

            head === fCS,
            toret = RandomChoice[CSList],

            head === fNN,
            toret = RandomChoice[NNList],

            head === fTB,
            toret = RandomChoice[TBList],

            True,
            Print["\n"];
            Print["Hey stai attento! ChooseArg non ha assegnato niete!"];
            Print["\n"];
    ];
    Return[toret];
];


HAC[]:=Module[{randhead , sostitutiprova},
    randhead = RandomChoice[EQList];

    sostituitiprova = randhead /. sostituzioni ;
    Return[ sostituitiprova /. { xHAC -> HAC} ];

];


HAC[fathhead_] := Module [{args , sostituiti , sostitutiprova} ,

    If[ CurrHacCall >= MaxHacCall,
        Print["max hac calls raggiunto!"];
        Return[Null],
```

```
        Null
    ];

    CurrHacCall ++;

    args = ChooseArg[fathhead];
    sostituitiprova = args /. sostituzioni ;
    Return[ sostituitiprova /. { xHAC -> HAC} ];
];

RigthInd[] := Module[{ind},
    CurrHacCall = 0 ;
    ind = HAC[];
    (*finche c'e' Null rifai l'individuo*)
    While[  !FreeQ[ind,Null]            ||
            LeafCount[ind] < MinLeaf    ||
            LeafCount[ind] > MaxLeaf    ,

            CurrHacCall =0;
            ind= HAC[];
    ];
    Return[ind];
];


FirstGeneration[] := Table[RigthInd[] , {Nind}];
```

9.3 crossover.m

```
TotalCrossFail = 0;
PartialCrossFail = 0;

CrossOver[{ind1_,ind2_}]:= Module[{ rami1,posrami1,ramo1,posramo1 ,
                                    pospadre1, headpadre1 ,listescelta1 ,
                                    firstgood , secondgood,
                                    livelli2 , head2 , candidati1   ,
                                    poscandidati , i,
                                    ramo2 , ramo2pos , ramo2padre ,
                                    ramo2padrepos , listascelta2 ,
                                    figlio1 , figlio2
                                    },
If[Random[]<PC,

    rami1 = Level[ind1,Infinity];

    (*voglio una lista ordinata a caso delle posizioni dei varirami*)
    posrami1 = Table[Position[ind1 , rami1 [[l]] ] , {l,Length[rami1]}  ];
    posrami1 = Flatten[posrami1,1];
    posrami1 = Union[posrami1];
    posrami1 = RandomSample[posrami1];

    (*caratteri di controllo per vedere se c'e' un punto di taglio*)
    (*con dei candidati compatibili per lo scambio*)
    firstgood = False;
    secondgood = False;

    (* prova in sequenza le varie posizioni, finche' non ne trovi una *)
    (* compatiblie per tagliare *)
    For[i=1 , i<=Length[posrami1] , i++ ,

        posramo1 = posrami1[[i]];
        ramo1 = Extract[ind1,posramo1];
        pospadre1 = Append[posramo1[[1;;-2]],0];
        headpadre1 = Extract[ind1,pospadre1];

        listascelta1 = ChooseList[headpadre1 , posramo1];
```

```
        livelli2 = Level[ind2,Infinity];
        head2 = Map[Head,livelli2];

        (* di tutte le sottoespressioni cerca quelle con una head compatibile *)
        (* con la head del ramo1*)
        candidati1 = Position[  head2 ,
                                x_ /; MemberQ[ Map[Head,listascelta1 ], x ]
        ];

        If[ Length[candidati1] == 0     ,
            firstgood = False;
                PartialCrossFail ++;
                (*se non ci sono candidati passa alla prossiama sottoespressione*)
                Continue[] ,
            firstgood = True
        ];

        (*a questo punto abbiamo almeno un candidati compatibile*)
        (*analogamente a quanto fatto prima posiziono a random le*)
        (*posizioni dei candidati*)
        levelcandidati = Extract[livelli2,candidati1];
        poscandidati = Table[Position[ind2,levelcandidati[[l]] ] ,{l,Length[
            levelcandidati]} ] ;
        poscandidati = Flatten[poscandidati , 1];
        poscandidati = Union[poscandidati];
        poscandidati = RandomSample[poscandidati];

        (*trova il primo candidato compatible per lo scambio contrario*)
        For[j=1 , j<=Length[poscandidati] , j++ ,

            ramo2pos = poscandidati[[j]];
            ramo2 = Extract[ind2,ramo2pos];

            (*la pos del padre ha come ultimo valore uno zero*)
            ramo2padrepos = Append[ramo2pos[[1;;-2]],0];
            ramo2padre = Extract[ind2,ramo2padrepos];

            (*ora devi controlla se si piu' fare lo scambio al contrario*)
            (*controlla se ramo1 puo' essere come argomento di ramo2fath*)

            listascelta2 = ChooseList[ ramo2padre , ramo2pos  ];

            If[MemberQ[Map[Head , listascelta2]  ,  Head[ramo1] ] ,
                secondgood = True ;
                    Break[]; ,
                    PartialCrossFail ++;
                    secondgood = False
            ];

        ];

        (*se hai trovato sia il punto di taglio che il candidato giusto*)
        (*esci dal ciclo*)
        If[firstgood && secondgood , Break[] , Null];

    ];(*FINE PRIMO FOR!*)

    (*************************************************************)
    (*      ora crossa effettivamente                            *)
    (*************************************************************)

    If[firstgood == False,Print["NESSUN PUNTO DI TAGLIO TROVATO"],Null];
    If[secondgood == False,Print["NESSUN CANDIDATO TROVATO"],Null];

    (*ReplacePart e' un comando molto comodo in quanto accetta l'output *)
    (*di position come argomento*)
    If[ firstgood && secondgood ,
            figlio1 = ReplacePart[ind1 , posramo1 -> ramo2];
            figlio2 = ReplacePart[ind2 , ramo2pos -> ramo1]; ,
```

```
        Print["IL CROSS NON PUO' AVVENIRE"];
            figlio1 = ind1;
            figlio2 = ind2;
            TotalCrossFail ++;
    ];


    Return[{figlio1,figlio2}];

,(*fine if random*)

Return[{ind1,ind2}]
];

];

(************************************************************)
(************************************************************)

(*resituisci la lista di compatibilita' per una determinata head*)
(*la posizione del ramo figlio serve per capire se il ramo figlio*)
(*e' il primo o il secondo argomento di DU*)
ChooseList[head_ , posramofiglio_ ]:= Module[{testhead , listascelta},

testhead = head;

    (*problema dut e duw*)
    (*devo vedere se e' il promo argomento o il secondo del DU*)
    If[ testhead === fDU ,
        If [ Last[posramofiglio] === 2 ,
                testhead = fDUT ,
                testhead = fDUW
        ]; ,
        Null
    ];
    Which[  testhead === fEQ,
            listascelta = EQList,

            testhead === fNOT,
            listascelta = NOTList,

            testhead === fDUT,
            listascelta = DUTList,

            testhead === fDUW,
            listascelta = DUWList,

            testhead === fMTT,
            listascelta = MTTList,

            testhead === fMTS,
            listascelta = MTSList,

            testhead === fCS,
            listascelta = CSList,

            testhead === fNN,
            listascelta = NNList,

            testhead === fTB,
            listascelta = TBList,

            True,
            Print["\n"];
            Print["Hey stai attento! ChooseArg non ha assegnato niete!"];
            Print["\n"];
    ];
    Return[listascelta];
];
```

9.4 fitness.m

```
(*************************************************************)
(*              VALUTAZIONE                                  *)
(*************************************************************)

finalsost = {   fEQ     :>  EQ  ,
                fNOT    :>  NOT ,
                fDU     :>  DU  ,
                fMTT    :>  MTT ,
                fMTS    :>  MTS ,
                fNN     :>  NN  ,
                fCS     :>  CS  ,
                fTB     :>  TB
};

DefaultStack = {u,n,i,v} ;
DefaultTable = {e,r,s,a,l};

EvalInd[ind_] := Block[{    TargetWord = {l,a,s,r,e,v,i,n,u},
                            MyStack = DefaultStack,
                            MyTable = DefaultTable,
                            BrokenDU = 0 ,
                            StepCount = 0,
                            MAXDU = 5,
                            newind,
                            rich = 0
                    } ,

    newind = ind /. finalsost;
    rich = Richness[ind];

    Return[{MyStack , MyTable , BrokenDU, StepCount , rich}];
];

(*************************************************************)
(*********          P U N T E G G I          ****************)
(*************************************************************)

easysubs = { l :> 1 , a :> 2 , s:> 3 , e:>4 ,r:> 5 , v:>6 , i:> 7 , n:>8 , u:>9 };

exprslist= {fEQ , fNOT , fDU , fMTT , fMTS , fCS ,fNN , fTB };

(*conta il numero di lettere che si trovano nella posizione corretta*)

RigthLetters[stack_] := Module[{newstack , score},
    newstack = stack /. easysubs;
    score =Count[   Table[newstack[[i]] === i , {i,Length[newstack]}   ],
                    True];
    Return[score];
];

(* conta quanto lo stack sia cambiato sia per differenza tra singole
    lettere che per differenza di lunghezza *)

DidSome[stack_] := Module[{score},
    (* guarda se ha cambiato la DefaultStack *)
    score =Count[   Table[  stack[[i]] =!= DefaultStack[[i]] ,
                            {i,Length[newstack]}   ],
                    True];
    score += Abs[Length[stack]-Length[DefaultStack]];
    Return[score];
];

(* conta la ricchezza di espressioni nell'individuo *)

Richness[ind_] := Module[{score, pos},
    score = Count [ Table[  FreeQ[ind,exprslist[[i]]] ,
```

```
                                {i,Length[exprslist] } ]     ,
                        False
    ];
    Return[score];
];

(*************************************************************)

FitnessParameters[evalres_] := Module[{rl,ds,ri,bd,st},
    (*valuta l'individuo e valuta risultato*)
(*  evalres = EvalInd[ind];*)

    rl = RigthLetters[evalres[[1]]];
    ds = DidSome[evalres[[1]]];
    ri = evalres[[5]];(* la richness viene calcolata in EvalInd*)
    bd = evalres[[3]];(* broken du*)
    st = evalres[[4]];(* numero di step*)

    Return[{rl,ds,ri,bd,st}];
];


Fitness[params_]:=Module[{toret},

    (*pesi dei singoli punteggi : *)
    (*i pesi dei punteggi rappresentano l'"ambiente"*)
    rlw =   250;
    dsw =   1;
    riw =   5;
    bdw =   500;
    stw =   1000;

    msl = 13;   (*mean step lol*)
    dsl = 5;    (*delta step lol*)
    mdl = 0;    (*mean broken du lol*)
    ddl = 2;    (*delta broken du lol*)

    MAXSTEP = 30;
    (*penalizza fortemente i programmi troppo lunghi*)
    If[ params[[4]] > MAXDU || params[[5]] >= MAXSTEP,
        toret=  0   ,
        toret=  params[[1]] rlw +
                params[[2]] dsw +
                params[[3]] riw +
                Lol[mdl,ddl,params[[4]]] * 1.0  bdw +
                Lol[msl,dsl,params[[5]]]  * 1.0 stw ;
    ];
    Return[toret];


];

GenCoppie[gen_List,normfitlist_List]:= Module[{coppie},
    (*supponendo che sia normalizzata*)
    coppie =Table[RandomChoice[(normfitlist * 1.0) -> gen , 2] ,{Nind/2}];
    Return[coppie];
];


Muta[ind_]:=Module[{},
    (* get all the subexpressions (our bits)*)
    subs = Level[ind,Infinity];
    positions = Table[Position[ind , subs[[i]]], {i, Length[subs]}];
    positions = Flatten[positions,1];
    (*togli dupplicati*)
    positions = Union[positions];
    (*muta*)
    (*Print[positions];*)
    Map[ChangeHead[ind ,#1]&   ,positions];
];
```

```
ChangeHead[ind_,posexpr_] := Module[{posfath , fathead , compheadlist , comphead ,
    replpos },
    If[ Random[] < PM ,
            (* get fath head*)
            expr = Extract[ind,posexpr];
            posfath  = Append[posexpr[[1;;-2]],0];
            fathed = Extract[ind,posfath];

            (* get compatible expr*)
            compheadlist = Map[Head,ChooseList[fathed , posexpr]];
            (* nella lista potrebbe esserci la stessa espressione*)
            compheadlist = Delete[  compheadlist ,
                                    Flatten[Position[compheadlist,
                                                     Head[expr]] ,
                                            1]
            ];
            comphead = RandomChoice[compheadlist];
            (* need the position of the "son head"*)
            If[Last[posexpr] =!= 0 ,
                replpos = Append[ posexpr  ,0],
                replpos = posexpr
            ];
            (* e' buggato, bisogna controllare anche se il figlio e' compatibile *)
            ind = ReplacePart[ind,replpos->comphead];
        Null
    ];

];

(* histories*)
parh={};
fith={};
normh={};
stackh = {};
stackdisth={};
devh={};
meanh = {};
meanleafh = {};
meandeph = {};
leafh = {};
deph = {};

Profiler[ingen_,fitlist_,params_,valutazioni_]:=Module[{stack},
    Print["\n PROFILING BEGIN\n"];

    (*parh = Append[parh,params];*)

    stacks = Table[valutazioni[[j]] [[1]] , {j,Nind}] ;
    stackdisth = Append[stackdisth , F2D[stacks]];
    stackh = Append[stackh , stacks];

    fith = Append[fith,fitlist];

    (* calculate std dev e media*)
    devh = Append[devh, StandardDeviation[ fitlist ]];
    meanh = Append[meanh, Mean[ fitlist ]];

    meanleafh = Append[meanleafh, Mean[Map[LeafCount,ingen]] ];
    (*leafh = Append[leafh, Map[LeafCount,ingen]] ;*)

    meandeph = Append[meandeph, Mean[Map[Depth,ingen]] ];
    (*deph = Append[deph, Map[Depth,ingen] ];*)

    Print["\n PROFILING END \n"];


];
```

```
StepGen[ingen_List] := Module[{ params , fitlist , normfitlist , coppie , outgen },
    Print["\n STEP BEGIN\n"];

    (* qui avviene la vautazione degli individui*)
    Print["valuto"];
    valutazioni = Map[EvalInd,ingen];

    Print["parametrizzo"];
    params = Map[FitnessParameters,valutazioni];

    Print["fitness"];
    fitlist = Map[Fitness, params];
    If[Min[fitlist] < 0 , fitlist -=  Min[fitlist] , Null];

    Print["profilo"];
    Profiler[ingen,fitlist,params,valutazioni];

    normfitlist = fitlist / Apply[Plus,fitlist];

    Print["crossover"];
    coppie = GenCoppie[ingen,normfitlist];
    outgen = Map[CrossOver,coppie];
    outgen = Flatten[outgen,1];

    Print["\n STEP END\n"];

    Return[outgen];
];
```

9.5 tools.m

```
<<perfetto.m

(*resituisci una lista di coppie {fitness , individui con quella fiteess}*)
(*si puo' applicare a qualsiasi lista*)
F2D[fitlist_] := Module[{sortfit, asd},
    sortfit = Sort[fitlist];
    asd = Table[ {Split[sortfit][[k]][[1]],
       Length[Split[sortfit][[k]]]}, {k, Length[Split[sortfit]]}];
    Return[asd];
];

(*analizza la singola generazione*)
(*richiede che sia abilitato il profiling su fith,leafh e deph*)
GenGraphs[n_]:= Module[{fitnessg, fitdistg , leafg , depg},
    (*build the graph*)
    fitnessg = ListPlot[fith[[n]] ,
                        AxesLabel->{ "individui","fitness" }    ,
                        PlotLabel->"fitness generazione " + n
    ];

    fitdistg = ListPlot[F2D[fith[[n]]] ,
                        AxesLabel->{"individui","fitness"  }    ,
                        PlotLabel->"distribuzioe fitness generazione " + n
    ];

    leafg = ListPlot[leafh[[n]] ,
                        AxesLabel->{"individui","LeafCount" }   ,
                        PlotLabel->"leaves " +n
    ];

    depg = ListPlot[leafh[[n]] ,
                        AxesLabel->{"individui","Depth"}    ,
                        PlotLabel->"depth "+n
    ];
```

```
    grid = {{fitnessg , fitdistg},{leafg,depg}};

    Show[GraphicsGrid[grid]]

];


(* brutto ma necessario*)
Lol[media_,sigma_,a_]:= Module[{Mymax , Myfunc},
    Myfunc = PDF[NormalDistribution[media,sigma]];
    Mymax = Apply[Myfunc,{media}];
    Return[Apply[Myfunc , {a}] / Mymax];
];


PlotTrends[]:= Module[{fit , leaf ,dep,dev},
    fit = ListPlot[meanh , PlotLabel->"mean fitness"];
    dev = ListPlot[devh , PlotLabel->"fitness stdvev"];
    leaf = ListPlot[meanleafh , PlotLabel->"mean leaves"];
    dep = ListPlot[meandeph , PlotLabel->"men depth"];

    grid = {{fit,dev},{leaf,dep}};

    Show[GraphicsGrid[grid]]
];

(*cerca nelle stack di tutte le generazioni se ci sono *)
(*stack corrette *)
good[]:= Module[{},
    Return[Position[stackdisth,TargetWord]];

];

(*data una generazione stampa valutazione elementi giusti*)
(*e salva gli individui giusti in "giusti" *)
ProfileGeneration[n_]:=Module[{},
    If[Length[good[]]=!= 0 ,
        Print["primo vincitore a gen: " , Extract[good[],{1,1}]];
        (*lavora solo su ultima generazione*)
        posgiusti = Position[stackh[[n]],TargetWord];
        giusti = Extract[gens[[n]] , posgiusti];
        valgiusti = Map[EvalInd,giusti];
        Print[valgiusti];
    ,
    Print["NESSUN VINCITORE"]
    ];
];

(*restituisce {generazione,numerogiusti per generazione} *)
popgiusti[]:= Module[{stackdist},
    posgood = good[];
    numpos = Map[Append[#1[[1 ;; -2]], 2] &, posgood];
    nums = Map[Extract[stackdisth,#1 ]&,numpos];
    giusgen = Map[Extract[#1,{1}]&,posgood];
    Return[Table[{giusgen[[k]] , nums[[k]]},{k,Length[giusgen]}]];

];

(*fai il profilo di una run*)
(*e salvalo su un file di nome filename*)
ProfileRun[filename_String]:=Module[{prefix,finaldest,stream},
    prefix = "results/";
    finaldest = prefix<>filename;
    stream = OpenWrite[finaldest];


    WriteString[stream,"popolazione totale:\n"];
    Write[stream,Nind];
```

```
    WriteString[stream,"parametri:\n"];
    WriteString[stream,"{rlw,dsw,riw,bdw,stw,msl,dsl,mdl,ddl}\n"];
    Write[stream,{rlw,dsw,riw,bdw,stw,msl,dsl,mdl,ddl}];

    WriteString[stream,"popolazione di elementi perfetti: \n"];
    Write[stream,perfetti[]];

    WriteString[stream,"popolazione di elementi giusti: \n"];
    Write[stream,popgiusti[]];

    WriteString[stream,"numero runs:\n"];
    Write[stream,gennumber];

    WriteString[stream,"MAXDU:\n"];
    Write[stream,MAXDU];

    WriteString[stream,"MAXSTEP:\n"];
    Write[stream,MAXSTEP];

    WriteString[stream,"\n\n"];
    Write[stream,PlotTrends[]];

    Close[stream];
];

PR[filename_String]:=ProfileRun[filename];

(* fai il profilo della run e stampalo a video *)
PR[]:= Module[{},
    If[Length[Position[gens,unperfetto]]=!=0,
        Print["ATTENZIONE INDIVIUDO PERFETTO TROVATO!!!"];
        Print["posizione:",Position[gens,unperfetto]] ,
        Null
    ];
    Print["popolazione di elementi giusti"];
    Print[popgiusti[]];
    Print["parametri:"];
    Print["{rlw,dsw,riw,bdw,stw,msl,dsl,mdl,ddl}"];
    Print[{rlw,dsw,riw,bdw,stw,msl,dsl,mdl,ddl}];
    Print["numero runs:"];
    Print[gennumber];


    PlotTrends[]


];

(* salva individuo su un file*)
salvaindividuo[ind_ , filename_String]:=Module[{prefix,finaldest,stream},
    prefix = "results/";
    finaldest = prefix<>filename;
    stream = OpenWrite[finaldest];
    Write[stream,ind];
    Close[stream];
];


(*restituisci coppie {generazione, numero perfetti}*)
perfetti[]:=Module[{posperf,subs,toret,posgood,veryperf,splitted},

    posperf = Sort[Position[gens,unperfetto]];
    posgood = Position[posperf,x_/;Length[x] ===2];
    veryperf = Extract[posperf,posgood];
    splitted = Split[veryperf,#1[[1]] == #2[[1]] &];
    toret = {#[[1]][[1]], Length[#]} & /@ splitted;

    Return[toret];
];
```

References

[1] Wolfram Research, Inc.: *Mathematica Edition: Version 8.0*, 2010.

[2] Melanie Mitchell: *An Introduction to Genetic Algorithms* , 1999.