UNIVERSITATEA POLITEHNICA BUCURESTI
FACULTATEA DE AUTOMATICA SI CALCULATOARE

Algoritmi genetici pentru rezolvarea problemelor prin evolutie si co-evolutie

Coordonator proiect:
Prof. dr. ing. **Adina Magda Florea**

Absolvent:
OSTAFIEV Sorin

2003

Cuprins

1. Introducere

Ideea de baza prezenta in aceasta lucrare se refera la aplicarea algoritmilor evolutivi pentru probleme din ce in ce mai complexe. De asemenea, notiuni clare de modularitate trebuie introduse pentru a oferi metode eficiente care sa permita solutiilor sa evolueze sub forma unor sub-componente interactive co-adaptate. Un exemplu de asemenea natura este educarea comportamentului, asa cum se poate intalni in domeniul roboticii, unde un comportament complex poate fi descompus in comportamente simple. Ceea ce face gasirea solutiilor la probleme de acest fel deosebit de dificila este faptul ca componenetele sistemului sunt puternic dependente una fata de alta. Din aceasta cauza co-adaptarea este o cerinta critica pentru evolutia lor.

Exista doua mari motive datorita carora algoritmii evolutivi clasici nu sunt intru totul adecvati pentru rezolvare unor astfel de probleme. Primul, populatia de indivizi evoluata de acesti algoritmi are o puternica tendinta de convergenta, proportional cu numarul de generatii. Al doilea, indivizii evoluati pe algoritmii traditionali reprezinta de obicei solutii complete care sunt evaluate independent. Deoarece nu sunt modelate interactii intre membrii populatiei, nu exista presiune evolutiva pentru aparitia co-adaptarii. Pentru a complica si mai mult problema, se poate sa nu stim de la inceput cate sub-componente ar trebui sa fie sau ce rol sa asociem fiecarei componente. Dificultatea apare in gasirea de solutii computationale ale paradigmei noastre evolutioniste in care componenetele evolueaza mai mult singure decat asistate. O problema importanta este identificarea si reprezentarea unor asemenea sub-componenete, furnizarea unui mediu in care ele pot interactiona si co-adapta si impartirea sarcinilor pentru rezolvarea problemei in asa fel incat evolutia sa se desfasoare fara interventia umana.

2. Evoluarea sub-componentelor co-adaptate

Pentru a extinde modelul computational al evolutiei trebuie sa avem in vedere anumite probleme precum interdependenta intre subcomponente, descompunerea problemei initiale, asocierea rolurilor si controlul diversitatii.

Problema descompunerii consta in determinarea numarului de subcomponenete si rolul pe care-l va juca fiecare. Pentru anumite probleme, o descompunere aproximativa poate fi cunoscuta inca de la inceput. Sa consideram o problema de optimizare a unei functii cu k variabile independente. O descompunere rezonabila ar fi impartirea problemei in k sarcini, fiecare asociata unei singure variabile. Oricum, exista multe probleme pentru care avem prea putina informatie sau deloc, cu privire la numarul sau rolurile sub-componenetelor.

A doua problema se refera la evolutia sub-componentelor interdependente. Daca o problema poate fi descompusa in subprobleme independente intre ele, in mod evident fiecare poate fi rezolvata independent. Din punct de vedere grafic, ne putem imagina fiecare sub-componenta progresand pentru a atinge cea mai inalta pozitie, neluand in calcul pozitiile celorlalte sub-componente. Din nefericire, multe probleme pot fi descompuse doar in sub-componente cu dependente complexe intre ele. Selectia naturala va rezulta in co-adaptare intr-un asemenea spatiu doar daca interdependentele intre sub-componente sunt modelate.

A treia problema este determinarea contributiei fiecarei sub-componente la solutia finala ca un intreg. Aceasta este des intalnita in teoria jocurilor.

A patra problema consta in mentinerea diversitatii indivizilor din populatie suficient de mult de-a lungul generatiilor pentru a fi siguri ca spatiul solutiilor a fost suficient de bine explorat. De obicei, odata ce o solutie buna a fost gasita, algoritmul genetic se termina, cel mai bun individ reprezentand solutia gasita. In paradigma co-evolutiva, cu toate ca unele sub-componente pot fi mai puternice decat altele, luate individual, alti indivizi, tinand cont de contributia lor la rezultat si luandu-i ca un tot, poti fi prezenti in solutia finala.

3. Algoritmii genetici

3.1. Introducere

Algoritmii genetici sunt proceduri de calcul robuste si adaptive modelate pe mecanismul sistemului de selectie naturala. Algoritmii genetici se comporta ca o metafora biologica si incearca sa emuleze cateva dintre procesele observate in evolutia naturala. Ei sunt vazuti ca tehnici de cautare si optimizare aleatoare dar structurate. Ei isi exprima abilitatea prin exploatarea eficienta a informatiei trecute, informatie ce va fi folosita in crearea de noi urmasi cu performante imbunatatite. Algoritmii genetici sunt executati iterativ pe o multime de solutii codificate, numita populatie, cu trei operatori de baza: *selectia/reproducerea*, *incrucisarea* si *mutatia*. Fiecare dintre aceste solutii codificate sunt referite ca indivizi ai populatiei sau cromozomi.

Pentru a gasi solutia optima a unei probleme, un algoritm genetic incepe cu o multime data (sau generata aleator) de solutii (cromozomi) si evolueaza multimi diferite dar mai bune de solutii intr-un ciclu al iteratiei. In fiecare din aceasta iteratie (numita de aici inainte *"generatie"*) functia de evaluare (functie ce masoara un criteriu de fitness) determina modul in care o solutie se apropie de solutia cea mai buna si, pe baza aceste evaluari, unele dintre ele (numite cromozomi parinti) sunt alese pentru reproducere. Este de asteptat faptul ca numarul de copii reproduse dupa un individ parinte sa fie direct proportional cu valoarea sa de fitness, incorporand astfel procedura de selectie naturala, pana la anumite limite. Procedura selecteaza cromozomi mai buni (cu o valoare mai mare a fitness-ului) si ii elemina pe cei rai. De aceea, performanta unui algoritm genetic depinde foarte mult de criteriul de evaluare a valorii fitness-ului. Pe acesti cromozomi parinti selectati sunt aplicati operatori genetici si astfel se genereaza cromozomi noi (urmasi).

Algoritmii genetici conventionali iau in considerare doar valoarea fitness-ului cromozomului in cauza pentru a-i masura aptitudinea pentru selectia in generatia urmatoare, adica fitness-ul unui cromozom este o functie de o valoare a functiei de evaluare. Fitness-ul unui cromoz **x** este **g[f(x)]**, unde **f(x)** este functia de evaluare, iar **g** este o alta functie care da valoarea fitness-ului primind o valoare **f(x)**. De aceea, un algoritm genetic conventional nu face discriminare intre doi urmasi identici, unul produs din parinti mai buni (cu o valoare mai

mare a fitness-ului) si altul din parinti comparabil mai slabi (cu o valoare mica a fitness-ului). In natura, in mod normal un urmas este mai potrivit daca stramosii sai (parintii) sunt mai buni, cu alte cuvinte un urmas poseda o capacitate mai mare de a se descurca mai bine in mediul sau daca apartine unei familii mai bune (starmosi bine adaptati). Deci, valoarea fitness-ului unui individ depinde si de valoarea fitness-ului stramosilor sai, in plus fata de propria valoare. Aceasta este probabil intuitia de baza pentru a da mai multa greutate cromozomilor mai bine adaptati (cu o valoare mai mare a fitness-ului) sa produca mai multi urmasi pentru generatia urmatoare.

3.2. Principii si caracteristici de baza

Algoritmii genetici intentioneaza sa mimeze cateva dintre procesele observate in evolutia naturala in urmatoarele feluri:

Evolutia este un proces care opereaza mai degraba asupra codificarii entitatilor biologice decat asupra fiintelor vii, mai precis evolutia are loc la nivelul cromozomilor.

Similar, algoritmii genetici opereaza pe codificarea solutiilor posibile (numite cromozomi) ale problemei prin manipularea sirurilor de caractere ale unui alfabet.

Selectia naturala este legatura dintre un cromozom si performantele evolutiei sale (masurate pe versiunea decodificata). Natura se supune principiului darwinian “supravietuirea celui mai bun”; cromozomii cu valori mai mari ale fitnessului se vor reproduce, in medie, mai des decat cei cu valori mici ale fitnessului.

In algoritmii genetici si selectia cromozomilor incearca sa mimeze aceasta procedura. Cromozomii mai adaptati se vor reproduce mai des in dezavantajul celor mai putin adaptati.

Evolutia biologica nu are memorie. Aici nautra actioneaza ca mediu si entitatile biologice sunt modificate pentru a se adapta in mediul lor. Tot ceea ce natura cunoaste despre evolutia cromozomilor buni este continut intr-un set de cromozomi ai indivizilor curenti si in procedura de decodificare a cromozomilor.

Asemanator, si algoritmii genetici opereaza orbi pe cromozomi. Ei folosesc doar informatia functiei de evaluare care actioneaza ca mediu. Bazandu-se pe aceasta informatie fiecare cromozom este evaluat si in timpul procesului de

selectie se da o importanta mai mare alegerii cromozomilor cu o valoare mai mare a fitness-ului.

Ca si in evolutia naturala, se introduce variatia in entitatile algortimilor genetici in timpul reproducerii. Incrucisarea si mutatia sunt operatorii de baza pentru reproducere.

Evolutia bazata pe algoritmi genetici porneste de la o multime de indivizi (multimea solutiilor initiale pentru functia de optimizat) si avanseaza din generetie in generatie prin operatii genetice. Inlocuirea unei populatii vechi cu una noua este cunoscuta sub numele de generatie atunci cand se inlocuiesc toti membrii unei populatii cu altii noi. O alta tehnica de reproducere inlocuieste unul sau mai multi indivizi la un moment dat, in loc sa inlocuiasca toata popuatia. Dupa cum s-a mentionat mai sus, algoritmul genetic are nevoie doar de o functie de evaluare ce actioneaza ca mediu pentru a observa potrivirea sau adaptabilitatea solutiilor (cromozomilor) derivate. In Figura 1 este prezentata o diagrama schematica a structurii de baza a unui algoritm genetic:

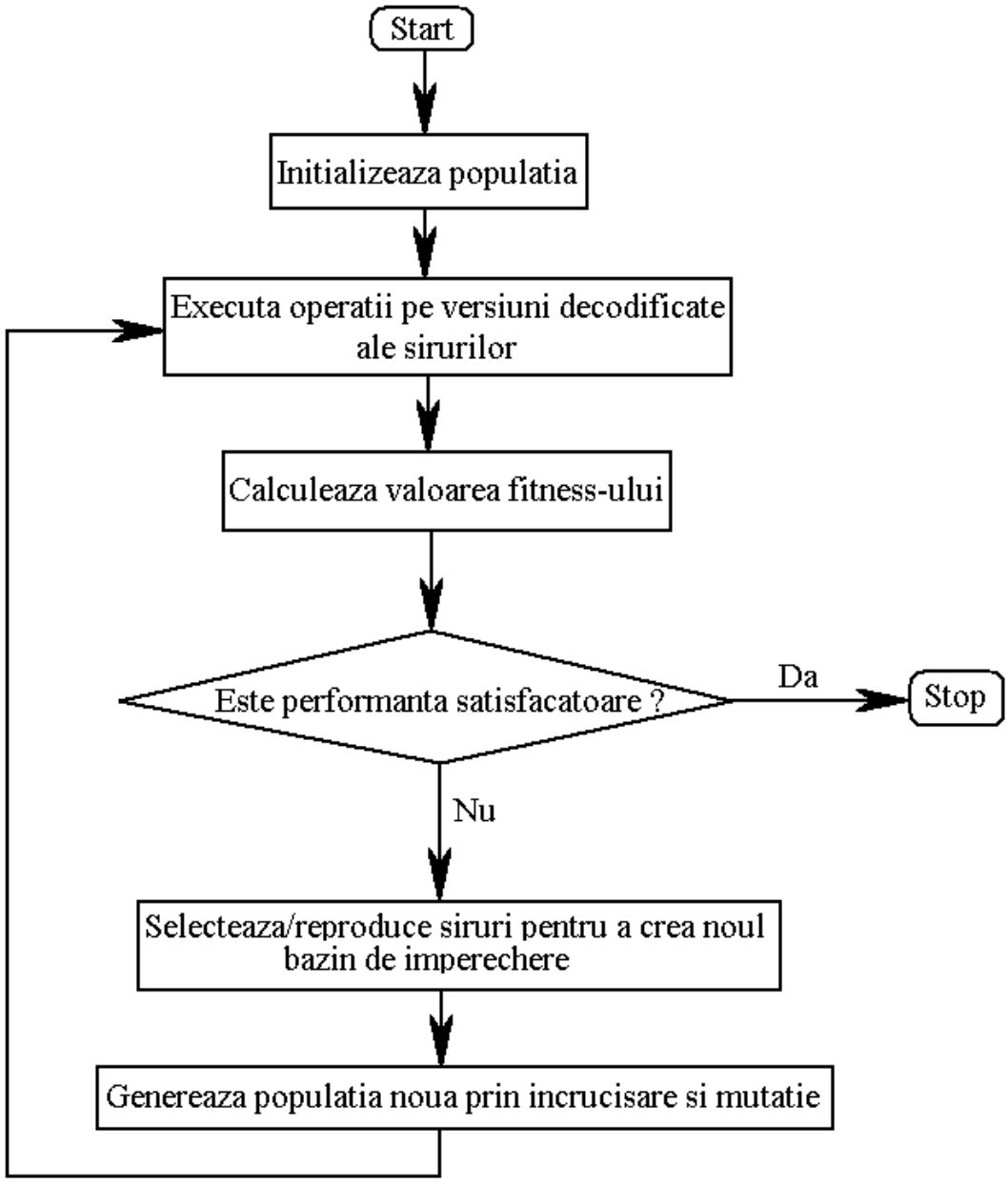


FIGURA 1 – Pasii de baza ai unui algoritm genetic

Un algoritm genetic consta tipic din urmatoarele componente:

O populatie de siruri sau de solutii posibile codificate (referite biologic ca cromozomi)

Un mecanism de codificare a solutiilor posibile (cel mai adesea un sir binar)

O functie de evaluare a fitness-ului cu tehnica asociata ei

Procedura de selectie/reproducere

Operatori genetici (incrucisare si mutatie)

Probabilitati de a efectua operatii genetice

Sa descriem pe scurt aceste componente:

Populatia -- Pentru a rezolva o problema de optimizare, un algoritm genetic porneste cu reprezentarea cromozomiala (structurala) a unei multimi de parametri $\{x_1, x_2, \ldots, x_p\}$. Multimea de parametri va fi codificata ca un sir de lungime finita peste un alafabet de lungime finita. De obicei cromozomii sunt siruri compuse din 0 si din 1. Spre exemplu, fie multimea $\{ a_1, a_2, \ldots, a_p \}$ o instanta a multimii de parametri si fie reprezentarea binara a lui $a_1, a_2, \ldots, a_p$ 10110, 00100, …, 11001. Atunci sirul 1011000100…1101 este o reprezentare cromozomiala a multimii de paramteri $\{ a_1, a_2, \ldots, a_p \}$. Este evident ca numarul de cromozomi (siruri) diferiti este 2^l, unde l este lungimea sirului. De fapt, fiecare cromozom este o instanta a unei solutii posibile. O multime de astfel de cromozomi intr-o generatie se numeste populatie. Marimea unei populatii poate varia de la o generatie la alta sau poate fi constanta. De obicei populatia initiala se alege in mod aleator.

Mecanismul de codoficare/decodificare – Este mecanismul de conversie a valorilor parametrilor unei posibile solutii in siruri binare rezultand reprezentari cromozomiale. Daca solutia unei probleme depinde de p parametri si daca dorim sa codificam fiecare parametru cu un sir binar de lungime q, atunci lungimea fiecarui cromozom va fi $p \times q$. Decodificarea este operatiunea inversa codificarii.

Functia de evaluare a fitness-ului si tehnica asociata ei – Functia de evaluare a fitness-ului este aleasa in functie de problema. Alegerea se face in asa fel incat sirurile mai potrivite, mai adaptate (posibile solutii) sa aiba o valoare mai mare a fitness-ului. Aceasta este singura modalitate dupa care se alege un cromozom pentru a se reproduce in generatia urmatoare.

Procedura de selectie/reproducere – Procesul de selectie/reproducere copiaza siruri individuale (numite cromozomi parinti) intr-o tentativa de noua populatie (numita bazinul de imperechere) pentru operatii genetice. Numarul copiilor repoduse pentru generatia urmatoare de un individ este de asteptat sa fie direct proportional cu valoarea sa de fitness, imitand astfel procedura de selectie naturala pana la un anumit punct. Cele mai frecvente proceduri de selectie sunt selectia prin roata ruletei si selectia lineara.

Operatorii genetici se aplica pe cromozomii parinti si sunt generati noi cromozomi (numiti urmasi). In cele ce urmeaza se vor descrie cei mai des folositi operatori genetici.

Incrucisarea – Scopul principal al incrucisarii este schimbul de informatie intre cromozomi parinti alesi in mod aleator pentru a nu pierde nici o informatie importanta (distrugere minima a structurilor cromozomiale ce sunt selectate pentru operatii genetice). De fapt se combina materialul genetic a doi cromozomi parinti pentru a produce urmasi in noua generatie. Operatie de incrucisare poate fi vazuta ca un proces in trei pasi. Intr-o prima faza, din bazinul de imperechere sunt alese perechi de cromozomi (numite perechi pentru reproducere). Al doilea pas determina, pe baza probabilitatii de incrucisare (generandu-se un numar aleator in intervalul [0, 1] si verificandu-se daca acesta este mai mic decat probabilitatea de incrucisare) daca aceste perechi de cromozomi vor fi incrucisate sau nu. In al treilea pas se efectueaza interschimbul de segmente cromozomiale intre perechile alese. Numarul segmentelor care se vor schimba si lungimea acestora depinde de tipul de incrucisare folosit. Unele dintre cele mai folosite tehnici de incrucisare sunt incrucisarea printr-un punct, incrucisarea prin doua puncte, incrucisarea prin puncte multiple, incrucisarea uniforma, incrucisarea shuffle-exchange. Pentru a ilustra modul in care sunt schimbate segmente din cromozomii parinti, sa consideram tehnica de incrucisare printr-un punct. Fie pozitia k selectata in mod aleator uniform intre 1 si l-1, unde l este lungimea sirului (mai mare decat 1). Cele doua noi siruri sunt create prin interschimbarea tuturor caracterelor de la pozitia (k + 1) pana la l. Fie :

a = 11000 10101 01000 … 01111 10001
b = 10001 01110 11101 … 00110 10100

doua siruri (parinti) selectate pentru operatia de incrucisare si fie numarul generat aleator egal cu 11 (unsprezece). Atunci, cei doi urmasi produsi (prin interschimbarea tutror caracterelor dupa pozitia 11) vor fi:

a' = 11000 10101 01101 … 00110 10100
b' = 10001 01110 11000 … 01111 10001

Mutatia – Scopul principal al mutatiei este de a introduce divesitate genetica in populatie. Cateodata ea ajuta la recastigarea informatiei pierdute in generatiile anterioare. Ca si in sistemul genetic natural, mutatia in algoritmii genetici este facuta ocazional. Se alege o pozitie aleatoare intr-un sir ales aleator si este inlocuita cu un alt caracter din alfabet. In cazul reprezentarii binare mutatia neaga valoarea bitului si este cunoscuta sub numele de mutatie de bit. De exemplu, sa presupunem ca al treilea bit al sirului *a* de mai sus a fost selectat pentru mutatie. Atunci, sirul transformat dupa mutatie va fi:

11100 10101 01000 … 01111 10001

In mod normal rata mutatiei este tinuta fixa. Pentru a sustine diversitatea (care poate fi pierduta datorita incrucisarii si a ratei foarte mici a mutatiei) intr-o populatie s-a propus o tehnica numita *mutatie adaptiva*, unde probabilitatea de a efectua o mutatie este facuta sa creasca (in loc de a fi tinuta fixa) pentru a creste omogenitatea genetica intr-o populatie. Mutatia nu da intotdeauna rezultate meritorii. Valori inalte ale ratei de mutatie pot conduce cautarea genetica intr-una aleatoare. Ea poate schimba valoarea unui bit important si de aceea poate afecta convergenta rapida catre o solutie buna. Mai mult, mutatia poate chiar incetini procesul de convergenta in stagiile finale ale algoritmilor genetici. In [Bhandari si Pal, 1994] a fost propusa o noua tehnica de mutatie cunoscuta sub numele de *mutatie directionata*.

Probabilitati pentru efectuarea operatiilor genetice – Probabilitatea de a efectua operatia de incrucisare este aleasa intr-un asemenea mod astfel incat recombinarea sirurilor puternice (cromozomi cu valori mari ale fitness-ului) creste fara intrerupere. In general, probabilitatea de incrucisare se afla undeva intre 0.6 si 0.9.

Deoarece mutatia intervine ocazional, este clar faptul ca probabilitatea de a efectua o mutatie va fi una foarte mica. Aceasta valoare se afla undeva intre 0.0001 si 0.01.

Probabilitatile de incrucisare si de mutatie pot fi tinute fixe de-a lungul operatiilor unui algoritm genetic sau pot fi adaptate (determinate automat in functie de mediu) pentru a imbunatati performantele, daca e posibil.

Intr-un algoritm genetic standard nu se pastreaza solutia cea mai buna obtinuta pana la un moment dat, marind sansele de a pierde obtinerea celor mai

bune solutii posibil. Strategiile *elitiste* sunt cele care vin sa depaseasca acesta problema prin copierea celui mai bun membru al fiecarei generatii in generatia urmatoare. Desi aceasta strategie poate mari viteza de dominare a unei populatii de catre un sir puternic (sir cu o valoare mare a fitness-ului), ea mareste performantele unui algoritm genetic folosind inlocuirea generatiilor. Recent au fost introduse conceptele de algoritmi genetici distribuiti si de algoritmi genetici paraleli.

Caracteristici distinctive – Urmatoarele caracteristici faciliteaza algoritmii genetici sa-si mareasca aplicabilitatea peste cele mai multe din strategiile de optimizare si de cautare:

Algoritmii genetici lucreaza simultan pe mai multe puncte si nu pe unul singur, ceea ce ajuta la introducerea unui paralelism implicit puternic in procedura computationala. Aceasta caracteristica previne si faptul ca algoritmul genetic sa nu se "impotmoloeasca" intr-un minim local.

Algoritmii genetici lucreaza cu o multime de parametri codificati, nu cu parametrii insusi. De aceea, rezolutia spatiului de cautare posibil poate fi controlata prin varierea lungimii codificarii parametrilor.

In algoritmii genetici nu e necesar ca spatiul de cautare sa fie continuu (spre deosebire de metodele de cautare bazate pe calcul).

Algoritmii genetici sunt orbi. Ei folosesc doar informatia functiei de evaluare si nu au nevoie de informati auxiliare, cum ar fi derivari din functia de optimizare.

Algoritmii genetici folosesc reguli de tranzitie probabilistice, nu deterministice.

Recent, algoritmii genetici gasesc aplicatii foarte raspandite in rezolvarea problemelor care cer cautari eficiente si practice in cercurile de afaceri, de stiinta si cele ingineresti, de exemplu: sinteza arhitecturii retelelor neuronale, problema comis-voiajorului, a colorarii grafurilor, optimizari numerice, sisteme de clasificare, recunoastere de forme, etc.

4. Problema acoperirii sirurilor

In cele ce urmeaza vom considera problema acoperirii sirurilor ca problema de referinta, in mare parte deoarece furnizeaza un mediu relativ simplu in care descompunerea si evolutia cooperativa pot fi usor explorate. Problema consta in gasirea unui set de N vectori binari care se "potrivesc" cat se poate de bine pe un alt set de K vectori binari si unde de regula K este mult mai mare decat N. Le vom numi pe acestea set de gasit si respectiv set sursa. Dandu-se faptul ca K este mult mai mare decat N, setul de gasit trebuie sa contina sabloane care se regasesc in multiple siruri din setul sursa pentru a acoperi setul sura in mod optim, cu alte cuvinte, setul de gasit trebuie sa generalizeze. Calitatea potrivirii intre doi vectori x si y de lungime L se noteaza cu S si se determina simplu prin numararea pozitiilor identice din cei doi vectori. Pentru a calcula calitatea potrivirii unui set M, aplicam pe rand fiecare element din el asupra setului destinatie si apoi calculam media aritmetica a acestora.

De asemnea in acest caz, dimensiunea setului de gasit poate sau nu sa fie specificata de la inceput. Problema descompunerii in acest context consta in determinarea marimea setului de gasit si acoperirea fiecarui set posibil asupra setului destinatie.

In particular, algoritmul evolutiv utilizat in acest caz este un algoritm genetic. In toate cazurile, initializarea populatiilor se face aleator , se foloseste un crossover in 2 puncte la o rata prestabilita si o mutatie ce consta in schimbarea starii unui bit tot la rata prestabilita depinzand de lungimea sirurilor, iar selectia indivizilor se face proportional cu fitness-ul indivizilor.

5. Reprezentarea si codificarea cunostintelor

Sistemele bazate pe algoritmi genetici sunt capabile sa invete din instante structurate ale conceptelor. In particular, o instanta a unui concept este reprezentata ca o colectie de obiecte, fiecare din ele fiind descris de un vector de atribute. Pentru tratarea acestui fel de reprezentare este mai adecvat un limbaj

pentru descrierea de concepte in logica cu predicate de ordinul I, adica un limbaj cu variabile.

Limbajul de descriere al conceptelor ***L***, folosit de sistem, este un limbaj cu predicate de ordinul I. Mai precis, ***L*** este un limbaj cu clause Horn, in care termenii pot fi variabile sau disjunctii de constante si negatia apare intr-o forma restrictiva. Disjunctia interna a fost larg folosita in invatarea conceptelor deoarece permite exprimarea compacta a ipotezelor inductive. Un exemplu de expresie atomica continand un termen disjunctiv este "forma(x, patrata $\vee$ rotunda)", care este echivalenta semantic cu "forma(x, patrata) $\vee$ forma(x, rotunda)", dar este mai naturala.

In sistemul propus, o formula atomica de aritate **m** are forma sintactica $P(x_1, x_2,.., x_m, \mathbf{K})$, unde $x_1, x_2,.., x_m$ sunt variabile, iar termenul **K** este o disjunctie de termeni constanti, $[v_1, v_2,.., v_n]$ sau negatia unei astfel de disjunctii, $\neg[v_1, v_2,.., v_n]$. Cel mult un termen poate fi o expresie disjuncta (pozitiva sau negativa), in timp ce toate celelalte trebuie sa fie variabile. Iata cateva exemple de expresii atomice bine formate:

$$forma(x_1, [\text{patrata, rotunda}]), forma(x_1, \neg[\text{triunghiulara, octogonala}])$$
$$distanta(x_1,x_2, [5,6,7]), superior(x_1,x_2)$$

De notat faptul ca expresia $forma(x_1, \neg[\text{triunghiulara, octogonala}])$ este echivalenta semantic cu $\neg forma(x_1, [\text{triunghiulara}]) \wedge \neg forma(x_1, [\text{octogonala}])$. De aceea, negatia interna a unui disjunct este echivalenta cu conjunctia atomilor negati in forma clauzala. Cand o formula φ este evaluata pe o instanta ξ a conceptului, fiecare variabila in φ va trebui sa fie legata de un obiect ce apare in descrierea lui ξ. Apoi, predicatele ce apar in φ sunt evaluate pe baza atributelor obiectului de care e legata variabila. Deoarece legatura intre variabilele din φ si obiectele din ξ se poate alege in multe moduri, se spune ca φ este adevarat pentru ξ daca si numai daca exista macar o alegere astfel incat toate predicatele ce apar in φ sunt adevarate. Modul in care semantica predicatelor este evaluata relativ la atributele obiectului trebuie definit explicit de catre utilizator inainte de a rula REGAL pe o aplicatie specifica. Pentru a clarifica acest lucru, sa presupunem ca o instanta a conceptului este compusa din trei obiecte:

o_1:<culoare = neagra, pozitie = 1, marime = 4, inaltime = 3>
o_2:<culoare = verde, pozitie = 2, marime = 1, inaltime = 1>
o_3:<culoare = albastra, pozitie = 3, marime = 10, inaltime = 5>

Fiecare obiect este descris de patru atribute: *culoare*, *pozitie*, *marime* si *inaltime*. Sa mai presupunem ca semantica pentru predicatele culoare(x_1,

[galbena, verde]), distanta(x_1, x_2, [2, 3, 4]), mai_mare(x_1,x_2) si inalt(x_1) este definita de expresiile:

membru(*culoare*(x_1), **K**);
membru(| *pozitie*(x_1) – *pozitie*(x_2) |, **K**);
marime(x_1)>*marime*(x_2);
inaltime(x_1)>6.

Atunci, predicatul *culoare*(x_1, [galbena, verde]) va fi adevarat cand variabila x_1 va fi legata de obiectul o_2 si va fi fals pentru orice alta alegere a legaturii (termenul **K** este legat la [galbena, verde]). Asemanator, *distanta*(x_1, x_2, [2, 3, 4]) este adevarat doar cand x_1 este legat de o_3 si x_2 de o1 (termenul **K** este legat la [2, 3, 4]), *mai_mare*(x_1, x_2) este adevarat pentru trei alegeri ale legaturilor <x_1=o_1, x_2=o_2>,<x_1=o_3, x_2=o_1>,<x_1=o_3, x_2=o_2> si *inalt*(x_1) este mereu fals.

In cele ce urmeaza se va prezenta modul in care descrierea conceptelor in limbajul ***L*** poate fi reprezentata pe un sir de biti de lungime fixa.

5.1. Ajustarea procesului de inductie folosind sabloane de limbaj

Totii algoritmii de invatare folosesc intr-un fel sau altul constrangeri, fie ele implicite sau explicite. O metoda larg raspandita in sisteme care integreaza inductia si deductia in structuri bazate pe logica predicatelor de ordinul I consta in folosirea unor constructii metasemantice pentru definirea unui set de formule admisibile ce urmeaza a fi explorate. Algoritmul propus limiteaza spatiul ipotezelor folosind un sablon de limbaj. Neformal, un sablon de limbaj este o formula Λ apartinand limbajului ***L***, astfel incat fiecare descriere de concept conjunctiva admisibila poate fi obtinuta din Λ prin stergerea unor constante din disjunctiile interne care apar in ea.

Inainte de a da o definitie formala a lui Λ, sa reamintim cateva noti uni logice introduse mai sus, pentru a defini conceptul de *forma completa* pentru un termen disjunctiv si pentru un predicat. Dandu-se o formula $\varphi(x_1, x_2,.., x_n)$, a evalua valoarea de adevar a formulei intr-un univers U inseamna sa se caute o legatura intre variabilele x_1, x_2,.., x_n si niste constante a_1, a_2,.., a_n, definite in U, astfel incat $\varphi(a_1, a_2,.., a_n)$ sa fie adevarat. De notat faptul ca in invatarea supervizata a conceptelor fiecare instanta de invatare constituie un univers specific. Putem da urmatoarea definitie:

Definita 2 : Fie un predicat $P(x_1, x_2,.., x_m, \mathbf{K})$, unde **K** este o disjunctie pozitiva de constante $[v_1, v_2,.., v_n]$. Se spune ca P este in *forma completa* daca multimea $[v_1, v_2,.., v_n]$ este in asa fel aleasa incat P poate fi satisfacut pentru orice legare a variabilelor $x_1, x_2,.., x_n$.

Definitia 2 stabileste faptul ca un predicat P care contine un termen disjunctiv in forma completa este adevarat pentru orice instanta a setului de invatare. Considerand din nou predicatul *culoare*(x, **K**), acesta va fi in forma completa daca **K** este multimea tuturor culorilor posibile. Pentru distinge predicatele in forma completa de celelalte, acestea vor fi scrise cu caractere cursive .

Definitia 3 : Un sablon de limbaj Λ este o formula a limbajului ***L*** care contine cel putin un predicat in forma completa.

Scopul unui sablon Λ este de a defini problema invatarii pentru algoritmul propus. Predicatele ce apar in Λ si nu sunt in forma completa au rolul de constrangere si trebuie sa fie satisfacute de o alegere specifica a legarii variabilelor din Λ. Dimpotriva, predicatele in forma completa (adevarate prin definitie) sunt folosite pentru a defini spatiul de cautare care caracterizeaza problema de invatare. Stergerea unei constante dintr-un termen complet pozitiv ce apare intr-un predicat P face ca predicatul P sa fie mai specific. Mai general, orice formula obtinuta prin renuntarea la unele constante din Λ este mai specifica decat insusi Λ. Dandu-se un sablon Λ, spatiul de cautare explorat de algoritm este restrans la multimea $\mathbf{H}(\Lambda)$ a formulelor care pot fi obtinute prin stergerea unor constante din termenii completi ce apar in Λ.

Este usor de observat faptul ca, avand o formula conjuctiva Λ, ea poate fi scrisa ca o conjunctie de doua subformule: Λ_n, compusa din predicate necomplete si Λ_c ,compusa din predicate complete. Sablonul trebuie declarat la inceput prin specificarea separata a lui Λ_n si a lui Λ_c. In particular, formula Λ_n poate fi goala, in timp ce Λ_c trebuie sa contina intotdeauna cel putin un predicat in forma completa.

Datorita faptului ca multimea de constante necesare sa completeze o expresie disjunctiva poate fi una foarte mare (chiar infinita), s-a introdus un simbol constant special "*" in Λ, cu rol de wildcard. Fie $K = [v_1, v_2,.., v_n]$ un termen complet; semnificatia constantei "*" este definita implicit ca $* = \neg [v_1, v_2,.., v_n]$. Cu alte cuvinte, inseamna "orice alta valoare care nu este mentionata explicit in termenul disjunctiv in care apare". Semantica asociata simbolului "*" este locala termenului disjunctiv in care apare si este definita static la declararea sablonului. Dupa aceasta isi mentine semnificatia initiala in toti termenii disjunctivi derivati

din acelasi termen in forma completa. Un exemplu de sablon de limbaj se gaseste in Figura 1, alaturi de doua formule apartinand spatiului de ipoteze asociat.

Sablonul Λ

Λ = greutate(x, [3, 4, 5]) $\wedge$ culoare(x, *[rosie, albastra, *]*) $\wedge$
$\wedge$ forma(x, *[patrata, triunghiulara, circulara, *]*) $\wedge$
$\wedge$ distanta(x, y, *[1, 2, 3, 4, 5, *]*)

Λ_n = greutate(x, [3, 4, 5])
Λ_c = culoare(x, *[rosie, albastra, *]*) $\wedge$ forma(x, *[patrata, triunghiulara, circulara, *]*) $\wedge$
$\wedge$ distanta(x, y, [1, 2, 3, 4, 5, *])

φ_1 = greutate(x, [3, 4, 5]) $\wedge$ culoare(x, [rosie]) $\wedge$ forma(x, $\neg$ [patrata, circulara]) $\wedge$
$\wedge$ distanta(x, y, [1, 2])
φ_2 = greutate(x, [3, 4, 5]) $\wedge$ culoare(x, $\neg$ [rosie]) $\wedge$ distanta(x, y, [1, 2])

FIGURA 2 – Exemplu de sablon de limbaj Λ cu subformulele sale Λ_n si Λ_c. Formulele φ_1 si φ_2 apartin spatiului $\mathbf{H}(\Lambda)$. Toate formulele se considera implicit a fi cuantificate existential.

Este demn de mentionat faptul ca folosirea constantei wildcard "*" duce natural la introducerea de termeni disjunctivi negati. De exemplu, predicatul forma(x, $\neg$ [patrata, circulara]), care apare in φ_1, este doar o rescriere a predicatului forma(x, [triunghiulara, *]), unde * = $\neg$ [patrata, triunghiulara, circulara], dupa definitia din sablon. Mai general, "*" poate fi intotdeauna eliminata din orice ipoteza inductiva prin introducerea unui termen negat. In sfarsit, se mai observa faptul ca se poate renunta la predicatele complete care apar in orice ipoteza inductiva fara a i se modifica semnificatia, fiind o tautologie. De exemplu, in formula φ_2 s-a renuntat la predicatul complet forma(x, *[patrata, triunghiulara, circulara, *]*).

5.2. Transformarea formulelor din logica cu predicate de ordinul I in siruri de biti

O ipoteza inductiva φ, apartinand spatiului $\mathbf{H}(\Lambda)$, corespunde unei reguli de implicatie $\varphi \rightarrow h$, h fiind numele conceptului tinta. Dupa cum se va arata in cele ce urmeaza, formulele din $\mathbf{H}(\Lambda)$ pot fi usor reprezentate ca siruri de biti de lungime fixa. Se poate observa faptul ca doar predicatele complete din Λ_c trebuie sa fie procesate, in timp ce subformula Λ_n trebuie sa apara, implicit, in orice ipoteza inductiva, astfel ca ea nu mai are nevoie sa fie reprezentata. Transformarea unei formule Λ_c intr-un sir de biti de lungime fixa $s(\Lambda_c)$ este imediata, fiecare constanta ce apare intr-un termen in forma completa fiind asociat unui bit in $s(\Lambda_c)$. Pastrand adiacenti bitii corespunzatori literalilor care sunt adiacenti in sablon, fiecare predicat in forma completa va corespunde unui substring specific si, de aceea, decodificarea unui sir de biti este directa.

Reprezentarea semantica a "genelor" in sirul de biti s-a facut dupa cum urmeaza: daca bitul corespunzator unui termen v dat intr-un predicat P este pozitionat pe 1, atunci v apartine disjunctiei interne curente a lui P, pe cand, daca este pozitionat pe 0, nu apartine acelei disjunctii a lui P. Deoarece eliminarea unui termen dintr-o disjunctie interna a unui predicat introduce constrangeri, adica il face mai specific, operatia de transformare a unui 1 in 0 intr-un sir de biti se va numi specializare. Dintr-un motiv contrar, operatia de transformare a unui 0 in 1 intr-un sir de biti se va numi generalizare. Exemple de siruri de biti care codifica formulele din Figura 1 sunt date in Figura 2.

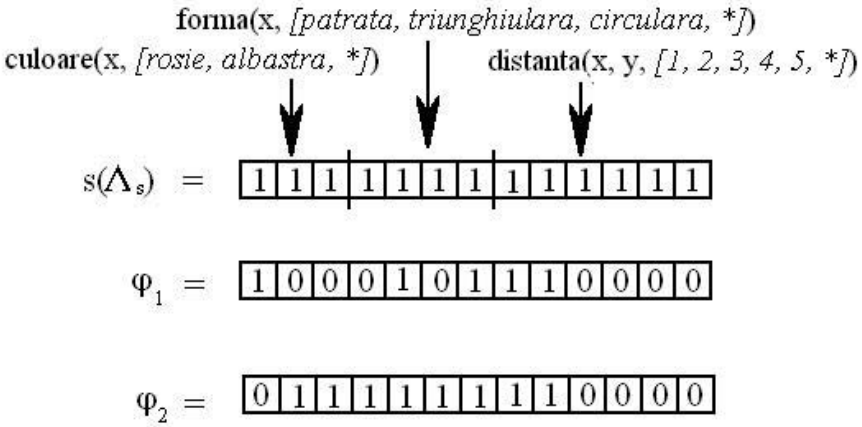


FIGURA 2 – Sirurile de biti corespunzatoare sablonului si formulelor si descrise in Figura 1. Fiecare termen, inclusiv '*', care apare intr-o disjunctie interna are un bit corespondent in sirul de biti.

Sirul de biti caracterizeaza in mod unic o ipoteza inductiva φ si se constituie in informatia procesata de algoritmul genetic. De fapt, in REGAL un individ este reprezentat de o structura mai complexa care contine si informatii procesate de procedura de evaluare. In particular, acesta contine un sir de biti F care specifica multimea de exemple ce satisfac φ, clasa h asignata formulei φ si valoarea de fitness f. In faza de evaluare se incearca potrivirea fiecarui individ nou generat cu toate exemlele de invatare si se umple sirul de biti F. Dupa aceasta se asigneaza clasa h: daca exista un singur concept tinta, clasa este univoc determinata, altfel sistemul asigneaza conceptul care are mai multi reprezentanti in instantele de invatare satisfacute de φ. In final este evaluata valoarea fitness-ului, f, tinand cont de calsa h asignata.

6. Operatorii genetici

In sectiunea ce urmeaza se vor descrie operatorii genetici si functia de evaluare a indivizilor (fitness) folositi in REGAL. Sistemul foloseste operatori de selectie, de incrucisare, de mutatie si de inmultire. Cele mai relevante diferenete fata de un algoritm genetic clasic se afla in operatorii de selectie si de inmultire; operatorii de incrucisare si de mutatie sunt mult mai apropiati de cei clasici.

6.1. Operatorii de incrucisare si de mutatie

Operatorul de incrucisare este special proiectat pentru problema in discutie si consta din: incrucisare din doua puncte, incrucisare uniforma, incrucisare prin generalizare si incrucisare prin specializare. Acest operator se aplica intre doi indivizi din populatie (adica sirurlie de biti din reprezentarea a doua formule φ si ψ).

Incrucisarea din doua puncte functioneaza dupa cum urmeaza: dandu-se doua siruri de biti (siruri parinti), incrucisarea produce urmasi selectand aleator doua puncte in sirul de biti si interschimband apoi bitii care se afla intre aceste doua puncte. In acest fel se schimba o secventa aleatoare de biti intre doi indivizi.

Operatorul de incrucisare uniforma alege mai intai inmod aleator un set de pozitii de biti si apoi produce urmasi prin interschimbarea valorilor bitilor corespunzatori din cei doi parinti.

Incrucisarea prin generalizare si prin specializare necesita explicatii suplimentare. Dupa cu s-a descris in Figura 2, sirul $s(\Lambda_s)$ este impartit in subsiruri, fiecare din ele corespunzand unui predicat specific P. In ambele tipuri de incrucisare, o multime D de predicate este selectata aleator din Λ_s. Incrucisarea prin specializare (generalizare) functioneaza dupa cum urmeaza:

Subsirurile din sirurile parinti s_1 si s_2, corespunzatoare predicatelor neselectate in multimea D, sunt copiate neschimbate in urmasii corespunzatori s_1' si s_2'.

Pentru fiecare predicat $P \in D$, un nou subsir si' este generat prin operatia SI (SAU) logic intre bitii corespunzatori subsirurilor $s_1(P_i)$ si $s_2(P_i)$. Apoi se copie subsirurile s_i' atat in s_1' cat si in s_2'.

Multimea D se creeaza asociind fiecarui predicat Pi o probabilitate egala de a fi selectat. Dupa cum s-a explicat in sectiunea !!!, transformand un 0 in 1 intr-un sir de biti se adauga un termen unei disjunctii interne ce apare in ipoteza inductiva corespunzatoare, aceasta devenind mai generala, adica mai putin constransa. De aici si notiunea de incrucisare prin generalizare, deoarece unramssi care se genereaza sunt mai generali decat parintii lor. Printr-un rationament asemanator se poate explica incrucisarea prin specializare.

In cele ce urmeaza vom nota cu **E** , respectiv cu **C**, multimea exemplelor pozitive, respectiv negative, ale conceptului tinta, si cu E si C cardinalitatea lor. Fie o pereche de siruri (s_1, s_2), generata de procedura de imperechere. Incrucisarea se va aplica cu o probabilitate pc. Apoi se va selecta stochastic modul de incrucisare tinand cont de caracteristicile celor doua siruri. Probabilitatile conditionale p_u pentru incrucisarea uniforma, p_{2pt} pentru incrucisarea prin doua puncte, p_s pentru incrucisarea prin specializare si p_g pentru incrucisarea prin generalizare se asociaza dupa cum urmeaza:

$$\begin{aligned} p_u &= (1 - a \cdot f_n) \cdot b \\ p_{2pt} &= (1 - a \cdot f_n) \cdot (1 - b) \\ p_s &= a \cdot f_n \cdot r \\ p_g &= a \cdot f_n \cdot (1 - r) \end{aligned} \qquad (5.1)$$

In expresia (4.1) a si b iau valori in intervalulu [0, 1] si sunt parametri ajustabili, f_n este valoarea medie normalizata a fitnesului celor doua formule φ_1 si φ_2, definite de sirurile s_1 si s_2:

$$f_n = \frac{f(\varphi_1) + f(\varphi_2)}{2 f_{Max}} \le 1 \qquad (5.2)$$

si r este raportul:

$$r = \frac{\left[n^+(\varphi_1) + n^-(\varphi_1) + n^+(\varphi_2) + n^-(\varphi_2)\right]}{(E + C)^2} \qquad (5.3)$$

unde $n^+(\varphi)$ si $n^-(\varphi)$ reprezinta numarul instantelor pozitive, respectiv negative, acoperite de φ.

Intr-un caz specific, valoarea medie normalizata f_n determina probabilitatea de selectie intre incrucisarea uniforma si cea prin doua puncte, $p_u + p_{2pt} = 1 - af_n$, pe de o parte, sau intre incrucisarea prin generalizare si cea prin

specializare, $p_s + p_g = af_n$, pe de alta. Cand s_1 si s_2 au valori mici ale fitness-ului, adica inca nu sunt ipoteze inductive bune, sansa de a aplica primele doua tipuri de incrucisari este mai mare pentru a exploata putere lor mai mare de explorare. Dimpotriva, valori mari ale fitness-ului privilegiaza folosirea incrucisarii prin specializare si prin generalizare pentru a rafina ipoteza inductiva. Alegerea intre incrucisarea uniforma si cea prin doua puncteeste controlata static de parametrul *b*, pe cand alegerea intre incrucisare prein generalizare si cea prin specializare este controlat a de parametrul *r* care, la randul sau, este evaluat in timp real pe baza exemplelor acoperite de s_1 si de s_2. Astfel, cand acestea acopera multe exemple, este preferata incrucisarea prin specializare pentru a mari sansele de a crea un urmas mai consistent, altfel incrucisarea prin generalizare primeste o sansa mai mare.

In ceea ce priveste operatorul de mutatie, acesta este identic cu cel folosit in algoritmii genetici clasici. El este aplicat urmasilor cu probabilitatea p_m = 0.00001 si poate afecta orice bit din $s(\Lambda_s)$.

Exemplu de operatii genetici **– Pentru a clarifica modul in care operatorii genetici functioneaza vom da un exemplu. Fie Λ sablonul de limbaj din Figura 1 si formulele:**

φ_1 = greutate(x, [3, 4, 5]) $\wedge$ culoare(x, [rosie, albastra]) $\wedge$
$\wedge$ forma(x, $\neg$ [patrata, circulara]) $\wedge$
$\wedge$ distanta(x, y, [1, 2])
φ_2 = greutate(x, [3, 4, 5]) $\wedge$ culoare(x, $\neg$ [albastra]) $\wedge$
$\wedge$ distanta(x, y, [1, 2, 3])

Atunci, aceste formule pot fi transformate, prin sablonul , in indivizii:

i_1 = 1100101110000
i_2 = 1011111111000

In Tabelul 1 este descris efectul operatorilor de incrucisare asupra acestor doi indivizi.

	Parinti	Formule
	1100101110000	culoare(x, [rosie, albastra]), forma(x, $\neg$ [patrata, circulara]) distanta(x, y, [1, 2])
	1011111111000	culoare(x, $\neg$ [albastra]), distanta(x, y, [1, 2, 3])

Xover	Urmasi	Formule
$u_{2,6,7,9}$	1000111110000 1111101111000	culoare(x,[rosie]), forma(x,[triunghiulara, circulara, *]), distanta(x, y, [1,2]) forma(x, [patrata, triunghiulara, *]), distanta(x, y, [1, 2, 3])
$2p_{5-8}$	0100111110000 1011101111000	culoare(x, [albastra]), distanta(x, y, [1, 2]) culoare(x, [rosie, *]), forma(x, [patrata, triunghiulara, *]), distanta(x, y, [1, 2, 3])
$s_{1-3,4-7}$	1000101110000 1000101111000	culoare(x, [rosie]), forma(x, [triunghiulara, *]), distanta(x, y, [1, 2]) culoare(x, [rosie]), forma(x, [triunghiulara, *]), distanta(x, y, [1, 2, 3])
$g_{4-7,8-13}$	1101111111000 1011111111000	culoare(x, [rosie, albastra]), distanta(x, y, [1, 2, 3]) culoare(x, [rosie, *]),distanta(x, y, [1, 2, 3])

Tabelul 1 – Exemplu de operatii de incrucisare. u si 2p semnifica incrucisarea uniforma si cea prin doua puncte, pe cand s si g inseamna incrucisare prin specializare si prin generalizare. Indicii operatorilor exprima care dintre biti au fost alesi de operatori.

Operatorii din Tabelul 1 sunt scrisi in forme de genul $s_{1-3,4-7}$, unde s exprima operatorul, in acest caz cel de incrucisare prin specializare, iar indicii denota bitii afectati de operatorul de incrucisare. Se observa ca in cazul incrucisarii prin doua puncte sunt afectati biti consecutivi, iar in cazul incrucisarii prin specializare si prin generalizare bitii afectati corespund subsirurilor predicatelor alese. De exemplu, $s_{1-3,4-7}$, corespunde operatorului de incrucisare prin specializare aplicat primelor doua predicate din sablomul de limbaj.

6.2. Operatorul de inmultire

Dupa cum se va explica mai jos, operatorul de selectie folosit de REGAL favorizeaza ipoteze ce acopera mai multe instante in setul de invatare. Daca o instanta anume nu e acoperita inca, va fi foarte probabil ca acesta sa genereze dinamic indivizi care sa o acopere. Aceasta facilitate este oferita de operatorul de inmultire, care actioneaza ca o functie care, primind o instanta pozitiva, intoarce o formula ce o acopera. Descrierea abstracta a algotimului de inmultire este urmatoarea:

Inmultire (ξ)

Fie $\xi \in \mathbf{E}$ o instanta pozitiva de invatare
Genereaza aleator un sir de biti s care defineste o formula $\varphi \in H(\Lambda)$
Selecteaza aleator o legare **b** pentru ξ care e compatibila cu $\mathbf{\Lambda_n}$
Pozitioneaza pe 1 cel mai mic numar de biti din s pentru a satisface φ pe ξ
intoarce (φ)

FIGURA 3 – Descrierea abstracta a algoritmului de inmultire.

Ca un exemplu, sa consideram din nou sablonul de limbaj din Figura 1 si sa presupunem ca exemplul ξ de acoperit consta din urmatoarea secventa de trei obiecte:

o_1:<culoare = albastra, pozitie = 1, greutate = 4, forma = patrata>
o_2:<culoare = verde, pozitie = 2, greutate = 1, forma = patrata>
o_3:<culoare = albastra, pozitie = 5, greutate = 10, forma = circulara>

Sa mai presupunem ca legarea **b,** <x = o_1; y = o_3>, satisface constrangerea greutate(x, [3, 4, 5]). Fie sirul s = "10000110101001", corespunzator formulei selectate φ= culoare(x, [rosie]) $\wedge$ forma(x, [triunghiulara, circulara]) $\wedge$ distanta(x, y, [1, 2, *]). Formula φ nu este satisfacuta de ξ. De aceea, sirul s va fi modificat in s'="1101110101001", corespunzator formulei φ' = culoare(x, [rosie, albastra]) $\wedge$ forma(x, [patrata, triunghiulara, circulara]) $\wedge$ distanta(x, y, [1, 2, *]) care este acum satisfacuta de ξ.

Dovedindu-se ca este asigurata compatibilitatea cu constrangerile impuse de sablonul Λ_n, algoritmul de inmultire incearca sa introduca un factor aleator cat mai mare posibil pentru a mari diversitatea genetica a populatiei. Acesta este folosit la inceputul cautarii pentru a initializa populatia si apoi, este apelat fie de operatorlul de selectie, fie ca o alternativa la operatorul clasic de mutatie.

6.3. Operatorul de selectie "sufragiu universal"

Operatorul de selectie este special croit pentru problema invatarii conceptelor. Ideea de baza poate fi explicata printr-o metafora. Formulele conjunctive se constitiuie in "candidati" spre a fi alesi intr-un parlament (populatia), unde exemplele pozitive de invatare sunt electori; un exemplu poate vota pentru una din formulele pe care le acopera. Marea diferenta dintre acest operator si altii propusi pana in momentul de fata este faptul ca indivizii ce vor fi imperecheati nu sunt alesi direct din populatia curenta, ci indirect prin selectarea unui numar egal de exemple pozitive, dupa cum se va descrie mai jos.

Fie populatia $A(t) = \left\{\varphi_1^{x_1(t)}, \ldots, \varphi_m^{x_m(t)}\right\}$ la momentul t o multime de cardinalitate M. A(t) contine m indivizi diferiti (adica formule conjunctive ale limbajului de descriere $\boldsymbol{L}$), fiecare dine ele avand multiplicitatea x_j(t) ($1 \le j \le m$). Fie COV(φ_j) submultimea lui $\mathbf{E}$ acoperita de φ_j. Procedura de selectie functioneaza dupa cum urmeaza: La fiecare generatie t, un numar g * M $\le$ M de exemple este ales aleator din setul de exemple pozitive $\mathbf{E}$. Parametrul g reprezinta proportia de indivizi selectati pentru imperechere. Fiecarui exemplu selectat ξ_k I se asociaza o multime $\mathbf{R}(\xi_k)$ continand formulele din A(t) care acopera exemplul ξ_k asociat. Multimea $\mathbf{R}(\xi_k)$ corespunde unei "roti de ruleta" r_k, impartita in sectoare, fiecare sector fiind asociat unei formule $\varphi_j \in \mathbf{R}(\xi_k)$. Marimea fiecarui sector asociat unei formule φ_j este proportional cu raportul dintre valoarea totala a fitness-ului formulei (adica fitness-ul lui φ_j inmultit cu multiplicitatea sa, x_j(t), in A(t)) si suma valorilor totale ale fitness-ului ale tuturor formulelor ce apar in $\mathbf{R}(\xi_k)$. Pentru fiecare rotire a ruletei r_k se alege o formula invingatoare. Iata un algortim mai formal al celor prezentate:

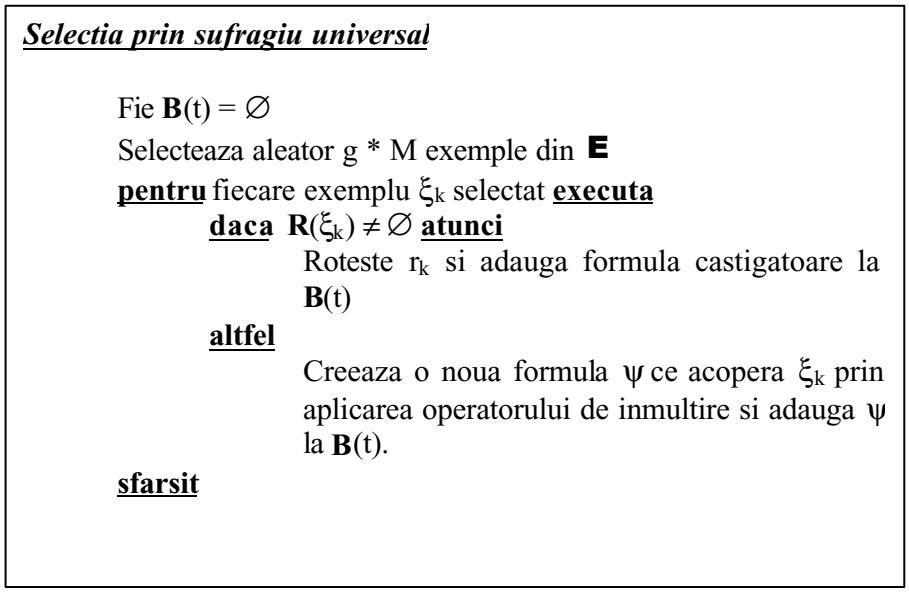


FIGURA 4 – Descrierea abstracta a algoritmului de slectie folosit. M este cardinalitatea globala a populatiilor, g $\in$ [0, 1] este un parametru numit *bresa dintre generatii* .

Figura 5 exprima o reprezentare grafica a ciclului de baza din REGAL. In conformitate cu metafora "parlamentului", la fiecare generatie g * M exemple pozitive trebuie sa-si exprime preferinta, prin "votarea" unuia dintre "reprezentativii" lor invartind propria roata a ruletei. E important de observat faptul ca doar formulele care acopera aceleasi exemple concureaza intre ele si ca exemplele "voteaza" (stochastic) pentru cea mai buna dintre ele. Procesul de selectie favorizeaza formule cu acoperire mai mare (formule ce apar in mai multe roti de ruleta) si formule cu o valoare mai mare a fitness-ului total (formule cu o probabilitate mai mare de a castiga in ruleta in care apar).

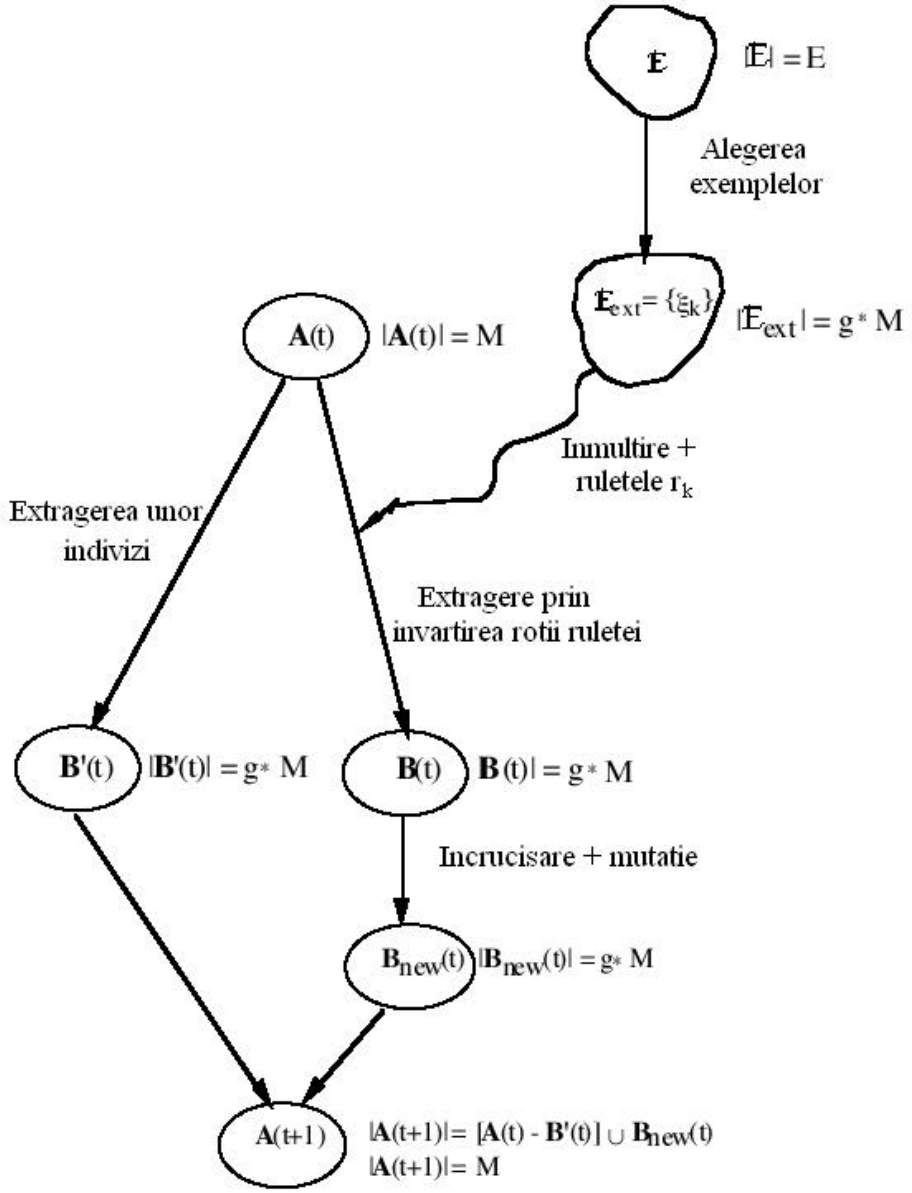


FIGURA 5 – Reprezentarea grafica a ciclului de baza al algoritmului

Numele de sufragiu universal deriva din faptul ca fiecare exemplu are aceeasi probabilitate de a fi extras. Mai precis, procesul extragerii exemplelor ξ_k din **E** la fiecare generatie, urmeaza o distributie binomiala cu probabilitatea de succes de 1/E si cu numarul de incercari egal cu g * M. Deoarece procesul de selectie este eacelasi la fiecare generatie, probabilitatea de distributie a numarului de selectii ale exemplului ξ_k in t generatii este tot una binomiala, cu g * M * t incercari si cu aceeasi probabilitate de succes de 1/E.

Considerand problema inversa a esantionarii lui Bernoulli, este usor de demonstrat ca pentru un exemplu generic ξ_k exista o probabilitate mai mare decat η de a fi extras in primele t generatii.

$$M \geq \frac{1}{gt} \frac{\lg\left(\frac{1}{\eta}\right)}{\lg\left(\frac{E}{E-1}\right)} \qquad (5.4)$$

Din relatia (5.4) deducem ca cel mai favorabil caz se obtine pentru g = 1. De fapt, cand intreaga populatie este innoita la fiecare generatie, un numar M mai mic este suficient pentru ca, dandu-se t, sa apara ξ_k.

Numarul mediu de generatii $\overline{t_{tot}}$ necesar pentru a extrage toate exemplele este cel mult:

$$\overline{t_{tot}} = \frac{E^2}{gM} \qquad (5.5)$$

Din nou, cea mai convenabila valoare este g= 1. Este clar faptul ca, in general, nu este necesar sa se execute toate cele E^2 incercari, deoarece orice formula, care nu concide cu descrierea unui singur exemplu, va acoperi mai multe concepte in acelasi timp.

Procedul de selectie bazat pe operatorul de sufragiu universal poate fi descris formal print-o matrice stochastica (E x M) **P**(t), ale carei linii corespund exemplelor ξ_k din **E**, si ale carei coloane corespund formulelor φ_j din **A**(t). Fiecare element $p_{kj}(t)$ al matricii este egal cu probabilitatea ca formula φ_j sa castige in ruleta corespunzatoare r_k, dandu-se un exemplu extras ξ_k:

$$\{\mathbf{P}_{kj}(t)\} = p_{kj}(t) = \frac{\nu_{kj} f_j x_j(t)}{\sum_{i=1}^{m} \nu_{kj} f_j x_j(t)} \qquad (1 \leq k \leq E,\ 1 \leq j \leq m) \qquad (5.6)$$

In relatia (4.6) f_j este valoarea fitness-ului formulei φ_j, iar ν_{kj} este functia caracteristica a multimii $COV(\varphi_j)$:

$$?_{kj} = \begin{cases} 1, & \text{daca } ?_k \in COV(\varphi_j) \\ 0, & \text{altfel} \end{cases} \qquad (1 \leq k \leq E,\ 1 \leq j \leq m) \qquad (5.7)$$

Elementele p_{kj} sunt normalizate relativ la linia k a matricii **P**.

6.4. Functia de fitness

In porblema invatarii conceptelor criteriile luate in cosiderare pentru a evalua solutiile au fost, in mod traditional, completitudinea, consistenta si simplicitatea. Daca primele doua concepte, completitudinea si consistenta, isi gasesc o definitie precisa in logica, cel al simplicitatii nu este univoc acceptat si gaseste interpretari diferite in contexte diferite. Cel mai frecvent folosit inteles al acestuia este de simlpicitate sintactica, care se reduce la numararea numarului de literali necesari pentru a reprezenta formula. Dezavantajul il reprezinta faptul ca definita depinde strict de limbajul de reprezentare. Mai mult, o aceeasi formula poate fi scrisa in forme echivalente avand valori diferite ale simplicitatii. In cazul algoritmului REGAL s-a adoptat o definitie computationala a simplicitatii, corespunzatoare numarului de teste care trebuie efectuate pentru a se verifica daca o formula este sataisfacuta de un exemplu. Acest lucru se face prin simpla verificare a faptului ca, pentru fiecare predicat din sablonul de limbaj, caracteristica asociata nu ia o valoare care nu apartine multimii de termeni din disjunctia interna corespunzatoare, mai exact se verifica daca este incalcata vreo conditie negata. Apoi, se obtine imediat o masura a complexitatii c(φ) pentru o formula φ prin numararea termenilor din sablon care nu apar in formula φ sau, echivalent, prin numararea zerourilor care apar sirul de biti s(φ). Pornind de la aceasta masura a complexitatii, s-a definit o masura a simplicitatii z(φ) ca fiind raportul:

$$z(\varphi) = \frac{lungimea(s(\varphi)) - c(\varphi)}{lungimea(s(\varphi))} \qquad (5.8)$$

Pentru consistenta s-a luat in considerare numarul de exemple negative acoperite de o formula φ:

$$w(\varphi) = n^{-}(\varphi) \qquad (5.9)$$

Deoarece operatorul de selectie trateaza completitudinea prin mecanismul invartirii rotii ruletei, doar consistenta si simplicitatea sunt luate in considerare in functia de fitness. Mai precis, se foloseste functia:

$$f(\varphi) = f(z, w) = (1 + Az)\, e^{-w} \qquad (5.10)$$

In relatia (5.10), parametrul A este o valoare ajustabila de utilizator, valoarea sa curenta fiind A = 0.1. Se observa ca $\mathbf{f_{Max}} = 1.1$ si ca $\lim_{w \to \infty} f(z, w) = 0$. Un grafic al functie f se afla in Figura 6.

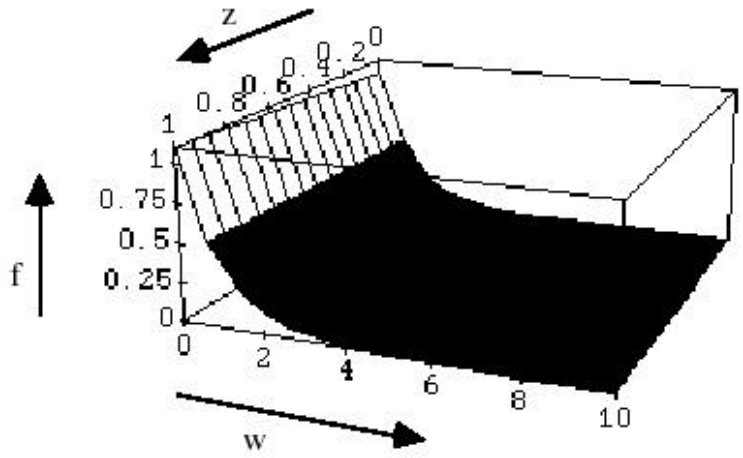


FIGURA 6 – reprezentarea grafica a functiei de fitness f(φ) in raport cu simplicitatea z si cu consistenta w a unei formule φ. Valori mari ale lui z (w) corespund unor descresteri ale simplicitatii (consistentei).

Expresia (5.10) deriva din experimente cu diverse functii de fitness. S-a dovedit a oferi evaluari rezonabile, dar nu se poate spune ca se vrea a fi o alerege unica sau cea mai buna.

Cu privire la complexitatea operatorului de selectie, pentru a calcula valorile probabilitatilor din relatia (5.6) trebuie evaluat, pentru fiecare φ_j, un numar de sume cel mult egal cu numarul de rulete disponibile: deoarece exista cel mult o ruleta pentru fiecare exemplu din **E**, atunci numarul de pasi executati este cel mult $E \cdot m \leq E \cdot M$. De aceea, complexitatea clobala este de ordinul **O**(E·M). Aceasta complexitate este in contrast cu cea care se obtine folosind functii partajate, adica **O**(M^2·E). Aceasta ultima valoare s-a obtinut pentru folosirea unei distante fenotipice la evaluarea acoperirii exemplelor, pentru a faca eo comparatie corecta cu operatorul de selectie "sufragiul universal".

In final, trebuie evidentiat un fapt important in ceea ce priveste evaluarea fitness-ului si probabilitatea compararii: in REGAL nu exista notiunea de valoare medie globala a fitness-ului; de fapt, probabilitatile din (5.6) sunt normalizate

doar relativ la valoarea medie a fitness-ului indivizilor ce apartin aceleiasi rulete. Indivizii care nu apar in aceeasi ruleta nu sunt comparabili. Argumentam ca exact aceasta proprietate de "localitate" a fitness-ului este cea care determina abilitatea algoritmului REGAL de a mentine specii diferite in echilibru.

In [Giordana, Neri si Saitta, 1994] a fost propusa o metoda generala de investigare a proprietatilor macroscopice ale populatiilor in evolutie in conformitate cu o matrice de tranzitie a probabilitatilor. Aceasta metoda, bazata pe definirea unei "*populatii medii virtuale*", permite depasirea unor dificultati tehnice implicate in acst tip de analiza, oferind totusi in acelasi timp estimari precise ale parametrilor ce controleaza evolutia. Metoda este foarte potrivita pentru o varietate de abordari si in [Neri si Saitta, 1995] se pot gasi aplicatiile sale in mai multe metode de cautare.

7. Teoria formarii speciilor si a niselor

Formarea speciilor pare in mod special interesanta pentru invatarea de concepte. In primul rind ofera posibilitatea invatarii simultane atat a mai multor disjuncti cat si a mai multor concepte. Mai mult, resursele de calcul sunt exploatate mai eficient prin evitarea redundantelor si a replicarii nefolositoare si prin exploatarea naturala a facilitatilor calculului distribuit. Metode propuse pentru formarea niselor si a speciilor includ "Crowding" [De Jong, 1975] si "Sharing Functions" [Goldberg & Richardson, 1987].

Crowding este o varianta a unui algoritm genetic simplu care inlocuieste indivizii vechi. Noii indivizi, generati prin incrucisare si mutatii, ii inlocuiesc pe cei vechi care sunt cei mai similari cu ei, corespunzator unei masuri date a similitudinii. In acest fel este posibil ca subpopulatiile sa creasca deoarece presiunea genetica tinde sa se manifeste in primul rand printre indivizii similari.

Metoda bazata pe functii partajate modifica probabilitatea de selectie cu scopul de a inhiba cresterea excesiva a presiunii genetice a unie subpopulatii. Acest lucru este obtinut prin reducerea valorii fitness-ului unui individ dupa numarul de indivizi asemanatori lui. In formularea initiala s-a luat in considerare partajarea genotipica. Valoarea fitness-ului f(φ), ascoiat unui individ φ, a fost considerata ca o recompensa a mediului pentru a fi impartita cu alti indivizi. Similaritatea dintre doi indivizi, φ si φ', a fost definita ca fiind distanta Hamming dintre sirurile de biti asociate lor. De asemenea, poate fi folosita si partajarea fenotipica, considerand distanta intre doi indivizi in domeniul lor semantic. Prin aplicarea atat a crowding-ului cat si a functiilor partajate asupra unui set de test

de probleme de optimizare, s-a observat ca crowding-ul a fost mai putin efectiv, deoarece nu poate gasi tot timpul fitness-ul maxim. Metoda s-a dovedit a functiona bine pentru probleme de invatare a mai multor concepte unimodale la un moment dat, dar nu a fost capabila sa permita o formare stabila a subpopulatiilor, reprezentative definitiilor disjuncte ale aceluiasi concept. In toate experimentele efectuate, pe termen lung, un singur disjunct a iesit ca fiind "cel ma bun".

Partajarea fenotipica sau genotipica functioneaza bine in spatii vectoriale, unde notiunea de distanta are o semantica clara. Deb si Goldberg au gasit o diferenta de performanta intre distanta fenotipica si cea genotipica, in favoarea celei fenotipice: folosind ca distanta genotipica definitia lui Hamming, algoritmul genetic a avut o comportare intermediara intre crowding si partajare fenotipica, avand abilitatea de a gasi toate maximele existente.

8. Concluzii

In aceasta lucrare s-a prezentat o metoda de abordare baza pe un algoritm co-evolutiv pentru rezolvarea problemelor si in particular pentru problema acoperirii optime a sirurilor [Forrest et al., 1993]. Credem ca am reusit sa aratam in aceasta lucrare faptul ca sistemul propus este capabil sa lucreze in domenii complexe si se poate descurca cu seturi mari de date, putand concura cu sisteme asemanatoare. Oricum, comportamentul sistemului si potentialul acestuia poate ridica mai multe intrebari decat raspunsuri.

Un aspect important este rolul pe care algoritmii genetici il pot juca in problemele de invatare automata (machine learning). Suntem de acord cu faptul ca problema invatarii trebuie sa exploateze cat mai mult posibil cunostintele de baza specifice domeniului, incercand sa limiteze astfel cautarea in spatiul ipotezelor prin constrangeri apriorice. De fapt, in acest caz algoritmii de cautare "simbolica" pot fi mult mai convenabili, atat din cauza ca ei pot exploata chiar acea informatie apriorica, cat si din cauza ca au nevoie de resurse computationale mai reduse atunci cand efectueaza cautari in spatii restranse. Pe de alta parte, atunci cand sunt disponibile putine cunostinte, sau cand acestea nu se conformeaza constrangerilor, spatiul de cautare ramane vast. Totusi, credem ca algoritmii genetic au potentialul de a deveni abordarea castigatoare a problemei, datorita puterii de explorare pe care o ofera fara a cere reducerea limbajului de reprezentare a ipotezelor.

Bibliografie

Hugues Juille, Jordan B. Pollack(1995)
Semantic Niching and Coevolution in Optimization Problems

Mitchell A. Potter, Kenneth A. De Jong (2000)
Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents

S. A. Kauffman and S. Johnsen (1991)
Co-evolution to the edge of chaos: Coupled fitness landscapes, poised states, and co-evolutionary avalanches

M. A. Potter, K. A. De Jong and J. J. Grefenstette (1995)
A coevolutionary approach to learning sequential decision rules

Eric V. Siegel (1999)
Competitively Evolving Decision Trees Against Fixed Training Cases for Natural Language Processing

S. Forrest, B. Javornik, R.E. Smith and S.A. Perelson (1993)
Using genetic algorithms to explore pattern recognition in the immune system

Mitchell A. Potter and K. A. De Jong(1994)
A cooperative coevolutionary approach to function optimization

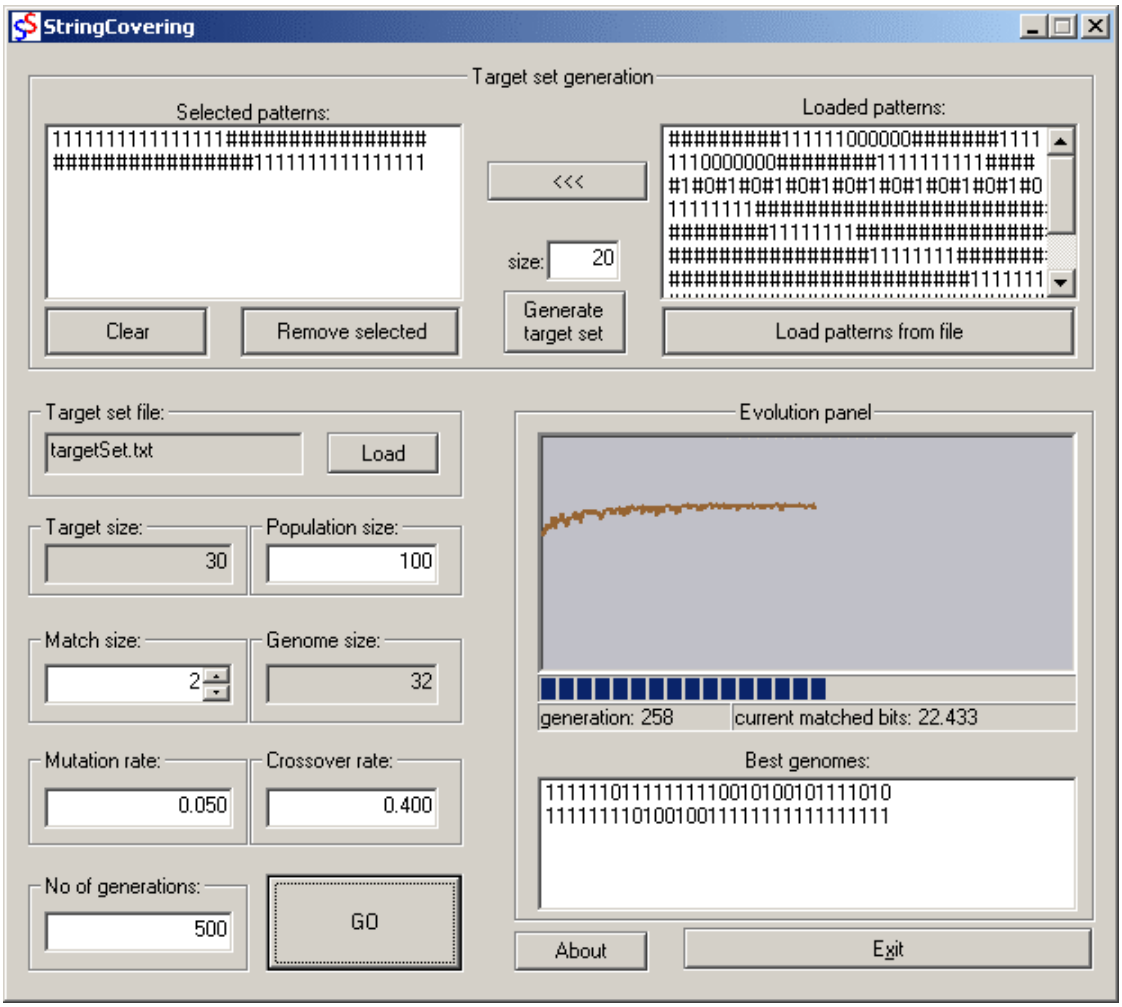
H. P. Schwefel (1995)
Evolution and Optimum Seeking

J. P. Rosca and D. H. Ballard (1994)
Hierarchical self-organization in genetic programming

R. E. Smith, S. Forrest and S. Perelson (1993)
Searching for diverse, cooperative populations with genetic algorithms

Anexe

A. Imagini ale aplicatiei



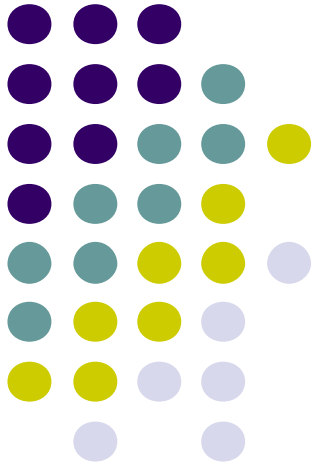
B. Prezentarea aplicatiei (slide-show)

Algoritmi genetici pentru rezolvarea problemelor prin evolutie si co-evolutie

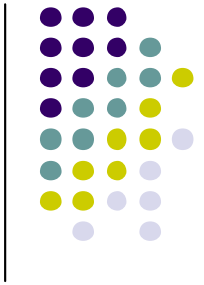
Coordonator proiect:
Prof. dr. ing. **Adina Magda Florea**

Absolvent:
Sorin OSTAFIEV

UNIVERSITATEA POLITEHNICA BUCURESTI
FACULTATEA DE AUTOMATICA SI CALCULATOARE
2003

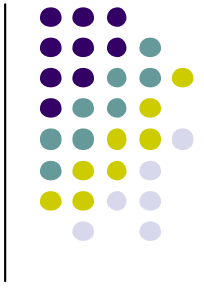


Scopul si motivatia



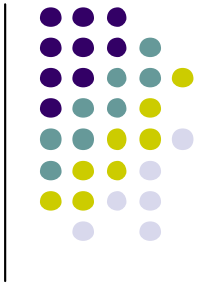
- Utilizarea algoritmilor evolutivi pentru rezolvarea problemelor din ce in ce mai complexe necesita oportunitati pentru ca sub-problemele rezolvate individual sa interactioneze
- Este prezentata o arhitectura pentru co-evolutia unor astfel de componente ca o colectie de specii cooperante

De ce algoritmii evolutivi clasici nu sunt suficient de buni?



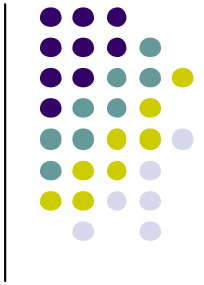
- Populatiile de indivizi au o puternica convergenta
  - se inlatura persistenta pe termen lung a sub-componentelor diverse
- Indivizii reprezinta solutii complete evaluate individual, in izolare
  - nu exista ocazii pentru aparitia co-adaptarii
- Cum ar trebui sa fie sub-componentele? Cate ar trebui sa fie?

Pentru extinderea modelului clasic trebuie sa avem in vedere urmatoarele aspecte



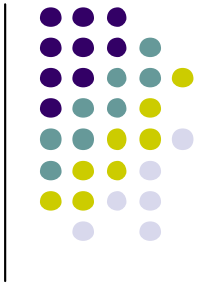
- Descompunerea problemei
  - de la inceput sau ca urmare a evolutiei algoritmului
- Dependentele dintre sub-componente
  - co-adaptarea poate aparea doar daca interactiunile sunt modelate
- Asocierea rolurilor
  - Determinarea contributiei fiecarei sub-componente la intreaga solutie
- Mentinerea diversitatii
  - Diversitatea trebuie pastrata mai mult decat in cazult algoritmilor evolutivi clasici

Functia de fitness



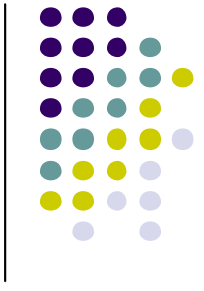
- Fiecare specie evolueaza in propria populatie si se adapteaza mediului prin intermediul algoritmului evolutiv
- Indivizii dintr-o specie sunt evaluati intre ei prin colaborarea cu indivizii “reprezentativi” din alte specii

Exemple



- Maximizarea unei functii *f(x)* cu *n* variabile independente (*n* specii, cate una pentru fiecare variabila).
  - indivizii sunt selectati in functie de cat de bine maximizeaza functia
- Dezvoltarea unui sistem de control bazat pe reguli pentru simularea unui robot autonom
  - fiecare populatie reprezinta un set de reguli pentru diferite comportamente
  - indivizii sunt selectati in functie de cat de bine completeaza seturile de reguli ale altor specii

Acoperirea sirurilor



- Gasirea unui set potrivit de *N* vectori binari care se potrivesc (cat mai bine cu putinta) pe un set sursa de *K* vectori binari (*K* >> *N*)
- In acest context, problema descompunerii consta in:
  - determinarea dimensiunii setului de potrivit
  - determinare acoperirii fiecarui element din set
- Calcularea fitness-ului unui set potrivit M se face folosind formula:

$$S(M) = \frac{1}{K} \sum_{i=1}^{K} \max(S(\vec{m}_1, \vec{t}_i), \ldots, S(\vec{m}_N, \vec{t}_i))$$

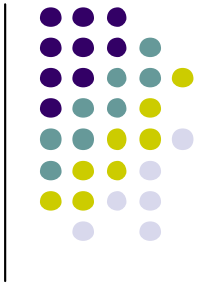
Acoperirea sirurilor - Exemplu

Fie setul sursa format din trei siruri de lungime 32

```
11111111111111111111111111111111
11111111110000000000000000000000
00000000000000000000001111111111
```

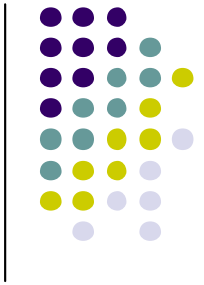
- Acoperirea optima folosind un singur sir:
  `11111111110000000000001111111111`
- Una din acoperiri cu 2 siruri:
  `11111111111111111111111111111111`
  `10010110110000000000001111110101`
- Acoperirea cu 3 siruri este formata exact din sirurile setului sursa

Algoritmul de baza



gen = 0
for each species *s* **do begin**
    $Pop_s(gen)$ = initializeaza populatia cu valori aleatorii
    evalueaza fitness-ul fiecarui individ $Pop_s(gen)$
end
while conditia de terminare = false **do begin**
    gen = *gen* +1
    for each species *s* **do begin**
        selecteaza $Pop_s\ (gen)$ din $Pop_s\ (gen\text{-}1)$ pe baza fitness-ului
        aplica operatorii genetici pe $Pop_s\ (gen)$
        evalueaza fitness-ul fiecarui individ $Pop_s\ (gen)$
    end
end

Concluzii



- Arhitectura cooperativa co-evolutiva poate fi privita ca o extensie generala a oricarei paradigme evolutive si nu doar a algoritmilor genetici
  - de exemplu: software de control al robotilor in medii multi-agent
- Furnizeaza stimuli pentru evolutia spre un anumit numar interdependent de sub-componente care acopera nise multiple
- Evolueaza spre un nivel de generalitate potrivit
- Deoarece domeniile devin tot mai complexe, evolutia speciilor ar putea fi condusa de mai mult decat fitness-ul sistemului pentru a putea produce o descompunere optima

C. Listingul codului sursa

```
1  ////////////////////////////////////////////////
2  // (c) 2003 Sorin OSTAFIEV (sorin@ostafiev.com) //
3  ////////////////////////////////////////////////
4
5  // GA.h: interface for the GA class.
6  //
7  //////////////////////////////////////////////////////////////////////
8
9  #if !defined(AFX_GA_H__7B677626_0DC7_4B47_9E97_734A880F694C__INCLUDED_)
10 #define AFX_GA_H__7B677626_0DC7_4B47_9E97_734A880F694C__INCLUDED_
11
12 #include "globals.h"    // Added by ClassView
13 #if _MSC_VER > 1000
14 #pragma once
15 #endif // _MSC_VER > 1000
16
17
18 #include <vector>
19 using namespace std;
20
21 class GA
22 {
23
24
25 public:
26     double getAverageMatchedBits();
27     void step();
28     friend class Population;
29     friend class Genome;
30
31
32     GA(const int npops, const int pop_size, const double crossoverrate, const double mutationrate);
33     virtual ~GA();
34
35     vector <Population> _populations;
36
37 private:
38
39     GA(const GA& ga);
40     GA& operator=(const GA &ga);
41
42     double _cross_over_rate;
43     double _mutation_rate;
44     int _number_of_populations;
45 };
46
47
48 #endif // !defined(AFX_GA_H__7B677626_0DC7_4B47_9E97_734A880F694C__INCLUDED_)
49
```

```cpp
////////////////////////////////////////////////
// (c) 2003 Sorin OSTAFIEV (sorin@ostafiev.com) //
////////////////////////////////////////////////

// GA.cpp: implementation of the GA class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "StringCovering.h"
#include "GA.h"


#include "Population.h"


#include <cassert>
using namespace std;

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

GA::GA(const int npops, const int pop_size, const double crossoverrate, const double mutationrate):
    _cross_over_rate(crossoverrate),
    _mutation_rate(mutationrate),
    _number_of_populations(npops)

{
    assert ((0.0f <= _mutation_rate) && (_mutation_rate <= 1.0f));
    assert ((0.0f <= _cross_over_rate) && (_cross_over_rate<= 1.0f));

    _populations.reserve(_number_of_populations);
    for (int i = 0; i < _number_of_populations; i++)
    {
        _populations.push_back(Population(this, pop_size, Genome::GENOME_RANDOM));
    };

}

GA::~GA()
{

}



void GA::step()
{
    int i = 0;
    for (i = 0; i < _populations.size(); i++)
    {

        _populations.at(i).pre_step();

    };


    for (i = 0; i < _populations.size(); i++)
    {

        _populations.at(i).step();

    };


    for (i = 0; i < _populations.size(); i++)
    {

        _populations.at(i).post_step();

    };

}


double GA::getAverageMatchedBits()
```

```
84  {
85      const int target_set_size = targetSet->_genomes.size();
86
87      assert (_number_of_populations == _populations.size());
88
89      double overall_fitness = 0;
90
91      for (GenomesVector::const_iterator i = targetSet->_genomes.begin(); i != targetSet->_genomes.end(); i++)
92      {
93          double max = 0;
94
95          for (int k = 0; k < _number_of_populations; k++)
96          {
97              assert(0 < _populations.at(k)._genomes.size());
98              const Genome& best = _populations.at(k)._genomes.at(0);
99
100             const double current_fitness = best.getGenomeFitness(*i);
101
102             if (max < current_fitness)
103             {
104                 max = current_fitness;
105             };
106
107         };
108
109         overall_fitness += max;
110
111     };
112
113     return (overall_fitness / target_set_size) * Genome::_genome_size;
114 }
115
```

```
1  ////////////////////////////////////////////////
2  // (c) 2003 Sorin OSTAFIEV (sorin@ostafiev.com) //
3  ////////////////////////////////////////////////
4
5  // Population.h: interface for the Population class.
6  //
7  //////////////////////////////////////////////////////////////////////
8
9  #if !defined(AFX_POPULATION_H__F09111B1_0480_45B1_A376_A230915BEF67__INCLUDED_)
10 #define AFX_POPULATION_H__F09111B1_0480_45B1_A376_A230915BEF67__INCLUDED_
11
12 #if _MSC_VER > 1000
13 #pragma once
14 #endif // _MSC_VER > 1000
15
16 #include "GA.h"
17 #include "Genome.h"
18 #include <vector>
19 using namespace std;
20
21 #include "globals.h"
22
23 class Population
24 {
25
26 public:
27     friend class Genome;
28     friend class GA;
29
30     const int getPopulationSize() const;
31     const GenomesVector::const_iterator getBestIndividual() const;
32
33     virtual ~Population();
34
35     Population(const GA* const ga, const int pop_size, Genome::genome_type type);
36     Population(const Population& population);
37     const Population& operator=(const Population &population);
38     GenomesVector _genomes;
39
40 private:
41     Population* _next_generation;
42     void pre_step();
43     void step();
44     void post_step();
45
46     int _population_size;
47
48     const GA* const _ga;
49     mutable GenomesVector::const_iterator _best_genome;
50
51 };
52
53
54 #endif // !defined(AFX_POPULATION_H__F09111B1_0480_45B1_A376_A230915BEF67__INCLUDED_)
55
```

```cpp
////////////////////////////////////////////////
// (c) 2003 Sorin OSTAFIEV (sorin@ostafiev.com) //
////////////////////////////////////////////////

// Population.cpp: implementation of the Population class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "StringCovering.h"
#include "Population.h"


#include <cassert>
#include <algorithm>
#include <functional>
using namespace std;


#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////


Population::Population(const GA* const ga, const int pop_size, Genome::genome_type type):
    _best_genome(NULL),
    _ga(ga),
    _population_size(pop_size),
    _next_generation(NULL)

{

        _genomes.reserve(_population_size);
        for (int i = 0; i < _population_size; i++)
        {
            _genomes.push_back(Genome(this, ga, type));
        };

}

Population::~Population()
{

}

const GenomesVector::const_iterator Population::getBestIndividual() const
{
    if (NULL == _best_genome)
    {
        double max_fitness = -1;
        for (GenomesVector::const_iterator i = _genomes.begin(); i != _genomes.end(); i++)
        {

            const double current_fitness = i->getPopulationFitness();
            assert ((0.0f <= current_fitness) && (current_fitness <= 1.0f));

            if (max_fitness < i->getPopulationFitness())
            {
                max_fitness = i->getPopulationFitness();
                _best_genome = (const GenomesVector::const_iterator) i;
            };

        };

        assert ((0.0f <= max_fitness) && (max_fitness <= 1.0f));

    };
    return _best_genome;
}

const int Population::getPopulationSize() const
{
    assert (0 < _genomes.size());
    return _genomes.size();
}


```

```
84  Population::Population(const Population& population):
85      _ga(population._ga)
86  {
87      operator=(population);
88  };
89
90
91  const Population& Population::operator=(const Population& population)
92  {
93      if (this != &population)
94      {
95          _best_genome = population._best_genome;
96          const_cast <const GA*> (_ga) = population._ga;
97          _genomes = population._genomes;
98          _population_size = population._population_size;
99          _next_generation = population._next_generation;
100         assert (_ga == population._ga);
101
102         for (int i = 0; i < population._genomes.size(); i++)
103         {
104             const_cast <const Population*> (_genomes.at(i)._population) = this;
105
106         };
107     };
108
109
110
111     return (*this);
112 };
113
114
115
116 void Population::pre_step()
117 {
118     assert (NULL == _next_generation);
119     _next_generation = new Population(NULL, _population_size, Genome::GENOME_ZERO);
120     assert (NULL != _next_generation);
121
122     assert(_genomes.size() == _population_size);
123
124     typedef pair <double, int> DOUBLE2INT;
125
126     vector <DOUBLE2INT> d2i;
127
128     d2i.reserve(_genomes.size());
129     for (int i = 0; i < _genomes.size(); i++)
130     {
131         d2i.push_back(make_pair(_genomes.at(i).getGAFitness(), i));
132     };
133
134     vector<DOUBLE2INT>::iterator i1 = d2i.begin();
135     vector<DOUBLE2INT>::iterator i2 = d2i.end();
136
137
138     sort (i1, i2, greater<DOUBLE2INT>());
139
140
141     double max_sum = 0;
142     for (i = 0; i < _genomes.size(); i++)
143     {
144         max_sum += d2i.at(i).first;
145     };
146
147
148     int k = 0;
149     double sum = 0;
150     for (i = 0; (i < _population_size) && (k < _population_size); i++)
151     {
152         sum += d2i.at(i).first;
153
154
155         const int delta = int(1 + double(sum * _population_size) / max_sum) - k;
156         assert (0 <= delta);
157
158         for (int j = k; j < k + delta; j++)
159         {
160             if (j < _population_size)
161             {
162                 _next_generation->_genomes.at(j) = _genomes.at(d2i.at(i).second);
163
164             };
165
166         };
```

```
167
168         k += delta;
169     };
170
171     assert (_population_size <= k);
172
173 }
174
175 void Population::step()
176 {
177     vector <bool> b(_population_size, false);
178
179     const double cross_over_rate = _ga->_cross_over_rate;
180     const int mutation_factor = int(100 * _ga->_mutation_rate);
181     assert (0 < mutation_factor);
182
183     const int cross_overs = int((_population_size * cross_over_rate) / 2);
184
185     for(int i = 0; i < cross_overs; i++)
186     {
187         const int i1 = rand() % _population_size;
188         const int i2 = rand() % _population_size;
189
190         _next_generation->_genomes.at(i1).Crossover(_next_generation->_genomes.at(i2));
191
192     };
193
194
195     for (i = 0; i < _population_size; i++)
196     {
197         if ((rand() % 100) < mutation_factor)
198         {
199             _next_generation->_genomes.at(i).Mutate();
200         };
201     };
202
203
204 };
205
206
207 void Population::post_step()
208 {
209
210     for (int i = 0; i < _genomes.size(); i++)
211     {
212         _genomes.at(i) = _next_generation->_genomes.at(i);
213         _genomes.at(i)._gaFitness = -1;
214         _genomes.at(i)._populationFitness = -1;
215
216     };
217
218     _best_genome = NULL;
219
220     // clean up
221     assert (NULL != _next_generation);
222     delete _next_generation;
223     _next_generation = NULL;
224 }
225
226
```

```
1  ////////////////////////////////////////////////
2  // (c) 2003 Sorin OSTAFIEV (sorin@ostafiev.com) //
3  ////////////////////////////////////////////////
4
5  // Genome.h: interface for the Genome class.
6  //
7  //////////////////////////////////////////////////////////////////////
8
9  #if !defined(AFX_GENOME_H__1B30186E_485F_434A_B615_BE44BC598C7A__INCLUDED_)
10 #define AFX_GENOME_H__1B30186E_485F_434A_B615_BE44BC598C7A__INCLUDED_
11
12 #if _MSC_VER > 1000
13 #pragma once
14 #endif // _MSC_VER > 1000
15
16 class Population;
17 class GA;
18 #include "globals.h"
19 #include <bitset>
20
21 using namespace std;
22
23 class Genome
24 {
25     static int _genome_size;
26
27 public:
28     void setFromPattern(const string& pattern);
29
30     friend class Population;
31     friend class GA;
32
33
34     enum genome_type
35     {
36         GENOME_RANDOM,
37         GENOME_ZERO
38     };
39
40     const double getGAFitness() const;
41     const double getPopulationFitness() const;
42     void Crossover(Genome& genome);
43     void Mutate();
44     Genome(const Population* const population, const GA* const ga, const genome_type type = GENOME_ZERO);
45     virtual ~Genome();
46
47
48     Genome(const Genome& genome);
49     Genome& operator=(const Genome &genome);
50
51     bitset <MAX_GENOME_LENGHT> _body;
52
53 private:
54
55     const double getGenomeFitness(const Genome& genome) const;  // genome over genome
56     const Population* const _population;                        // which population we belong
57     const GA* const _ga;                                        // ...and which ga
58
59     mutable double _populationFitness;      // fitness within population
60     mutable double _gaFitness;              // fitness within ga
61
62 };
63
64
65 #endif // !defined(AFX_GENOME_H__1B30186E_485F_434A_B615_BE44BC598C7A__INCLUDED_)
66
```

```cpp
////////////////////////////////////////////////
// (c) 2003 Sorin OSTAFIEV (sorin@ostafiev.com) //
////////////////////////////////////////////////

// Genome.cpp: implementation of the Genome class.
//
//////////////////////////////////////////////////////////////////////


#include "stdafx.h"
#include "StringCovering.h"
#include "Genome.h"

#include "globals.h"
#include <cassert>
#include "Population.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif



//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

Genome::Genome(const Population* const population, const GA* const ga, const genome_type type /* = GENOME_ZERO */)
    _population(population),
    _ga(ga),
    _populationFitness(-1),
    _gaFitness(-1),
    _body(_genome_size)
{
    switch (type)
    {

        case GENOME_RANDOM:
            {

                for (int i = 0; i < _genome_size; i++)
                {
                    if (0 == (rand() & 1))
                    {
                        _body.reset(i);
                    }
                    else
                    {
                        _body.set(i);
                    };
                };
            };
            break;

        case GENOME_ZERO:
            {
                _body.reset();
            };
            break;

        default:
            assert(0);
            break;
    };
}

Genome::~Genome()
{

}

void Genome::Mutate()
{
    assert (_genome_size == _body.size());

    const int mutation_bit = rand() % _genome_size;
    _body.flip(mutation_bit);

}

void Genome::Crossover(Genome &genome)
```

```
84  {
85      if (this != &genome) //cross with someone, but not with me...
86      {
87          assert (_genome_size == _body.size());
88          const int half_genome_size = _genome_size / 2;
89          for (int i = 0; i < half_genome_size; i++)
90          {
91              const bool b = _body.at(i);
92              _body.at(i) = genome._body.at(i);
93              genome._body.at(i) = b;
94
95          };
96
97      };
98
99  }
100
101
102 // genome over population
103 const double Genome::getPopulationFitness() const
104 {
105
106     if (_populationFitness < 0)
107     {
108         double population_fitness = 0;
109
110         for (GenomesVector::const_iterator i = targetSet->_genomes.begin(); i != targetSet->_genomes.end(); i++)
111         {
112             population_fitness += getGenomeFitness(*i);
113
114         };
115         population_fitness /= targetSet->getPopulationSize();
116
117         _populationFitness = population_fitness;
118         assert ((0.0f <= _populationFitness) && (_populationFitness <= 1.0f));
119     };
120
121     return _populationFitness;
122 }
123
124 // genome over genome;
125 const double Genome::getGenomeFitness(const Genome &genome) const
126 {
127     double genome_fitness = 0;
128     for (int i = 0; i < _genome_size; i++)
129     {
130         if (genome._body.at(i) == _body.at(i))
131         {
132             genome_fitness++;
133         };
134     };
135
136     return genome_fitness / _genome_size;
137 }
138
139
140 const double Genome::getGAFitness() const
141 {
142     if (NULL == _ga)
143     {
144         return -1;
145     };
146
147     if (_gaFitness < 0)
148     {
149         double ga_fitness = 0;
150
151
152         for (GenomesVector::const_iterator i = targetSet->_genomes.begin(); i != targetSet->_genomes.end(); i++)
153         {
154
155             double max_fitness = -1;
156             for (PopulationsVector::const_iterator k = _ga->_populations.begin(); k != _ga->_populations.end(); k+
157             {
158                 double current_fitness = -1;
159
160
161                 if (_population != k)
162                 {
163                     current_fitness = i->getGenomeFitness(*k->getBestIndividual());
164                 }
165                 else
166                 {
```

```
                    current_fitness = i->getGenomeFitness(*this);
                };
                assert ((0.0f <= current_fitness) && (current_fitness <= 1.0f));



                if (max_fitness < current_fitness)
                {
                    max_fitness = current_fitness;
                };


            };

            assert ((0.0f <= max_fitness) && (max_fitness <= 1.0f));

            ga_fitness += max_fitness;

        };

        ga_fitness /= targetSet->_genomes.size();

        _gaFitness = ga_fitness;
        assert ((0.0f <= _gaFitness) && (_gaFitness <= 1.0f));
    };

    return _gaFitness;
}


Genome::Genome(const Genome& genome):
    _ga(genome._ga),
    _population(genome._population)
{
    operator=(genome);
};


Genome& Genome::operator=(const Genome &genome)

{
    if (this != &genome)
    {
        _body = genome._body;
        const_cast <const GA*> (_ga) = genome._ga;
        _gaFitness = genome._gaFitness;
        const_cast <const Population*> (_population) = genome._population;
        _populationFitness = genome._populationFitness;


        assert (_ga == genome._ga);
        assert (_populationFitness == genome._populationFitness);
    };

    return (*this);
};


void Genome::setFromPattern(const string &pattern)
{
    assert (_genome_size == _body.size());
    assert (_genome_size == pattern.size());

    for (int i = 0; i < _genome_size; i++)
    {
        switch (pattern.at(_genome_size - 1 - i))
        {
        case '0':
            _body.at(i) = false;
            break;

        case '1':
            _body.at(i) = true;
            break;

        default:
            if (0 == rand() % 2)
            {
                _body.at(i) = false;
            }
            else
            {
                _body.at(i) = true;
```

```
250                 };
251                 break;
252 
253             };
254 
255         };
256 
257     }
258 
```

```
1  ////////////////////////////////////////////////
2  // (c) 2003 Sorin OSTAFIEV (sorin@ostafiev.com) //
3  ////////////////////////////////////////////////
4
5  #ifndef __global__h__
6  #define __global__h__
7
8  #pragma warning(disable:4786)   //stl issue - decoration too long
9
10 #include <vector>
11
12 using namespace std;
13
14 class Genome;
15 class Population;
16
17 typedef vector<Genome> GenomesVector;
18 typedef vector<Population> PopulationsVector;
19
20 const MAX_GENOME_LENGHT = 32;
21
22
23 extern Population* targetSet;
24
25 #endif
```

```
1  ////////////////////////////////////////////////
2  // (c) 2003 Sorin OSTAFIEV (sorin@ostafiev.com) //
3  ////////////////////////////////////////////////
4
5  // StringCovering.h : main header file for the STRINGCOVERING application
6  //
7
8  #if !defined(AFX_STRINGCOVERING_H__F93D04E3_5CC0_408C_A598_27A29EF3CA33__INCLUDED_)
9  #define AFX_STRINGCOVERING_H__F93D04E3_5CC0_408C_A598_27A29EF3CA33__INCLUDED_
10
11 #if _MSC_VER > 1000
12 #pragma once
13 #endif // _MSC_VER > 1000
14
15 #ifndef __AFXWIN_H__
16     #error include 'stdafx.h' before including this file for PCH
17 #endif
18
19 #include "resource.h"       // main symbols
20
21 /////////////////////////////////////////////////////////////////////////////
22 // CStringCoveringApp:
23 // See StringCovering.cpp for the implementation of this class
24 //
25
26 class CStringCoveringApp : public CWinApp
27 {
28 public:
29     CStringCoveringApp();
30
31 // Overrides
32     // ClassWizard generated virtual function overrides
33     //{{AFX_VIRTUAL(CStringCoveringApp)
34     public:
35     virtual BOOL InitInstance();
36     //}}AFX_VIRTUAL
37
38 // Implementation
39
40     //{{AFX_MSG(CStringCoveringApp)
41         // NOTE - the ClassWizard will add and remove member functions here.
42         //    DO NOT EDIT what you see in these blocks of generated code !
43     //}}AFX_MSG
44     DECLARE_MESSAGE_MAP()
45 };
46
47
48 /////////////////////////////////////////////////////////////////////////////
49
50 //{{AFX_INSERT_LOCATION}}
51 // Microsoft Visual C++ will insert additional declarations immediately before the previous line.
52
53 #endif // !defined(AFX_STRINGCOVERING_H__F93D04E3_5CC0_408C_A598_27A29EF3CA33__INCLUDED_)
54
```

```cpp
////////////////////////////////////////////////
// (c) 2003 Sorin OSTAFIEV (sorin@ostafiev.com) //
////////////////////////////////////////////////

// StringCovering.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "StringCovering.h"
#include "StringCoveringDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CStringCoveringApp

BEGIN_MESSAGE_MAP(CStringCoveringApp, CWinApp)
    //{{AFX_MSG_MAP(CStringCoveringApp)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG
    ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CStringCoveringApp construction

CStringCoveringApp::CStringCoveringApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CStringCoveringApp object

CStringCoveringApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CStringCoveringApp initialization

BOOL CStringCoveringApp::InitInstance()
{
    AfxEnableControlContainer();

    // Standard initialization
    // If you are not using these features and wish to reduce the size
    //  of your final executable, you should remove from the following
    //  the specific initialization routines you do not need.

#ifdef _AFXDLL
    Enable3dControls();         // Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic();   // Call this when linking to MFC statically
#endif

    CStringCoveringDlg dlg;
    m_pMainWnd = &dlg;
    int nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
        // TODO: Place code here to handle when the dialog is
        //  dismissed with OK
    }
    else if (nResponse == IDCANCEL)
    {
        // TODO: Place code here to handle when the dialog is
        //  dismissed with Cancel
    }

    // Since the dialog has been closed, return FALSE so that we exit the
    //  application, rather than start the application's message pump.
    return FALSE;
}

```

```
1  ////////////////////////////////////////////////
2  // (c) 2003 Sorin OSTAFIEV (sorin@ostafiev.com) //
3  ////////////////////////////////////////////////
4
5  // StringCoveringDlg.h : header file
6  //
7
8  #if !defined(AFX_STRINGCOVERINGDLG_H__C7A2133D_47D6_42CB_B33B_711F041DF8DF__INCLUDED_)
9  #define AFX_STRINGCOVERINGDLG_H__C7A2133D_47D6_42CB_B33B_711F041DF8DF__INCLUDED_
10
11 #if _MSC_VER > 1000
12 #pragma once
13 #endif // _MSC_VER > 1000
14
15 /////////////////////////////////////////////////////////////////////////////
16 // CStringCoveringDlg dialog
17
18 #include "MyStatic.h"
19
20 class CStringCoveringDlg : public CDialog
21 {
22 // Construction
23     friend unsigned int DoEvolution(void *data);
24
25 public:
26     void EnableInterface();
27     void DisableInterface();
28     void UpdateEvolutionPanel();
29     void UpdateDialogFromValues();
30     bool UpdateValuesFromDialog();
31     bool UpdateValuesFromDialog1();
32     CStringCoveringDlg(CWnd* pParent = NULL);   // standard constructor
33
34 // Dialog Data
35     //{{AFX_DATA(CStringCoveringDlg)
36     enum { IDD = IDD_STRINGCOVERING_DIALOG };
37     CButton m_ok;
38     CButton m_load_target;
39     CButton m_load_patterns;
40     CButton m_select_pattern;
41     CButton m_generate;
42     CButton m_remove_selected;
43     CButton m_clear_patterns;
44     CButton m_go;
45     CMyStatic   m_evolution;
46     CListBox    m_bests;
47     CProgressCtrl   m_progress;
48     CStatic m_gen_no;
49     CStatic m_matched_bits;
50     CEdit   m_target_set_size;
51     CListBox    m_selected_patterns;
52     CListBox    m_patterns;
53     CEdit   m_target_set_file;
54     CEdit   m_target_size;
55     CEdit   m_genome_size;
56     CEdit   m_match_size;
57     CEdit   m_population_size;
58     CEdit   m_number_of_generations;
59     CEdit   m_mutation_rate;
60     CEdit   m_crossover_rate;
61     //}}AFX_DATA
62
63     // ClassWizard generated virtual function overrides
64     //{{AFX_VIRTUAL(CStringCoveringDlg)
65     protected:
66     virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
67     //}}AFX_VIRTUAL
68
69 // Implementation
70
71 private:
72     void InitializeValues();
73     int target_size;
74     int population_size;
75     int match_size;
76     int number_of_generations;
77     double crossover_rate;
78     double mutation_rate;
79     int genome_size;
80
81     CSpinButtonCtrl* spin;
82
83     CDC* pDC;
```

```
84      int target_set_size;
85
86      int gen_no;
87      double matched_bits;
88
89      int w, h;
90      int last_x, last_y;
91
92      bool running;
93  protected:
94      HICON m_hIcon;
95
96      // Generated message map functions
97      //{{AFX_MSG(CStringCoveringDlg)
98      virtual BOOL OnInitDialog();
99      afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
100     afx_msg void OnPaint();
101     afx_msg HCURSOR OnQueryDragIcon();
102     afx_msg void OnGo();
103     afx_msg void OnLoadTargetFile();
104     afx_msg void OnLoadPatterns();
105     afx_msg void OnSelectPatterns();
106     afx_msg void OnRemoveSelected();
107     afx_msg void OnClearPatterns();
108     afx_msg void OnGenerateTargetSet();
109     afx_msg void OnDblclkPatterns();
110     afx_msg void OnDblclkSelectedPatterns();
111     afx_msg void OnKillfocusTsize();
112     afx_msg void OnKillfocusPopulationSize();
113     afx_msg void OnKillfocusGenomesNo();
114     afx_msg void OnKillfocusMutationRate();
115     afx_msg void OnKillfocusCrossoverRate();
116     afx_msg void OnKillfocusNoOfGenerations();
117     virtual void OnOK();
118     afx_msg void OnAbout();
119     afx_msg void OnClose();
120     //}}AFX_MSG
121     DECLARE_MESSAGE_MAP()
122 };
123
124 //{{AFX_INSERT_LOCATION}}
125 // Microsoft Visual C++ will insert additional declarations immediately before the previous line.
126
127 #endif // !defined(AFX_STRINGCOVERINGDLG_H__C7A2133D_47D6_42CB_B33B_711F041DF8DF__INCLUDED_)
128
```

```cpp
////////////////////////////////////////////////
// (c) 2003 Sorin OSTAFIEV (sorin@ostafiev.com) //
////////////////////////////////////////////////

// StringCoveringDlg.cpp : implementation file
//

#include "stdafx.h"
#include "StringCovering.h"
#include "StringCoveringDlg.h"


#include "GA.h"
#include "Population.h"
#include "Genome.h"
#include "MyStatic.h"


#include <string>
#include <fstream>
#include <cassert>
using namespace std;



#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif



/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

// Dialog Data
    //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CAboutDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    //{{AFX_MSG(CAboutDlg)
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{{AFX_DATA_INIT(CAboutDlg)
    //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAboutDlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{{AFX_MSG_MAP(CAboutDlg)
        // No message handlers
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CStringCoveringDlg dialog
const COLORREF background_color = RGB(192, 192, 200);
const COLORREF line_color = RGB(150, 100, 50);

```

```
84
85
86  CStringCoveringDlg::CStringCoveringDlg(CWnd* pParent /*=NULL*/)
87      : CDialog(CStringCoveringDlg::IDD, pParent)
88  {
89      //{{AFX_DATA_INIT(CStringCoveringDlg)
90          // NOTE: the ClassWizard will add member initialization here
91      //}}AFX_DATA_INIT
92      // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
93      InitializeValues();
94      pDC = NULL;
95      spin = NULL;
96      m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
97
98      m_evolution.background_color = background_color;
99      m_evolution.line_color = line_color;
100
101 }
102
103 void CStringCoveringDlg::DoDataExchange(CDataExchange* pDX)
104 {
105     CDialog::DoDataExchange(pDX);
106     //{{AFX_DATA_MAP(CStringCoveringDlg)
107     DDX_Control(pDX, IDOK, m_ok);
108     DDX_Control(pDX, IDC_LOAD_TARGET_FILE, m_load_target);
109     DDX_Control(pDX, IDC_LOAD_PATTERNS, m_load_patterns);
110     DDX_Control(pDX, IDC_SELECT_PATTERNS, m_select_pattern);
111     DDX_Control(pDX, IDC_GENERATE_TARGET_SET, m_generate);
112     DDX_Control(pDX, IDC_REMOVE_SELECTED, m_remove_selected);
113     DDX_Control(pDX, IDC_CLEAR_PATTERNS, m_clear_patterns);
114     DDX_Control(pDX, IDC_GO, m_go);
115     DDX_Control(pDX, IDC_EVOLUTION, m_evolution);
116     DDX_Control(pDX, IDC_BESTS, m_bests);
117     DDX_Control(pDX, IDC_PROGRESS, m_progress);
118     DDX_Control(pDX, IDC_GENNO, m_gen_no);
119     DDX_Control(pDX, IDC_MATCHED_BITS, m_matched_bits);
120     DDX_Control(pDX, IDC_TSIZE, m_target_set_size);
121     DDX_Control(pDX, IDC_SELECTED_PATTERNS, m_selected_patterns);
122     DDX_Control(pDX, IDC_PATTERNS, m_patterns);
123     DDX_Control(pDX, IDC_TARGET_SET_FILE, m_target_set_file);
124     DDX_Control(pDX, IDC_TARGET_SIZE, m_target_size);
125     DDX_Control(pDX, IDC_GENOME_SIZE, m_genome_size);
126     DDX_Control(pDX, IDC_GENOMES_NO, m_match_size);
127     DDX_Control(pDX, IDC_POPULATION_SIZE, m_population_size);
128     DDX_Control(pDX, IDC_NO_OF_GENERATIONS, m_number_of_generations);
129     DDX_Control(pDX, IDC_MUTATION_RATE, m_mutation_rate);
130     DDX_Control(pDX, IDC_CROSSOVER_RATE, m_crossover_rate);
131     //}}AFX_DATA_MAP
132 }
133
134 BEGIN_MESSAGE_MAP(CStringCoveringDlg, CDialog)
135     //{{AFX_MSG_MAP(CStringCoveringDlg)
136     ON_WM_SYSCOMMAND()
137     ON_WM_PAINT()
138     ON_WM_QUERYDRAGICON()
139     ON_BN_CLICKED(IDC_GO, OnGo)
140     ON_BN_CLICKED(IDC_LOAD_TARGET_FILE, OnLoadTargetFile)
141     ON_BN_CLICKED(IDC_LOAD_PATTERNS, OnLoadPatterns)
142     ON_BN_CLICKED(IDC_SELECT_PATTERNS, OnSelectPatterns)
143     ON_BN_CLICKED(IDC_REMOVE_SELECTED, OnRemoveSelected)
144     ON_BN_CLICKED(IDC_CLEAR_PATTERNS, OnClearPatterns)
145     ON_BN_CLICKED(IDC_GENERATE_TARGET_SET, OnGenerateTargetSet)
146     ON_LBN_DBLCLK(IDC_PATTERNS, OnDblclkPatterns)
147     ON_LBN_DBLCLK(IDC_SELECTED_PATTERNS, OnDblclkSelectedPatterns)
148     ON_EN_KILLFOCUS(IDC_TSIZE, OnKillfocusTsize)
149     ON_EN_KILLFOCUS(IDC_POPULATION_SIZE, OnKillfocusPopulationSize)
150     ON_EN_KILLFOCUS(IDC_GENOMES_NO, OnKillfocusGenomesNo)
151     ON_EN_KILLFOCUS(IDC_MUTATION_RATE, OnKillfocusMutationRate)
152     ON_EN_KILLFOCUS(IDC_CROSSOVER_RATE, OnKillfocusCrossoverRate)
153     ON_EN_KILLFOCUS(IDC_NO_OF_GENERATIONS, OnKillfocusNoOfGenerations)
154     ON_BN_CLICKED(IDC_ABOUT, OnAbout)
155     ON_WM_CLOSE()
156     //}}AFX_MSG_MAP
157 END_MESSAGE_MAP()
158
159 /////////////////////////////////////////////////////////////////////////////
160 // CStringCoveringDlg message handlers
161
162
163 int Genome::_genome_size = MAX_GENOME_LENGHT;
164
165 Population* targetSet = NULL;
166
```

```
167 const string target_set_file_name = "targetSet.txt";
168 const string patterns_file = "patterns.txt";
169
170 vector<string> patterns;
171 vector<string> selected_patterns;
172 vector<string> bests;
173
174 CStringCoveringDlg* mainDlg = NULL;
175
176
177 string stringFromDouble(const double d)
178 {
179     char ss[100];
180     sprintf(ss, "%f", d);
181
182     string s(ss);
183
184     string::size_type p = s.find_first_of(".");
185     if (p != string::npos)
186     {
187         if (p + 4 < s.size())
188         {
189             s = string(s, 0, p + 4);
190         };
191
192     };
193
194
195     return s;
196 };
197
198 string stringFromInt(const int i)
199 {
200     char s[100];
201     itoa(i, s, 10);
202     return string(s);
203 };
204
205 int intFromString(const string& s)
206 {
207     int i = 0;
208     i = atoi(s.c_str());
209     return i;
210 };
211
212 double doubleFromString(const string& s)
213 {
214     double d = 0.0f;
215     d = atof(s.c_str());
216     return d;
217 };
218
219 string stringFromCWnd(const CWnd* const p)
220 {
221     char s[2001];
222     p->GetWindowText(s, 2000);
223     return string(s);
224 };
225
226 BOOL CStringCoveringDlg::OnInitDialog()
227 {
228     CDialog::OnInitDialog();
229
230     // Add "About..." menu item to system menu.
231
232     // IDM_ABOUTBOX must be in the system command range.
233     ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
234     ASSERT(IDM_ABOUTBOX < 0xF000);
235
236     CMenu* pSysMenu = GetSystemMenu(FALSE);
237     if (pSysMenu != NULL)
238     {
239         CString strAboutMenu;
240         strAboutMenu.LoadString(IDS_ABOUTBOX);
241         if (!strAboutMenu.IsEmpty())
242         {
243             pSysMenu->AppendMenu(MF_SEPARATOR);
244             pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
245         }
246     }
247
248     // Set the icon for this dialog.  The framework does this automatically
249     //  when the application's main window is not a dialog
```

```
250     SetIcon(m_hIcon, TRUE);         // Set big icon
251     SetIcon(m_hIcon, FALSE);        // Set small icon
252
253     // TODO: Add extra initialization here
254
255
256     running = false;
257
258     spin = (CSpinButtonCtrl*)GetDlgItem(IDC_SELECT_NO_GENOMES);
259     assert (NULL != spin);
260     spin->SetBuddy(&m_match_size);
261     spin->SetRange(1, 100);
262
263
264     m_progress.SetRange(0, 100);
265
266     OnLoadTargetFile();
267     OnLoadPatterns();
268     UpdateDialogFromValues();
269
270
271     pDC = m_evolution.GetDC();
272
273     RECT rect;
274     m_evolution.GetClientRect(&rect);
275     w = rect.right;
276     h = rect.bottom;
277
278     m_evolution.w = w;
279     m_evolution.h = h;
280
281     //CBitmap* bmp = new CBitmap();
282     //BOOL boo = bmp->CreateCompatibleBitmap(pDC, 1000, 1000);
283     //pDC->SelectObject(bmp);
284
285
286     return TRUE;  // return TRUE  unless you set the focus to a control
287 }
288
289 void CStringCoveringDlg::OnSysCommand(UINT nID, LPARAM lParam)
290 {
291     if ((nID & 0xFFF0) == IDM_ABOUTBOX)
292     {
293         CAboutDlg dlgAbout;
294         dlgAbout.DoModal();
295     }
296     else
297     {
298         CDialog::OnSysCommand(nID, lParam);
299     }
300 }
301
302 // If you add a minimize button to your dialog, you will need the code below
303 //  to draw the icon.  For MFC applications using the document/view model,
304 //  this is automatically done for you by the framework.
305
306 void CStringCoveringDlg::OnPaint()
307 {
308     if (IsIconic())
309     {
310         CPaintDC dc(this); // device context for painting
311
312         SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);
313
314         // Center icon in client rectangle
315         int cxIcon = GetSystemMetrics(SM_CXICON);
316         int cyIcon = GetSystemMetrics(SM_CYICON);
317         CRect rect;
318         GetClientRect(&rect);
319         int x = (rect.Width() - cxIcon + 1) / 2;
320         int y = (rect.Height() - cyIcon + 1) / 2;
321
322         // Draw the icon
323         dc.DrawIcon(x, y, m_hIcon);
324     }
325     else
326     {
327         CDialog::OnPaint();
328     }
329 }
330
331 // The system calls this to obtain the cursor to display while the user drags
332 //  the minimized window.
```

```
333 HCURSOR CStringCoveringDlg::OnQueryDragIcon()
334 {
335     return (HCURSOR) m_hIcon;
336 }
337 
338 
339 unsigned int DoEvolution(void *data);
340 
341 void CStringCoveringDlg::OnGo()
342 {
343     // TODO: Add your control notification handler code here
344 
345     if (false == running)
346     {
347 
348         if (false == UpdateValuesFromDialog())
349         {
350             AfxMessageBox("Invalid parameters!", MB_OK | MB_ICONSTOP);
351 
352         }
353         else
354         {
355 
356             srand(time(NULL));
357 
358             m_evolution.v.clear();
359             assert (NULL != targetSet);
360 
361             last_x = -1;
362             last_y = -1;
363 
364             mainDlg = this;
365 
366             running = true;
367             DisableInterface();
368             m_go.SetWindowText("Stop...");
369             CWinThread* interfaceThread = AfxBeginThread(DoEvolution, NULL);
370 
371         };
372     }
373     else
374     { // oops! we are in the middle of something...
375 
376         running = false;
377     };
378 
379 }
380 
381 bool CStringCoveringDlg::UpdateValuesFromDialog()
382 {
383     target_set_size = intFromString(stringFromCWnd(&m_target_set_size));
384     population_size = intFromString(stringFromCWnd(&m_population_size));
385     match_size = intFromString(stringFromCWnd(&m_match_size));
386 
387     mutation_rate = doubleFromString(stringFromCWnd(&m_mutation_rate));
388     crossover_rate = doubleFromString(stringFromCWnd(&m_crossover_rate));
389 
390     number_of_generations = intFromString(stringFromCWnd(&m_number_of_generations));
391 
392     if (target_set_size <= 0)
393     {
394         return false;
395     };
396 
397     if ((population_size <= 0) || (1000 <= population_size))
398     {
399         return false;
400     };
401 
402     if (match_size <= 0)
403     {
404         return false;
405     };
406 
407     if (number_of_generations <= 0)
408     {
409         return false;
410     };
411 
412     if ((mutation_rate < 0) || (1 < mutation_rate))
413     {
414         return false;
415     };
```

```
416
417     if ((crossover_rate < 0) || (1 < crossover_rate))
418     {
419         return false;
420     };
421
422     return true;
423
424 }
425
426
427 bool CStringCoveringDlg::UpdateValuesFromDialog1()
428 {
429     int tsize = 0;
430
431     tsize = intFromString(stringFromCWnd(&m_target_set_size));
432
433     const int count_selected = selected_patterns.size();
434
435     if ((0 < count_selected) && (0 < tsize))
436     {
437         target_set_size = tsize;
438         return true;
439     };
440
441
442     return false;
443
444 }
445
446 void CStringCoveringDlg::UpdateDialogFromValues()
447 {
448     m_match_size.SetWindowText(stringFromInt(match_size).c_str());
449     m_population_size.SetWindowText(stringFromInt(population_size).c_str());
450     m_number_of_generations.SetWindowText(stringFromInt(number_of_generations).c_str());
451     m_mutation_rate.SetWindowText(stringFromDouble(mutation_rate).c_str());
452     m_crossover_rate.SetWindowText(stringFromDouble(crossover_rate).c_str());
453     m_genome_size.SetWindowText(stringFromInt(genome_size).c_str());
454     m_target_size.SetWindowText(stringFromInt(target_size).c_str());
455     m_target_set_file.SetWindowText(target_set_file_name.c_str());
456
457
458     m_target_set_size.SetWindowText(stringFromInt(target_set_size).c_str());
459
460
461     m_patterns.ResetContent();
462     for (int i = 0; i < patterns.size(); i++)
463     {
464         m_patterns.AddString(patterns.at(i).c_str());
465
466     };
467
468     m_selected_patterns.ResetContent();
469     for (i = 0; i < selected_patterns.size(); i++)
470     {
471         m_selected_patterns.AddString(selected_patterns.at(i).c_str());
472
473     };
474
475 }
476
477 void CStringCoveringDlg::InitializeValues()
478 {
479     population_size = 50;
480     match_size = 3;
481     number_of_generations = 200;
482     crossover_rate = 0.6;
483     mutation_rate = 0.15;
484     genome_size = 32;
485     target_size = 0;
486
487     target_set_size = 100;
488
489 }
490
491 void CStringCoveringDlg::OnLoadTargetFile()
492 {
493     // TODO: Add your control notification handler code here
494     ifstream target(target_set_file_name.c_str());
495
496
497     int n = 0;
498     target >> n;
```

```
499     assert (0 < n);
500
501
502     delete targetSet;
503     targetSet = NULL;
504
505     targetSet = new Population(NULL, n, Genome::GENOME_ZERO);
506
507
508
509     string s("");
510     int i = 0;
511     while (false == target.eof())
512     {
513         getline(target, s);
514         if ("" != s)
515         {
516             targetSet->_genomes.at(i).setFromPattern(s);
517
518             i++;
519
520         };
521
522     };
523
524
525     assert (n == i);
526
527     target_size = n;
528
529     assert (NULL != targetSet);
530
531     UpdateDialogFromValues();
532
533 }
534
535 void CStringCoveringDlg::OnLoadPatterns()
536 {
537     // TODO: Add your control notification handler code here
538     patterns.clear();
539     selected_patterns.clear();
540
541     ifstream patterns_stream(patterns_file.c_str());
542
543     string s("");
544     while (false == patterns_stream.eof())
545     {
546         getline(patterns_stream, s);
547         if ("" != s)
548         {
549             patterns.push_back(s);
550         };
551     };
552
553     UpdateDialogFromValues();
554
555 }
556
557 void CStringCoveringDlg::OnSelectPatterns()
558 {
559     // TODO: Add your control notification handler code here
560     vector <string> tmp(selected_patterns);
561
562     char s[2000];
563
564     for (int i = 0; i < m_patterns.GetCount(); i++)
565     {
566         if (0 != m_patterns.GetSel(i))
567         {
568             m_patterns.GetText(i, s);
569             const string what_to_insert(s);
570
571             bool sw = true;
572             for (int k = 0; k < tmp.size(); k++)
573             {
574                 if (what_to_insert == tmp.at(k)) // already in list
575                 {
576                     sw = false;
577                 };
578             };
579
580
581             if (true == sw)
```

```
582             {
583                 tmp.push_back(what_to_insert);
584             };
585         };
586     };
587 
588 
589     selected_patterns = tmp;
590 
591     UpdateDialogFromValues();
592 }
593 
594 void CStringCoveringDlg::OnRemoveSelected()
595 {
596     // TODO: Add your control notification handler code here
597     vector <string> tmp;
598 
599     char s[2000];
600 
601     for (int i = 0; i < m_selected_patterns.GetCount(); i++)
602     {
603         if (0 == m_selected_patterns.GetSel(i)) // if not selected... we keep it
604         {
605             m_patterns.GetText(i, s);
606             const string what_to_insert(s);
607 
608             tmp.push_back(what_to_insert);
609         };
610     };
611 
612 
613     selected_patterns = tmp;
614 
615     UpdateDialogFromValues();
616 
617 }
618 
619 void CStringCoveringDlg::OnClearPatterns()
620 {
621     // TODO: Add your control notification handler code here
622     selected_patterns.clear();
623 
624     UpdateDialogFromValues();
625 }
626 
627 void CStringCoveringDlg::OnGenerateTargetSet()
628 {
629     // TODO: Add your control notification handler code here
630     if (false == UpdateValuesFromDialog1())
631     {
632         AfxMessageBox("Invalid parameters!", MB_OK | MB_ICONSTOP);
633 
634     }
635     else
636     {
637 
638         Population population(NULL, target_set_size, Genome::GENOME_ZERO);
639 
640         const int patterns_no = selected_patterns.size();
641         assert (0 < patterns_no);
642 
643 
644 
645         for (int i = 0; i < target_set_size; i++)
646         {
647             const int what_pattern = rand() % patterns_no;
648             assert ((0 <= what_pattern) && (what_pattern < patterns_no));
649 
650             const string& p = selected_patterns.at(what_pattern);
651 
652             population._genomes.at(i).setFromPattern(p);
653         };
654 
655         {
656             ofstream of(target_set_file_name.c_str());
657             of << target_set_size << endl;
658             for (i = 0; i < target_set_size; i++)
659             {
660                 of << population._genomes.at(i)._body << endl;
661             };
662         };
663 
664         ShellExecute(NULL, NULL, target_set_file_name.c_str(), NULL, NULL, SW_SHOWNORMAL);
```

```
665
666
667     };
668 }
669
670 void CStringCoveringDlg::OnDblclkPatterns()
671 {
672     // TODO: Add your control notification handler code here
673     OnSelectPatterns();
674 }
675
676 void CStringCoveringDlg::OnDblclkSelectedPatterns()
677 {
678     // TODO: Add your control notification handler code here
679     OnRemoveSelected();
680 }
681
682 void CStringCoveringDlg::OnKillfocusTsize()
683 {
684     // TODO: Add your control notification handler code here
685     UpdateValuesFromDialog();
686     UpdateValuesFromDialog1();
687 }
688
689 void CStringCoveringDlg::OnKillfocusPopulationSize()
690 {
691     // TODO: Add your control notification handler code here
692     UpdateValuesFromDialog();
693     UpdateValuesFromDialog1();
694
695 }
696
697 void CStringCoveringDlg::OnKillfocusGenomesNo()
698 {
699     // TODO: Add your control notification handler code here
700     UpdateValuesFromDialog();
701     UpdateValuesFromDialog1();
702
703 }
704
705 void CStringCoveringDlg::OnKillfocusMutationRate()
706 {
707     // TODO: Add your control notification handler code here
708     UpdateValuesFromDialog();
709     UpdateValuesFromDialog1();
710
711 }
712
713 void CStringCoveringDlg::OnKillfocusCrossoverRate()
714 {
715     // TODO: Add your control notification handler code here
716     UpdateValuesFromDialog();
717     UpdateValuesFromDialog1();
718
719 }
720
721 void CStringCoveringDlg::OnKillfocusNoOfGenerations()
722 {
723     // TODO: Add your control notification handler code here
724     UpdateValuesFromDialog();
725     UpdateValuesFromDialog1();
726
727 }
728
729 void CStringCoveringDlg::OnOK()
730 {
731     // TODO: Add extra validation here
732
733     m_evolution.ReleaseDC(pDC);
734     pDC = NULL;
735
736     delete targetSet;
737     targetSet = NULL;
738
739     CDialog::OnOK();
740 }
741
742 void CStringCoveringDlg::UpdateEvolutionPanel()
743 {
744     assert ((0 <= gen_no) && (gen_no < number_of_generations));
745
746
747     m_gen_no.SetWindowText((string("generation: ") + stringFromInt(gen_no + 1)).c_str());
```

```
748
749     m_matched_bits.SetWindowText((string("current matched bits: ") + stringFromDouble(matched_bits)).c_str());
750
751     m_progress.SetPos(((gen_no + 1) * 100) / number_of_generations);
752
753     m_bests.ResetContent();
754     for (int i = 0; i < bests.size(); i++)
755     {
756         m_bests.AddString(bests.at(i).c_str());
757     };
758
759
760     int x = (gen_no * w) / (number_of_generations - 1);
761     int y = ((MAX_GENOME_LENGHT - matched_bits) * h) / MAX_GENOME_LENGHT;
762
763     m_evolution.v.push_back(make_pair<int, int>(x, y));
764     m_evolution.Invalidate(FALSE);
765
766     last_x = x;
767     last_y = y;
768
769
770 }
771
772
773 unsigned int DoEvolution(void *data)
774 {
775     assert(NULL != mainDlg);
776
777
778     mainDlg->pDC->FillSolidRect(0, 0, mainDlg->w, mainDlg->h, background_color);
779
780     bool sw_stopped = false;
781
782     // GA(match_size, population_size, cross_over_rate, mutation_rate)
783     GA ga(mainDlg->match_size, mainDlg->population_size, mainDlg->crossover_rate, mainDlg->mutation_rate);
784
785     {
786
787         for (int i = 0; i < mainDlg->number_of_generations; i++)
788         {
789
790             ga.step();
791
792             mainDlg->gen_no = i;
793             mainDlg->matched_bits = ga.getAverageMatchedBits();
794
795             {
796                 bests.clear();
797                 bests.reserve(ga._populations.size());
798
799                 for (int k = 0; k < ga._populations.size(); k++)
800                 {
801                     assert (0 < ga._populations.at(k)._genomes.size());
802
803                     bests.push_back(ga._populations.at(k)._genomes.at(0)._body.to_string());
804
805                 };
806             };
807
808
809             mainDlg->UpdateEvolutionPanel();
810
811
812             if (false == mainDlg->running) // hmmm... someone stopped us
813             {
814                 AfxMessageBox("Stopped...");
815
816                 sw_stopped = true;
817                 i = mainDlg->number_of_generations;
818                 mainDlg->m_go.SetWindowText("GO");
819                 mainDlg->EnableInterface();
820
821
822             };
823
824             if (MAX_GENOME_LENGHT == mainDlg->matched_bits)
825             {
826                 AfxMessageBox("You've reached GOD!\nEvolution cannot continue anymore.", MB_OK | MB_ICONINFORMATIOI
827
828                 i = mainDlg->number_of_generations;
829                 mainDlg->m_go.SetWindowText("GO");
830                 mainDlg->EnableInterface();
```

```
                sw_stopped = true;
                mainDlg->running = false;


            };

        };
    };

    if (false == sw_stopped)
    {
        AfxMessageBox("Done");

        mainDlg->running = false;
        mainDlg->m_go.SetWindowText("GO");
        mainDlg->EnableInterface();

        mainDlg->m_evolution.Invalidate();
    };

    assert (false == mainDlg->running);
    return 0;
};

void CStringCoveringDlg::OnAbout()
{
    // TODO: Add your control notification handler code here
    CAboutDlg about;
    about.DoModal();

}

void CStringCoveringDlg::DisableInterface()
{
    m_target_set_size.EnableWindow(FALSE);
    m_patterns.EnableWindow(FALSE);
    m_selected_patterns.EnableWindow(FALSE);
    m_clear_patterns.EnableWindow(FALSE);
    m_remove_selected.EnableWindow(FALSE);
    m_generate.EnableWindow(FALSE);
    m_select_pattern.EnableWindow(FALSE);
    m_load_patterns.EnableWindow(FALSE);
    m_target_set_file.EnableWindow(FALSE);
    m_load_target.EnableWindow(FALSE);
    m_match_size.EnableWindow(FALSE);
    spin->EnableWindow(FALSE);
    m_mutation_rate.EnableWindow(FALSE);
    m_crossover_rate.EnableWindow(FALSE);
    m_number_of_generations.EnableWindow(FALSE);
    m_ok.EnableWindow(FALSE);
    m_population_size.EnableWindow(FALSE);

}

void CStringCoveringDlg::EnableInterface()
{
    m_target_set_size.EnableWindow(TRUE);
    m_patterns.EnableWindow(TRUE);
    m_selected_patterns.EnableWindow(TRUE);
    m_clear_patterns.EnableWindow(TRUE);
    m_remove_selected.EnableWindow(TRUE);
    m_generate.EnableWindow(TRUE);
    m_select_pattern.EnableWindow(TRUE);
    m_load_patterns.EnableWindow(TRUE);
    m_target_set_file.EnableWindow(TRUE);
    m_load_target.EnableWindow(TRUE);
    m_match_size.EnableWindow(TRUE);
    spin->EnableWindow(TRUE);
    m_mutation_rate.EnableWindow(TRUE);
    m_crossover_rate.EnableWindow(TRUE);
    m_number_of_generations.EnableWindow(TRUE);
    m_ok.EnableWindow(TRUE);
    m_population_size.EnableWindow(TRUE);

}

void CStringCoveringDlg::OnClose()
{
    // TODO: Add your message handler code here and/or call default
    if (true == running)
    {
        AfxMessageBox("Evolution in progress.\nYou cannot stop it!", MB_OK | MB_ICONINFORMATION);
        return;
```

```
914       };
915
916       CDialog::OnClose();
917   }
918
```

```
1  ////////////////////////////////////////////////
2  // (c) 2003 Sorin OSTAFIEV (sorin@ostafiev.com) //
3  ////////////////////////////////////////////////
4
5  #if !defined(AFX_MYSTATIC_H__8CF9223E_205D_484B_9D3B_FF7CC1DF1DC1__INCLUDED_)
6  #define AFX_MYSTATIC_H__8CF9223E_205D_484B_9D3B_FF7CC1DF1DC1__INCLUDED_
7
8  #if _MSC_VER > 1000
9  #pragma once
10 #endif // _MSC_VER > 1000
11 // MyStatic.h : header file
12 //
13
14 /////////////////////////////////////////////////////////////////////////////
15 // CMyStatic window
16
17 #include <vector>
18 using namespace std;
19
20 class CMyStatic : public CStatic
21 {
22 // Construction
23 public:
24     CMyStatic();
25
26 // Attributes
27 public:
28
29 // Operations
30 public:
31
32 // Overrides
33     // ClassWizard generated virtual function overrides
34     //{{AFX_VIRTUAL(CMyStatic)
35     //}}AFX_VIRTUAL
36
37 // Implementation
38 public:
39     COLORREF background_color;
40     COLORREF line_color;
41     int w, h;
42     virtual ~CMyStatic();
43
44     vector < pair<int, int> > v;
45     // Generated message map functions
46 protected:
47     //{{AFX_MSG(CMyStatic)
48     afx_msg BOOL OnEraseBkgnd(CDC* pDC);
49     afx_msg void OnPaint();
50     //}}AFX_MSG
51
52     DECLARE_MESSAGE_MAP()
53 };
54
55 /////////////////////////////////////////////////////////////////////////////
56
57 //{{AFX_INSERT_LOCATION}}
58 // Microsoft Visual C++ will insert additional declarations immediately before the previous line.
59
60 #endif // !defined(AFX_MYSTATIC_H__8CF9223E_205D_484B_9D3B_FF7CC1DF1DC1__INCLUDED_)
61
```

```cpp
////////////////////////////////////////////////
// (c) 2003 Sorin OSTAFIEV (sorin@ostafiev.com) //
////////////////////////////////////////////////

// MyStatic.cpp : implementation file
//

#include "stdafx.h"
#include "stringcovering.h"
#include "MyStatic.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMyStatic

CMyStatic::CMyStatic():
    w(0),
    h(0)
{
}

CMyStatic::~CMyStatic()
{
}


BEGIN_MESSAGE_MAP(CMyStatic, CStatic)
    //{{AFX_MSG_MAP(CMyStatic)
    ON_WM_ERASEBKGND()
    ON_WM_PAINT()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMyStatic message handlers


BOOL CMyStatic::OnEraseBkgnd(CDC* pDC)
{
    // TODO: Add your message handler code here and/or call default
    pDC->FillSolidRect(0,0, w, h, background_color);
    return true;

    return CStatic::OnEraseBkgnd(pDC);
}

void CMyStatic::OnPaint()
{
    CPaintDC dc(this); // device context for painting

    // TODO: Add your message handler code here
    CPen pen(PS_SOLID, 2, line_color);
    CPen* old_pen = dc.SelectObject(&pen);

    for (int i = 0; i < v.size(); i++)
    {
        const int x = v.at(i).first;
        const int y = v.at(i).second;
        if (0 == i)
        {
            dc.MoveTo(x, y);
            dc.SetPixel(x, y, line_color);
        }
        else
        {

            dc.LineTo(x, y);
        };

    };

    dc.SelectObject(old_pen);

    // Do not call CStatic::OnPaint() for painting messages
}

```