NA CD: Pełna wersja pakietu SPHINX 4.0 z systemem ekspertowym PC-Shell!
Bootowalna płyta z gotowymi do uruchomienia programami i przykładami z numeru – wypróbuj natychmiast bez instalacji • Stuttgart Neural Network Simulator 4.2 • Java Neural Network Simulator 1.1 • NeuroSolutions 4.3 • pmars 0.9.2 • Corewars 0.9.13 • Framsticks 2.9 rc3

Software 2.0 Extra!
Numer 10

Sztuczna Inteligencja

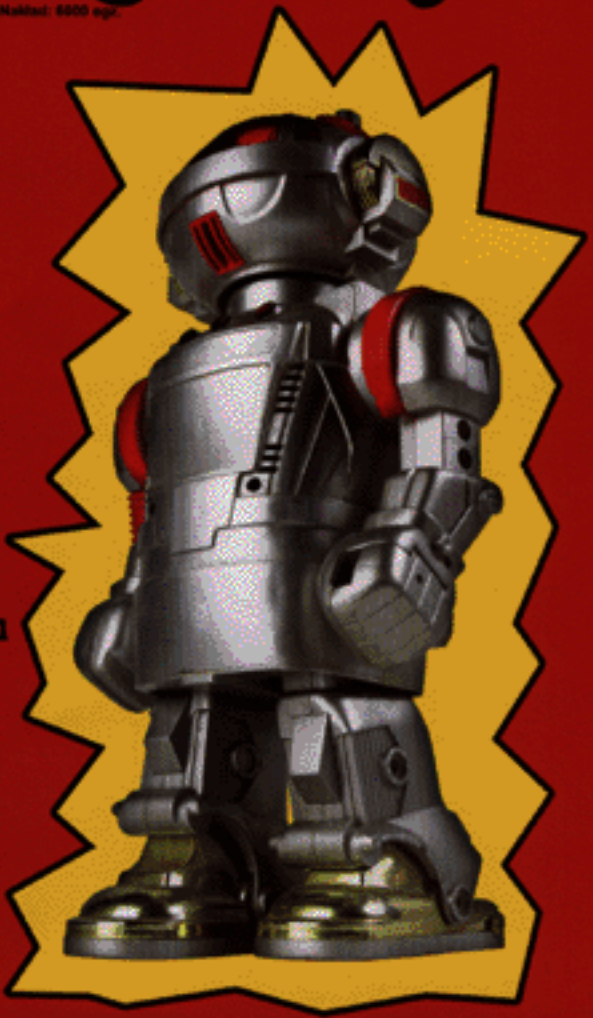
Cena: 29,80 zł ISSN 1644-6615 Stawka VAT 0% Nakład: 6000 egz.

Chatterboty
Jak komputer rozumie człowieka

Algorytmy genetyczne
Dobór naturalny najlepszych rozwiązań

Core Wars
Tworzymy inteligentnego wojownika

OCR z siecią neuronową
Własny system rozpoznawania znaków



Algorytmy genetyczne w praktyce

Jacek Czerniak

Algorytmy genetyczne są istotną dziedziną sztucznej inteligencji i znajdują zastosowanie w rozwiązywaniu takich problemów optymalizacji, jak układanie planów zajęć czy opisana w artykule Tomasza Michniewskiego optymalizacja funkcji oceniającej programu szachowego. W tym artykule poznamy algorytmy genetyczne na przykładzie konkretnej implementacji w C++ Builderze. Nasz program nie będzie rozwiązywał żadnego konkretnego zadania – posłuży nam jedynie do zamodelowania procesów ewolucyjnych oraz wygenerowania jak najsilniejszej populacji osobników.

Szybki start

Opisywane w tym artykule algorytmy stanowią część dołączonego na płycie programu *Genetyk*. Aby go włączyć wystarczy uruchomić komputer z płyty, a po załadowaniu systemu kliknąć ikonę *Genetyk* w katalogu *Scripts*. Program można też zainstalować w systemie Windows – wystarczy uruchomić plik instalatora *setup.exe* i postępować zgodnie z instrukcjami na ekranie.

Mechanizmy ewolucyjne

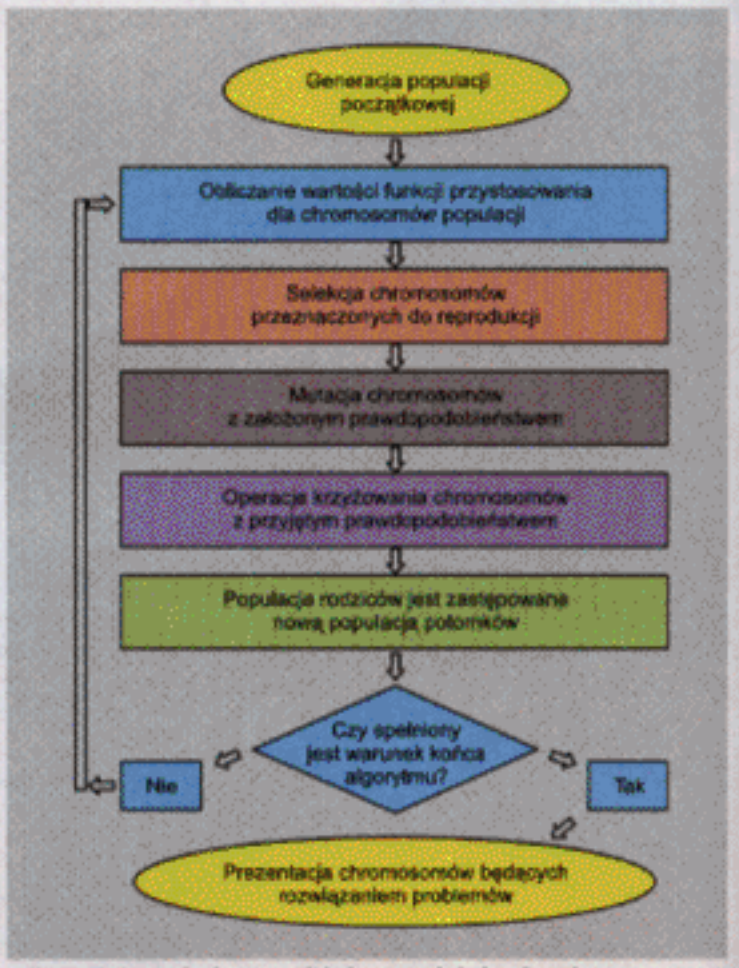
W naszym algorytmie genetycznym pojedynczemu genowi odpowiada jeden bit (o stanie logicznym 1 lub 0), natomiast chromosom jest ciągiem takich bitów (czyli genów). Przykładową populację zapisanych w tej konwencji osobników przedstawia Rysunek 2. Osobniki widoczne na rysunku to faktycznie same chromosomy, więc na razie będziemy posługiwać się zamiennie pojęciami chromosom i osobnik, choć w dalszej części artykułu różnica ta będzie już miała znaczenie.

Funkcja przystosowania

Funkcja przystosowania (dopasowania) opisuje kierunek ewolucji algorytmu. Funkcja ta jest kluczowym elementem algorytmu genetycznego, gdyż najczęściej konstruujemy algorytm nie tylko po to, by ewoluował, ale przede wszystkim by rozwiązał jakiś problem (dokonał optymalizacji badanego procesu). Niewłaściwe sformułowanie funkcji przystosowania może zatem w skrajnym przypadku doprowadzić do tego, że nasz algorytm wyłoni osobnika o niepożądanych dla nas własnościach i w ten sposób oczekując spotkania z doktorem Jekyllem spotkamy się z panem Hydem... Przyjmijmy zatem, że wychodząc od jakiejś bliżej nieznanej populacji początkowej chcemy uzyskać populację, w której osobniki będą miały przewagę jedynek w swoich chromosomach. Oczekujemy zatem populacji, w której będzie przynajmniej jeden Superman, którego chromosom składa się z samych jedynek, a u pozostałych osobników liczba jedynek jest znacznie podwyższona. Rysunek 3 przedstawia naszą populację początkową wraz z wynikiem funkcji przystosowania `f(x)`.

Selekcja

Mechanizm selekcji w naszym algorytmie będzie odzwierciedleniem obowiązującego w naturze prawa przetrwania najsilniejszych (najlepiej przystosowanych). Chcemy zatem, by silniejszy osobnik miał większe szanse zostać protoplastą nowego pokolenia niż osobnik słabszy. Standardowym rozwiązaniem jest tutaj metoda koła ruletki, według której każdemu osobnikowi przydzielany jest wycinek koła o powierzchni odpowiadającej stop-



Rysunek 1. *Schemat blokowy działania algorytmu genetycznego*

Autor nabierał doświadczenia pracując jako programista w dziale R&D firmy Lucent Technologies. Obecnie jest pracownikiem naukowo-dydaktycznym na Wydziale Informatyki Politechniki Szczecińskiej.
Kontakt z autorem: *jczerniak@wi.ps.pl*

Podstawowe pojęcia genetyczne

- Osobnik: podstawowy uczestnik operacji genetycznych; w programie odpowiada mu pewna struktura danych.
- Populacja: zbiór osobników, które biorą udział w procesie ewolucji.
- Chromosom: zestaw genów opisujących osobnika; w naszym przypadku jest łańcuch znaków.
- Selekcja: proces wyboru osobników na rodziców nowej populacji, odpowiadający mechanizmowi selekcji naturalnej.
- Krzyżowanie: operacja wymiany niektórych genów między dwoma osobnikami wybranymi w procesie selekcji.
- Mutacja: zmiana losowo wybranego genu (podobnie jak w naturze).
- Funkcja przystosowania: funkcja określająca stopień zgodności cech danego osobnika z założonym trendem rozwoju populacji, czyli stopień przystosowania osobnika do środowiska.

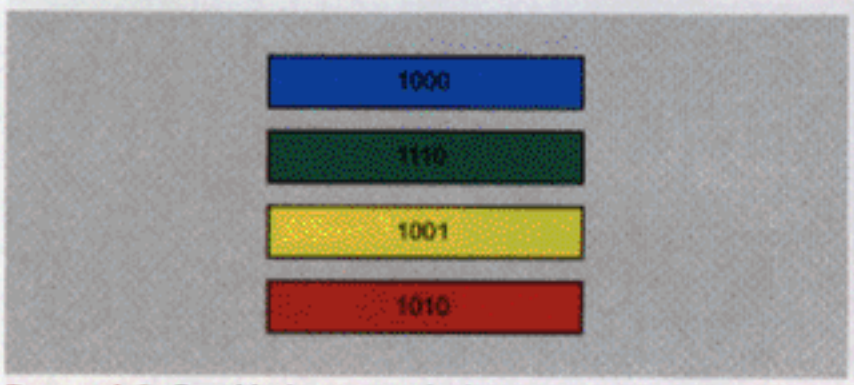
niowi jego przystosowania. W naszym przypadku oznacza to, że osobnik z większą ilością jedynek w chromosomie dostaje większy fragment koła, czyli ma większe szanse na wybór, a co za tym idzie na rozmnażanie się.

Mutacja

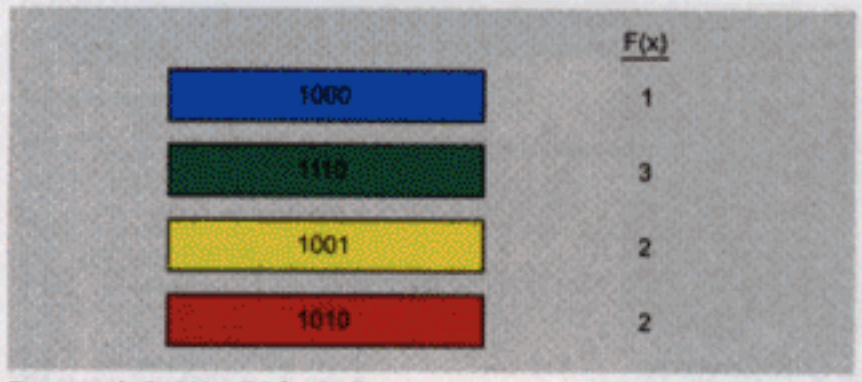
Kolejnym mechanizmem ewolucyjnym, który musi się znaleźć w naszym algorytmie, jest mutacja. Podobnie jak w naturze, mutacja polega na zmianie losowo wybranego genu i z matematycznego punktu widzenia stanowi zabezpieczenie algorytmu przed osiągnięciem ekstremum lokalnego. W naturze ekstremum lokalne odpowiada wynikom krzyżowania osobników zbyt blisko spokrewnionych. Operacja mutacji zapobiega wpadnięciu algorytmu w swoistą pętlę bez wyjścia i będziemy ją realizować za pomocą operatora negacji logicznej NOT.

Krzyżowanie

Kolejnym etapem algorytmu genetycznego jest operacja krzyżowania, w wyniku której z dwojga rodziców powstaje dwoje potomków, co widzimy na Rysunku 6. Na początku losowo wybieramy punkt krzyżowania, oznaczony na rysunku linią przerywaną (w zależności od implementacji takich punktów może też być kilka). Chromosomy rodziców są następnie rozdzielane po linii podziału i ze złożenia powstałych w ten sposób części powstają chromosomy potomków. Tworzony przez nas algorytm zalicza się do grupy algorytmów bez pamięci, tak więc następnym etapem



Rysunek 2. *Przykładowa populacja*



Rysunek 3. *Wyniki funkcji przystosowania*

działania jest zastąpienie populacji rodziców populacją potomków.

Struktury danych

Kod zamieszczony na Listingu 1 zawiera podstawowe dane, jakie wykorzystywane są w programie. Na początku określamy liczebność populacji `max_pop` i maksymalną długość chromosomu `max_gen`, a następnie deklarujemy i inicjujemy zmienną `pop` odpowiedzialną za zliczanie populacji. Potrzebujemy jeszcze zliczać osobników w bieżącej populacji, do czego posłuży nam zmienna `how_m_individuals`.

Pora na zdefiniowanie struktury danych `individual`, odpowiadającej pojedynczemu osobnikowi. W naszym przypadku struktura składa się z łańcucha tekstowego przechowującego chromosom danego osobnika (sam chromosom to 20 znaków, ale musimy zadeklarować łańcuch o długości 21 znaków, by przechować dodatkowy znak końcowy), wartości funkcji przystosowania dla osobnika (zmienna `strength`) oraz dwóch zmiennych pomocniczych.

Populacja początkowa

Procedura odpowiedzialna za wygenerowanie populacji startowej dla algorytmu jest zamieszczona na Listingu 2. Przy wywołaniu procedury przekazujemy do niej dwa parametry: wskaźnik do tablicy zawierającej populację oraz wielkość populacji. Dla każdego osobnika populacji inicjalizujemy pola struktury wartościami zerowymi, a następnie przechodzimy po poszczególnych genach i wpisujemy do nich losowo wybraną wartość 0 lub 1. Na końcu do zmiennej opisującej siłę osobnika wpisywana jest wartość licznika jedynek, odpowiadającego stosowanej w innych częściach algorytmu funkcji przystosowania (Listing 3).

Listing 1. *Dane opisujące populację oraz struktura odpowiadająca osobnikowi*

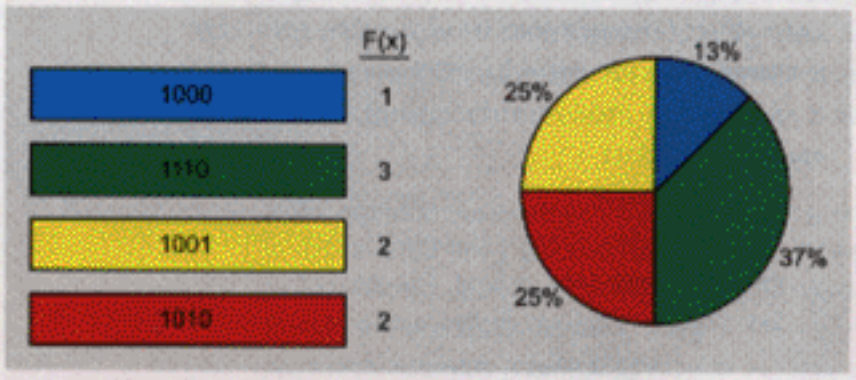
```
const int max_pop=20; //maksymalna liczność populacji
const int max_gen=20; //maksymalna długość chromosomu
int pop=0; //zmienna pop liczy kolejne populacje
unsigned long int how_m_individuals=0; //ilu było osobników

struct individual {
   char chromosome[21]; //max_gen+\0  =20+1
   int  strength; //wartość funkcji dopasowania
   unsigned long int  cargo; //numer chromosomu
   int parent; //ile razy był rodzicem
};
```

Listing 2. *Procedura generująca populację początkową*

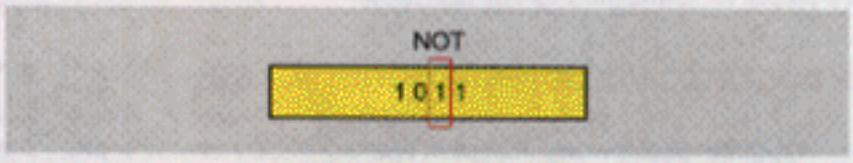
```
void gen_start_population(struct individual *ptr_P,
        int max) {
   char letter[2];
   char *blank="abcdefghijklmnopretu";
   char tmp_dna[21];
   int how=0;
   randomize();
   for(int i=0;i< max ;i++) {
      //czyści profilaktycznie
      strcpy(ptr_P[i].chromosome,blank);
      ptr_P[i].strength=0;
      ptr_P[i].parent=0;
      ptr_P[i].cargo= ++how_m_individuals;
      how=0;
      strcpy(tmp_dna,"");
      for(int j=0;j< max_gen ;j++) {
         if (random(100)<=50)
            strcpy(letter,"0");
         else {
            strcpy(letter,"1");
            how++;
            }
         strcat(tmp_dna, letter);
      }
      strcpy(ptr_P[i].chromosome,tmp_dna);
      ptr_P[i].strength=how;
   }
```

Selekcja osobników

Do implementacji mechanizmu selekcji wykorzystamy wspomnianą już zasadę ruletki, ale nie w postaci kołowej, tylko liniowej. Ideę takiego rozwiązania przedstawia Rysunek 8. Na osi odłożone są przedziały odpowiadające poszczególnym chromosomom, posortowane rosnąco według wartości funkcji przystosowania, a my losujemy jeden punkt na osi. Dzięki temu zachowana jest zasada wyboru silniejszego, gdyż lepiej przystosowany osobnik ma większą szansę na wybór (odpowiada mu większy przedział na osi). Podejście to przedstawia Listing 5. Mnożenie przez 100 obliczonej sumy przedziałów ma na celu jedynie polepszenie własności generatora pseudolosowego i w przypadku użycia dobrych generatorów może zostać pominięte. Funkcja `qsort` jest implementacją algorytmu sortującego Quicksort.



Rysunek 4. *Selekcja metodą koła ruletki*



Rysunek 5. *Operacja mutacji genu*

Mutacja

Funkcja odpowiedzialna za mutację genu w chromosomie losowo wybiera osobnika do zmutowania z podanym przez nas prawdopodobieństwem mutacji. W procesie mutacji pomocniczą rolę odgrywa funkcja `bingo`, pokazana na Listingu 7. Pozwala ona w oparciu o prawdopodobieństwo mutacji i liczność populacji określić, czy jakikolwiek gen zostanie poddany mutacji, a jeśli tak, to określić jego współrzędne. Na podstawie tych współrzędnych funkcja `mutation` znajduje odpowiedni gen i zmienia jego wartość na przeciwną. Po zakończeniu mutacji uaktualniamy wartość funkcji dostosowania osobnika (zmienna `strength`).

Krzyżowanie

Listing 8 przedstawia mechanizm krzyżowania osobników. Do procedury krzyżującej przekazujemy wskaźniki do par rodziców i potomków oraz ilość punktów krzyżowania. W naszej przykładowej implementacji zignorujemy parametr `pc` określający prawdopodobieństwo krzyżowania, co w praktyce oznacza, że uzyskanie potomstwa z każdej pary jest pewne. Losujemy punkty krzyżowania w ilości podanej do procedury w parametrze `points`, a dla każdego kolejnego punktu sprawdzamy, czy nie pokrywa się on z punktem wcześniej wylosowanym. Samo krzyżowanie polega na naprzemiennym złożeniu chromosomów rodziców zgodnie z posortowanymi rosnąco punktami krzyżowania. Gdy powstaną już chromosomy potomków, to nowe osobniki otrzymują kolejne numery w populacji.

Kompletny program

Mamy już wszystkie niezbędne struktury danych i funkcje składowe, pora więc na złożenie programu w całość. Ciało programu przedstawia Listing 2, zawierający kod funkcji uruchamianej w chwili naciśnięcia przycisku *Start*. Pierwszym istotnym krokiem jest podanie prawdopodobieństw wystąpienia krzyżowania i mutacji – w naszym przypadku wynoszą one odpowiednio 0,3 i 0,003, co odpowiada prawdopodobieństwom występującym w naturze. Operujemy na małych populacjach, więc w dalszych etapach programu prawdopodobieństwa te są nieco większe, aby efekt mutacji był lepiej widoczny.

Pora na zainicjowanie populacji, co czynimy deklarując cztery tablice osobników (czyli struktur `individual`).

Listing 3. *Funkcja przystosowania*

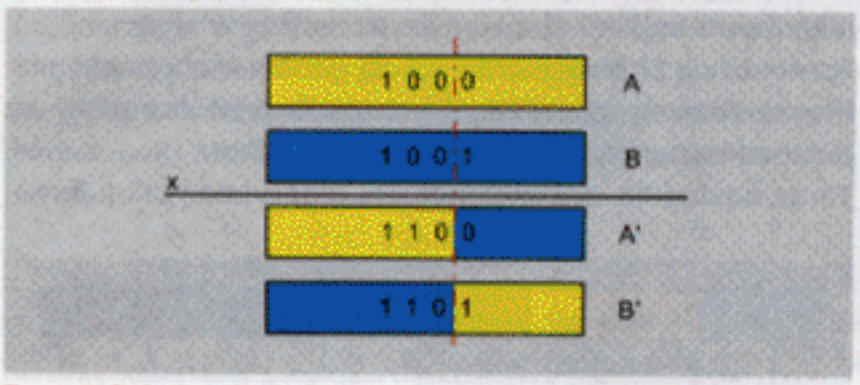
```
int aim_function(char *ptr_a,int chr ) {
   int k;
   int how=0;
   for(k=0;k<chr;k++)
         if (ptr_a[k]=='1') how++;
   return(how);
```

Listing 4. *Funkcja selekcji osobników*

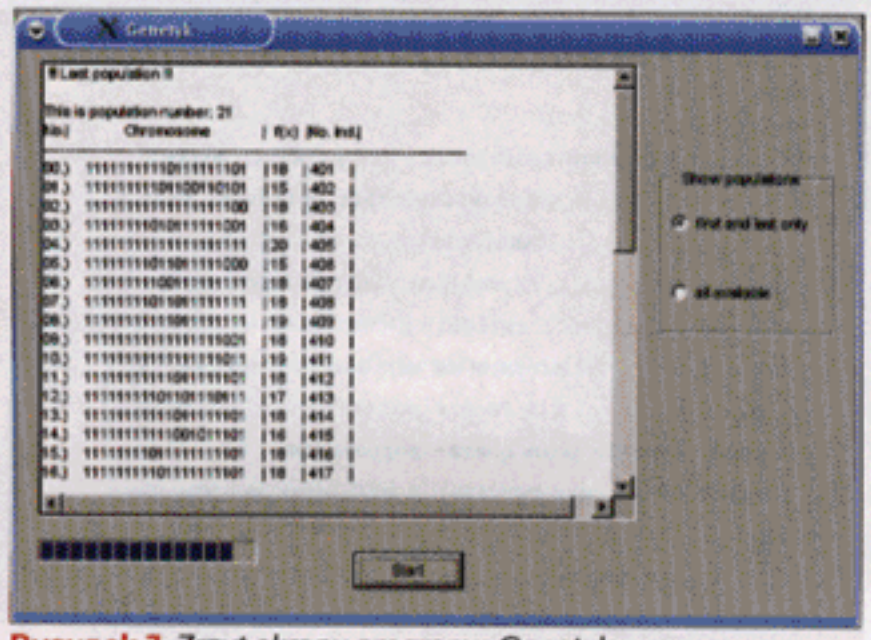
```
struct individual selection(struct individual *ptr_P) {
   struct individual score;
   int sum=0; //wyniki funkcji dopasowania
   int chance;
   //jakby sie nie udało to będzie widać od razu
   strcpy(score.chromosome,"cogito ergo sum");
   score.strength=999;
   score.cargo=999;
   score.parent=999;
   for(int i=0;i<max_pop;i++)
      sum=sum+ ptr_P[i].strength;
   sum=sum*100;
   // Quicksort z parametrami:
   // tablica, ilość elementów, rozmiar elementu, ←
      funkcja porównująca
   qsort(ptr_P,max_pop, sizeof(struct individual),comp_
                        strength);
   chance =random(sum);
   sum=0;
   int i;
   for(i=0;i<max_pop;i++) {
      sum=sum+ptr_P[i].strength;
      if(chance<=sum*100) { // trafienie w srutary przedział
         ptr_P[i].parent++;
         score= ptr_P[i];
         break;
      }
   }
   return(score);
}
```

Będziemy ich używać do przechowywania: populacji rodziców, populacji dzieci, dwojga rodziców i dwojga potomków. Następnie inicjujemy generator liczb pseudolosowych, generujemy losową populację początkową (funkcja `gen_start_population`) oraz wyliczamy i zapisujemy wartość funkcji przystosowania dla każdego osobnika w tej populacji za pomocą procedury `estimate_population`. Procedurę wywołujemy raz dla każdego osobnika, podając jako parametr jego chromosom.

Na Rysunku 7 widzimy efekt działania funkcji `show_population` wypisującej populację podaną jako pierwszy parametr (w tym wypadku populację rodziców `P`). Drugi parametr podawany do funkcji określa ilu osobników ma zostać wyświetlonych, przy czym zero oznacza pokazanie całej popu-



Rysunek 6. *Operacja krzyżowania chromosomów*



Rysunek 7. *Zrzut ekranu programu Genetyk*

lacji. Ciało właściwego algorytmu ewolucyjnego umieszczone jest w pętli `while`, której warunkiem jest wynik wywołania funkcji testującej `end_condition`. Funkcja ta zwraca prawdę, jeśli przynajmniej w jednym chromosomie ilość jedynek jest zgodna z założoną – w naszym przypadku jest to 20 (czyli wszystkie geny). Działanie algorytmu rozpoczynamy od zwiększenia licznika populacji i wyświetlenia jego wartości na pasku stanu, po czym przechodzimy do fazy selekcji osobników. Najpierw wybierany jest jeden rodzic, a potem drugi, po czym następuje sprawdzenie, czy przypadkiem nie wybrano tego samego osobnika dwukrotnie. Jest to zabezpieczenie zgodne z założeniem, że osobnik nie może być hermafrodytą, czyli mieć potomków z samym sobą, co z kolei odpowiada mechanizmowi rozmnażania większości organizmów występujących w przyrodzie. Oczywiście możemy łatwo zmienić to założenie poprzez wykomentowanie odpowiedniego fragmentu kodu – zachęcam do zbadania uzyskanych w ten sposób efektów.

Listing 5. *Funkcja dokonująca mutacji chromosomu*

```
int mutation(struct individual *ptr_P, float pm, ←
         int gen, int chr ) {
   int how=0,w,k;
   struct coordinates coords[max_pop];
   how= bingo(coords, pm, gen, chr);
   for(int i=0;i<how;i++) {
      w=coords[i].w;
      k=coords[i].k;
      //to się nie powinno zdarzyć
      if (k>max_gen) k=max_gen;
      if (ptr_P[w].chromosome[k]=='1') {
         ptr_P[w].chromosome[k]='0';
         ptr_P[w].strength=ptr_P[w].strength-1;
      }
      else if (ptr_P[w].chromosome[k]=='0') {
            ptr_P[w].chromosome[k]='1';
            ptr_P[w].strength=ptr_P[w].strength+1;
      }
   }
   return(how);
}
```

Listing 7. *Procedura dokonująca krzyżowania osobników*

```
void crossing(struct individual *ptr_parents, struct
                    individual *ptr_children, float pc,
                    int points) {
  int t_pc[max_gen]; // tablica punktów krzyżowania
  char child0[21]="", child1[21]="";
  int offset=2; // miejsce na min i max, czyli
  t_pc[1]=0;
  t_pc[2]=max_gen;
  for(int i=offset;i<points+offset;i++) {
    t_pc[i]=random(max_gen);
    //zapewnienie unikalności punktów krzyżowania
    for(int j=0;j<i;j++) {
      //wymuś ponowne losowanie tego punktu
      if (t_pc[j]==t_pc[i]) i--;
    }
  }
  // Quicksort z parametrami:
  // tablica, ilość elementów, rozmiar elementu,
    funkcja porównująca
  qsort(t_pc,punkty+offset, sizeof(int),comp);

  for(int i=1;i<points+offset;i++) {
    div_t even;
    even= div(i,2);
    String s_temp;
    if(even.rem==0) { //parzysty
      FormGenetyk->Edit->Text= ptr_parents[0].chromosome;
      s_temp=FormGenetyk->Edit->Text.SubString(
          t_pc[i-1],t_pc[i]-t_pc[i-1]);
      strcat(child1,s_temp.c_str());
      FormGenetyk->Edit->Text= ptr_parents[1].chromosome;
    }
    else {
      FormGenetyk->Edit->Text= ptr_parents[0].chromosome;
      s_temp=FormGenetyk->Edit->Text.SubString(
          t_pc[i-1],t_pc[i]-t_pc[i-1]);
      strcat(child0,s_temp.c_str());
      FormGenetyk->Edit->Text= ptr_parents[1].chromosome;
      s_temp=FormGenetyk->Edit->Text.SubString(
          t_pc[i-1],t_pc[i]-t_pc[i-1]);
      strcat(child1,s_temp.c_str());
    }
  }
  strcpy(ptr_children[0].chromosome, child0);
  strcpy(ptr_children[1].chromosome, child1);
  //numer genu
  ptr_children[0].cargo= ++how_m_individuals;
  ptr_children[1].cargo= ++how_m_individuals;
  //ile razy był rodzicem
  ptr_children[0].parent=0;
  ptr_children[1].parent=0;
```

Listing 6. *Funkcja wyznaczająca współrzędne genu*

```
int bingo(struct coordinates *ptr_coords,
      float frequency, int gen, int chr) {
  int number= gen *chr; //ilość genów * ilość osobników
  int how, chance, w, k;
  how= INT(frequency*number);

  for(int i=0;i<how;i++) {
    chance =random(number);
    bool cut=false;
    for(w=1;w<gen;w++) {
      for(k=2;k<chr;k++) {
        if (k+w*chr==chance) {
          cut=true;
          break;
        }
      }
    if (cut) break;
    }
    ptr_coords[i].w=w;
    ptr_coords[i].k=k;
  }
  return(how);
}
```

Gdy rodzice są już wybrani, możemy przystąpić do krzyżowania wybranej pary. Najpierw losowo generujemy ilość punktów krzyżowania i zapamiętujemy ją w zmiennej `how_m_pc`, a następnie wywołujemy procedurę `crossing`, która odpowiada za proces krzyżowania. Jako parametry podajemy populację źródłową, populację wynikową, prawdopodobieństwo krzyżowania (wcześniej zapisane w zmiennej `pc`) oraz ilość punktów krzyżowania. W opisywanej implementacji nie korzystamy z wyliczonej wcześniej losowej ilości zapisanej w `how_m_pc`, tylko podajemy eksperymentalnie dobraną ilość punktów krzyżowania, w naszym przypadku wynoszącą trzy. Dwoje utworzonych w ten sposób potomków dodajemy do populacji wynikowej.

Po stworzeniu populacji potomków poddajemy wchodzące w jej skład osobniki mutacji, wywołując funkcję `mutation` z czterema parametrami: populacją, prawdopodobieństwem mutacji (zapisanym wcześniej w zmiennej `pm`), liczebnością populacji oraz ilością genów w chromosomie. Przystosowanie poszczególnych osobników oceniamy za pomocą wspominanej już procedury `estimate_population`. Teraz następuje ciekawy moment algorytmu. Jak już mó-

Rysunek 8. *Zmodyfikowana metoda koła ruletki*

Listing 8. *Ciało główne programu*

```
void fastcall TFormGenetyk::ButtonOKClick(TObject *Sender) {
  //zerowanie ilości osobników na początku próby
  how_m_individuals=0;
  //liczność populacji startowej
  const int start_pop=max_pop;
  float pc=0.3; //prawdopodobieństwo krzyżowania
  float pm=0.003; //prawdopodobieństwo mutacji
  int how_m_pc=2; //ile wylosować punktów krzyżowania
  struct individual P[start_pop]; //deklaracja populacji P
  struct individual C[start_pop]; //deklaracja populacji C
  //używane przy krzyżowaniu
  struct individual parents[2],children[2];
  randomize();
  entry++; //ile razy startowano
  FormGenetyk->Memo->Clear(); //czyści memo
  //utworzenie populacji początkowej P
  gen_start_population(P, max_pop);
  //ocena wartości funkcji dopasowania osobników ←
    nowej populacji P
  estimate_population(P, max_pop, max_gen);
  show_population(P,0);

  while (end_condition(P)) {
    pop++;
    FormGenetyk->Progress->StepIt(); //pasek stanu
    //selekcja rodziców do krzyżowania
    int i;
    for(i=0;i<max_pop;i=i+2) {
      //selekcja zmodyfikowaną metodą ruletki
      parents[0]= selection(P);
      int cr_sel=0;
      do {   //żeby nie był hermafrodytą
        parents[1]= selection(P);
        cr_sel++;
        FormGenetyk->Edit1->Text= i2s(cr_sel);
        }
      while(parents[0].cargo==parents[1].cargo);
    //operacja krzyżowania
      // losowy wybór ilości punktów krzyżowania
      do how_m_pc= random(max_pop/2);
        while(how_m_pc==0);
        // zamiast 3 można wstawić how_m_pc
        crossing(parents,children, pc, 3);
        C[i]= children[0];
        C[i+1]= children[1];
      }// koniec tworzenia populacji dzieci
      //mutacja w populacji C
      mutation(C, pm, max_pop, max_gen);
      //ocena wartości funkcji dopasowania osobników ←
        nowej populacji C
      estimate_population(C, max_pop, max_gen);

      //populacje rodziców P zastępuje populacja dzieci C
      for(int i=0;i<max_pop;i++) {
        P[i]=C[i];
        strcpy(C[i].chromosome,"");
        C[i].strength=0;
        C[i].cargo=0;
      }
      //jeżeli użytkownik chce wszystkie
      if (FormGenetyk->RG->ItemIndex==1){
        //ok. 26 populacji przepełnia Memo
        show_population(P,0);
      }
      //przepełnione Memo
      if (FormGenetyk->RG->ItemIndex==1 && pop>=26) {
        FormGenetyk->Memo->Clear();
        FormGenetyk->Memo->Lines->Insert(0," !!!Memo ←
          PRZEPEŁNIONE - zbyt wiele populacji aby ←
          pokazać!!!");
        show_population(P,0);
        end_condition(P);
      }
      //jeżeli pierwsza i ostatnia
      if (FormGenetyk->RG->ItemIndex==0)
        show_population(P,0);
      FormGenetyk->Memo->Lines->Insert(0," !!! Ostatnia ←
          populacja !!!");
      pop=1;
```

wiliśmy, nasz algorytm genetyczny należy do grupy klasycznych algorytmów bez pamięci, co w praktyce oznacza, że populacja rodziców jest nadpisywana przez populację potomków. W ten sposób następuje bezpowrotna strata populacji rodziców, ale niektóre cechy rodziców przetrwają w ich potomstwie, co odzwierciedla naturalny mechanizm dziedziczenia. Ostatnia część kodu to już tylko nieco zawiłe operacje związane z wyświetlaniem wyników w polu *Memo* i zabezpieczeniem się przed przepełnienia tego pola.

Podsumowanie

Algorytmy genetyczne sprawdzają się najlepiej w zadaniach, które mają wiele możliwych (poprawnych) rozwiązań, a problem polega na wyłonieniu spośród nich rozwiązania optymalnego lub bliskiego optymalnemu. Stworzony przez nas mechanizm ewolucyjny można łatwo rozbudować tak, by rozwiązywał konkretny, zadany problem. W tym celu każdy osobnik musi odpowiadać jednemu możliwemu rozwiązaniu z dziedziny problemu, a funkcję przystosowania trzeba tak sformułować, by zwracała wartość odpowiadającą sile danego rozwiązania w kontekście badanego zadania. Zastosowanie algorytmów genetycznych do optymalizacji funkcji oceniającej programu szachowego przedstawia w swoim artykule Tomasz Michniewski – w tym przypadku osobnikami są różne zestawy parametrów funkcji oceniającej, a siłę osobników określają wyniki rozegranych między nimi partii szachów. ■