Oprogramowanie `gabi`

dr inż. Jarosław Arabas
Instytut Systemów Elektronicznych
Politechnika Warszawska
ul. Nowowiejska 15/19, 00-665 Warszawa
email: `jarabas@ipe.pw.edu.pl`
pok. 231, tel. 660 77 31, fax 825 23 00

1 Założenia ogólne

Oprogramowanie `gabi` , załączone do niniejszej książki, ma na celu ilustrację podstawowych elementów algorytmów ewolucyjnych omawianych w tekście. Większość wyników eksperymentów powstała przy jego użyciu — wyjątek stanowią prace cytowane.

Autor stawia sobie za cel udostępnić zestaw funkcji i procedur, które mogą służyć do szybkiego podjęcia przez Czytelnika samodzielnego eksperymentowania z algorytmami ewolucyjnymi. W przypadku, gdy książka miałaby służyć jako podręcznik, oprogramowanie może być wykorzystane jako rama do zajęć laboratoryjnych lub projektowych. Powyższe względy zadecydowały o wyborze języka programowania. Jest nim ANSI C ze względu na jego stosunkowo powszechną znajomość i dostępność systemów programowania. Interfejs użytkownika, ze względu na przenośność, jest celowo bardzo ubogi i posługuje się tekstami i liczbami przekazywanymi za pomocą standardowych strumieni oraz plików konfiguracyjnych. Założenie o braku interakcyjności jest podyktowane wygodą wielokrotnej symulacji i ułatwia prowadzenie zbieranie statystyk, zrównoleglanie i automatyczne przetwarzanie wyników eksperymentów.

2 Opis funkcjonalny oprogramowania

Na informację o osobniku składa się genotyp (w postaci tablicy wartości zmiennych niezależnych), tablicę parametrów (wykorzystywanych do samoczynnej adaptacji), wartość funkcji przystosowania i wiek osobnika (mierzony liczbą sukcesji, które przetrwał nienaruszony). Osobniki są elementami populacji — bazowej i potomnej. Każdy osobnik podlega ocenie podczas prezentacji środowisku — zostaje wówczas określona jego wartość funkcji przystosowania. Wartość ta jest obliczana dokładnie raz — w momencie utworzenia nowego osobnika i nie podlega ponownemu przeliczeniu nawet wówczas, gdy jest on przetrzymywany w populacji bazowej dłużej niż jedną generację. Założenie takie pozwala przy statycznym środowisku (w którym funkcja przystosowania jest niezmienna w czasie) uniknąć zbędnych obliczeń, jednak w przypadku środowisk dynamicznych (o funkcji przystosowania zmiennej w czasie) należałoby od niego odstąpić.

Środowisko, w jakim odbywa się ewolucja, jest określone przez funkcję przystosowania. W obecnej wersji oprogramowania `gabi` jest ona tworzona na podstawie zadania optymalizacji z funkcją celu i ograniczeniami; w przypadku obecności tych ostatnich wymagają jeszcze zdefiniowania metody uwzględniania ograniczeń. Elementem tego środowiska jest metoda próbkowania obszaru zainteresowania, wykorzystana do inicjacji genotypów osobników populacji bazowej.

Oprogramowanie `gabi` dopuszcza wykorzystanie wielu typów operatorów genetycznych w zależności od specyfiki rozwiązywanego zagadnienia. Operatory te są wybierane losowo z puli dostępnych operatorów — o wyborze decyduje zmienna losowa o znanym, podanym przez użytkownika rozkładzie. Stwarza to możliwość adaptowania rozkładu wraz z przebiegiem ewolucji.

Nad przebiegiem ewolucji czuwają dwa monitory: populacji i zadania. Pierwszy z nich ma za zadanie prowadzić bieżące statystyki populacji bazowej, takie jak informacje o najgorszym, najlepszym, przeciętnym osobniku, o wieku populacji itp. Są one wykorzystywane przede wszystkim przez niektóre metody reprodukcji (np. reprodukcję proporcjonalną), mogą także posłużyć do określenia kryterium zatrzymania (np. bazującego na zaniku różnorodności populacji) czy do adaptacji parametrów (np. zasięgu mutacji). Z kolei monitor zadania jest jakby zewnętrznym „obserwatorem" algorytmu ewolucyjnego — nie wnika w głąb populacji, lecz obserwuje najlepszego znalezionego dotychczas osobnika, czyli bieżące oszacowanie rozwiązania.

Monitor ten służy przede wszystkim do określenia spełnienia prostyc, zewnętrznych kryteriów zatrzymania, bazujących na szybkości zmian rozwiązania czy też na przekroczeniu limitu obliczeń wartości funkcji przystosowania.

Program rozpoczyna swoje działanie od wczytania ustawień z plików konfiguracyjnych. Określane są w nich metody reprodukcji i sukcesji, sposób reprezentacji osobnika, rodzaje wykorzystywanych operatorów genetycznych, kryterium zatrzymania oraz wartości parametrów niezbędnych do pełnego zdefiniowania algorytmu ewolucyjnego. Środowisko, szczególne operatory genetyczne (jeśli jest potrzeba ich użycia) oraz inne elementy nie mieszczące się w „standardzie" algorytmów ewolucyjnych ogólnego przeznaczenia wymagają zdefiniowania w postaci fragmentów oprogramowania, podlegających kompilacji i konsolidacji z dostępnym kodem.

Po zdefiniowaniu algorytmu i zadania, program przechodzi do zainicjowania populacji bazowej i obliczenia wartości funkcji przystosowania dla jej elementów. Następnie wykonywana jest pętla główna algorytmu z cyklem reprodukcji, operacji genetycznych i sukcesji. Jest ona powtarzana tak długo, dopóki nie zostanie spełnione kryterium zatrzymania. Podczas działania pętli głównej wyprowadzane są na standardowe wyjście wartości dostępne poprzez monitory populacji i zadania. Format i ilość wyprowadzanej informacji są określane przez użytkownika w odpowiednim pliku konfiguracyjnym. Błędy techniczne oraz niespełnienie wewnętrznych wymagań algorytmu ewolucyjnego (takich jak próba ustawienia wartości prawdopodobieństwa powyżej jedności) są sygnalizowane na standardowym wyjściu błędów. Oprogramowanie `gabi` nie wykorzystuje standardowego strumienia wejściowego.

3 Sposób implementacji

Większość informacji jest umieszczona w strukturach (struct). Są one wskazywane przez odpowiednie wskaźniki. Przyjęto konwencję nazewnictwa, w której wskaźnik do struktury nosi nazwę odpowiadającą w jak największym stopniu reprezentowanemu przez nią obiektowi, natomiast sama struktura ma taką samą nazwę, zakończoną literą 'S'.

Struktury są wykorzystywane na dwa sposoby. Część z nich zawiera jedynie informacje o charakterze konfiguracyjnym — są to `ALGORYTMS, LOSOWES, OGRANICZENIAS, RAPORTYS, SELEKCJAS, STOPS, ZADANIES`. Wartości pól powyższych struktur są nadawane na podstawie zawartości plików konfiguracyjnych w części inicjującej ustawienia. W czasie działania algorytmu ewolucyjnego, struktury te są wykorzystywane wyłącznie do odczytu. W programie występuje dokładnie jedna kopia każdej z nich, wskazywana przez zmienną globalną.

Inaczej wygląda sytuacja w przypadku struktur `MONPOP, MONZAD, OSOBNIKS, POPS`, które służą do przechowywania informacji przetwarzanych podczas działania algorytmu ewolucyjnego.

Struktura `OPERATORYS` ma w obecnej wersji oprogramowania charakter konfiguracyjny, lecz stwarza możliwość wykonania adaptacji prawdopodobieństw wykonania operatorów.

3.1 Struktury konfiguracyjne

- `ALGORYTMS`

unsigned nPop	liczność populacji bazowej
unsigned Samoadaptacja	wartość 1 powoduje włączenie samoczynnej adaptacji zasięgu mutacji, zaś 0 jej wyłączenie

- `LOSOWES`

unsigned tJednostajny	numer funkcji implementującej rozkład jednostajny
unsigned tNormalny	numer funkcji implementującej rozkład normalny
unsigned tInicjacji	sposób inicjacji ziarna generatora liczb pseudolosowych: 1– na podstawie czasu systemowego, 2–na podstawie podanej wartości ziarna
unsigned InitZiarno	wartość ziarna przy inicjacji w trybie 2

- `MONPOP`

unsigned Maksimum	numer osobnika będącego maksimum w populacji
unsigned Minimum	jw., dotyczy minimum
OSOBNIK Sredni	osobnik średni — nie istnieje w populacji, lecz jest tworzony poprzez uśrednianie
double Zroznicowanie	wartość zróżnicowania populacji

- `MONZAD`

OSOBNIK Rozwiazanie	kopia najlepszego osobnika znalezionego w całym uruchomieniu algorytmu
double Poprawa	różnica przystosowania obliczana gdy został znaleziony nowy najlepszy osobnik
unsigned LPrzystosowan	liczba obliczeń wartości funkcji przystosowania
unsigned Generacja	aktualny numer generacji algorytmu
double *OkienkoGen	tablica przechowująca najlepsze dotychczas znalezione rozwiązania w ostatnich `SzerStop` generacjach
double *OkienkoPrzyst	jw, przechowywane są najlepsze dotychczas znalezione wartości z ostatnich `SzerStop` obliczeń przystosowania

- `OGRANICZENIAS`

unsigned tOgraniczen	sposób uwzględniania ograniczeń
unsigned tFKary	typ funkcji kary
double Lamarkizm	prawdopodobieństwo ewolucji lamarkowskiej w ograniczeniach z naprawą
double WagaOgr	waga z jaką funkcje kary wchodzą do przystosowania (funkcja celu z wagą 1)

- `OPERATORYS`

unsigned nOperatorowC	liczba operatorów typu 1
unsigned *tOperatoraC	numery operatorów typu 1
double *PstwoC	prawdopodobieństwa operatorów typu 1
unsigned *nRodzicow	liczba rodziców operatorów typu 1
unsigned *nDzieci	liczba potomków operatorów typu 1
unsigned nOperatorowM	liczba operatorów typu 2
unsigned *tOperatoraM	numery operatorów typu 2
double *PstwoM	prawdopodobieństwa operatorów typu 2
unsigned nPKrzyz	liczba punktów krzyżowania n-punktowego
double TAU	wartość τ dla mutacji samoadaptacyjnej
double TAUP	wartość τ' dla mutacji samoadaptacyjnej
double MinSigma	minimalna wartość σ dla mutacji samoadaptacyjnej lub wartość σ dla przypadu bez samoadaptacji
double pMut	minimalna wartość prawdopodobieństwa mutacji bitu dla mutacji samoadaptacyjnej lub jego wartość dla przypadu bez samoadaptacji
unsigned NakladanieOp	wartość różna od zera włącza działanie operatorów typu 2, równa zeru — wyłącza je

- `OSOBNIKS`

unsigned nGenow	liczba genów chromosomu
TGEN* Genotyp	genotyp osobnika
unsigned nPar	liczba parametrów kodowanych w osobniku (w celu samoadaptacji)
double* Parametry	tablica parametrów do samoadaptacji
double Przystosowanie	wartość funkcji przystosowania osobnika
unsigned MaxWiek	maksymalny dopuszczalny wiek osobnika
unsigned Wiek	rzeczywisty wiek osobnika
unsigned LKopii	liczba kopii, które usyskał osobnik

- `POPS`

unsigned nOsobnik	liczba osobników w populacji
OSOBNIK* Osobniki	tablica osobników populacji
MONPOP* Monitor	monitor populacji

- `RAPORTYS`

char *RapGen	Format wyprowadzania informacji o genie (wg konwencji używanej przez funkcję `printf`, z pominięciem znaków cudzysłowu, np. %f oznacza wyprowadzenie liczby zmiennoprzecinkowej)
char* RapPop	Format raportu stanu populacji w postaci łańcucha znakowego. Dopuszczalne litery: 'o' – Osobniki, 'z' – Zroznicowanie, 'n' – miNimum, 's' – Srednia, 'x' – maXimum.
char *RapOs	Format raportu stanu populacji w postaci łańcucha znakowego. Dopuszczalne litery: 'x' - zmienne niezalezne X, 'p' - Parametry, 'w' - Wiek, 'k' - liczba Kopii, 'f' - wartosc Funkcji przystosowania.
char *RapSredni	Format raportu osobnika średniego. Dopuszczalne litery jak w raporcie osobnika.
char *RapMin	Format raportu osobnika minimum. Dopuszczalne litery jak w raporcie osobnika.
char *RapMax	Format raportu osobnika maksimum. Dopuszczalne litery jak w raporcie osobnika.
char *RapZad	Format raportu zadania. Dopuszczalne litery: 't' - numer bieżącej generacji, 'r' - osobnik będący Rozwiązaniem.
char *RapRozw	Format raportu osobnika będącego rozwiązaniem zadania. Dopuszczalne litery jak w raporcie osobnika.

Łańcuchy zawierające formaty raportów nie mogą zawierać białych znaków. Zamiast nich, stosuje się znaki: '~', '\\', '\t','\n', które powodują wyprowadzenie w raporcie znaków ' ','\',tabulacji i przejścia do nowego wiersza.

- `SELEKCJAS`

unsigned tReprodukcji	wybór metody reprodukcji
unsigned tSukcesji	wybór metody sukcesji
unsigned nTurnieju	wielkość turnieju
double nElity	liczba oosbników stanowiących elitę
unsigned tRang	typ funkcji rangowej
double obciecie	ułamek osobników reprodukowanych w reprodukcji z obcięciem
double fRWykladnik	wykładnik wykładniczej funkcji rangowej
unsigned maxWiek	maksymalny wiek w zmodyfikowanej sukcesji elitarnej

- `STOPS`

unsigned tStopu	typ krytrium zatrzymania
unsigned SzerStop	liczba generacji po której musi nastąpić poprawa przystosowania
double WysStop	minimalna różnica pomiędzy przystosowaniem osobników z końca i pocztku przedziału czasu `SzerStop`
unsigned MaxGen	liczba genracji po której następuje zatrzymanie
unsigned MaxPrzyst	liczba obliczeń wartości funkcji przystosowania po której następuje zatrzymanie
double MinPrzyst	satysfakcjonująca wartość przystosowania

- `ZADANIES`

unsigned nLiter	licza różnych wartości dyskretnych
unsigned nZmiennych	liczba zmiennych niezależnych
FUNKCJA fcelu	wskazanie do funkcji celu
char Minimalizacja	mnożniki funkcji celu (1 – minimalizacja, -1 – maksymalizacja
unsigned nNierown	liczba ograniczeń nierównościowych
FUNKCJA *nierownosciowe	tablica wskazań do funkcji ograniczeń nierównościowych
unsigned nRown	liczba ograniczeń równościowych
FUNKCJA *rownosciowe	tablica wskazań do funkcji ograniczeń równościowych
MONZAD *Monitor	monitor zadania
unsigned saKostkowe	wartość logiczna mówiąca o obecności ograniczeń kostkowych
TGEN *GorneX	tablica ograniczeń kostkowych górnych
TGEN *DolneX	tablica ograniczeń kostkowych dolnych
char* plikDanych	nazwa pliku z danymi (do parametryzacji zadania)
unsigned tInic	sposób tworzenia pierwszej populacji bazowej
double* Rozklad	rozkład prawdopodobieństwa wyboru poszczególnych wartości dyskretnych przy losowej inicjacji

nazwa pliku	nazwa struktury
algorytm.par	ALGORYTMS
losowe.par	LOSOWES
ogranicz.par	OGRANICZENIAS
genetyka.par	OPERATORYS
raporty.par	RAPORTYS
selekcja.par	SELEKCJAS
stop.par	STOPS
zadanie.par	ZADANIES

Tablica 1: Zestawienie plików konfiguracyjnych i odpowiadających im struktur

4 Pliki konfiguracyjne

Każdej ze struktur konfiguracyjnych odpowiada plik, w którym znajdują się wartości nadawane opowiednim polom struktury podczas inicjacji programu. Zawartość pliku konfiguracyjnego ma następującą składnię:

<zawartość pliku>::=<zawartość pusta>|<nazwa><wartość>|<określenie tablicy>

<określenie tablicy>::=<nazwa><liczba całkowita bez znaku><nazwa tablicy>{<wartość>}

<nazwa tablicy>::=['a'-'z']['a'-'z']*

<nazwa>::=['a'-'z']['a'-'z']*

<wartość>::=<liczba całkowita bez znaku>|<liczba rzeczywista>|<łańcuch znakowy>

Każdy z wymienionych wyżej atomów musi być otoczony białymi znakami.

Pola struktur konfiguracyjnych mogą zawierać pojedyncze wartości (liczby lub łańcuch znakowy) lub ich tablice. Stąd koniecznosc uwzględnienia tych dwóch przypadków w składni zawartości pliku.

Para <nazwa><wartość> służy do inicjacji takiego pola, którego wartość jest liczbą. <nazwa> musi być nazwą tego pola (pisaną wyłącznie *małymi literami*), którego <wartość> będzie nadana. Należy zwrócić szczególną uwagę na zgodność typów. Podczas wczytywania zawartości plików konfiguracyjnych nie jest prowadzona kontrola zgodności typów. Nazwy nie mające odpowiednika w odpowiedniej strukturze konfiguracyjnej będą pomijane bez zgłoszenia błędu ani ostrzeżenia.

Inicjowanie pola tablicowego wymaga więcej wysiłku. Wszystkie tablice w strukturach konfiguracyjnych są tworzone dynamicznie, zaś informacja o wielkości tablicy jest przechowywana w oddzielnym polu struktury. Dlatego kolejność tworzenia tablicy jest następująca. Na początku należy zainicjować pole określające wielkość tablicy. Wykonuje się to podając parę <nazwa><wartość>. Następnie, podaje się <nazwę tablicy>, a po niej wartości tablicy w liczbie dokładnie odpowiadającej jej rozmiarom (należy pamiętać o zgodności typów danych z oczekiwanymi w tablicy). Oprogramowanie nie jest odporne na błędnie wprowadzone dane.

W tablicy 1 podane są nazwy plików konfiguracyjnych i odpowiadających im struktur.

5 Sposób działania oprogramowania

5.1 Funkcja `main()`

Oprogramowanie `gabi` działa w sposób następujący. Wykonanie rozpoczyna się od wywołania funkcji `okreslParametry();`. Jej zadaniem jest wykonanie kolejnych sekwencji polegających na otwarciu pliku konfiguracyjnego, zainicjowaniu odpowiedniej struktury konfiguracyjnej i zamknięciu pliku. W wyniku utworzone zostają zmienne globalne zawierające wskaźniki na struktury konfiguracyjne. W pliku `"gabi.log"` zapisane zostają wszystkie rozpoznane i wczytane nazwy pól struktur wraz z ich wartościami.

Następnie zostaje uruchomiona funkcja `initLosowe();`, która dokonuje inicjacji generatora liczb pseudolosowych.

Ostatnią czynnością funkcji `main ()` jest wywołanie funkcji `petlaGlowna();`, która implementuje pętlę główną algorytmu ewolucyjnego.

5.2 Funkcja `petlaGlowna();`

Całość algorytmu ewolucyjnego, zapisana jako funkcja `petlaGlowna();` jest zaimplementowana zgodnie ze standardowym algorytmem ewolucyjnym. Faza początkowa algorytmu ewolucyjnego składa się z inicjacji populacji bazowej (`popInic(Pt);`), po której rozpoczyna się pętla główna algorytmu, wykonywana dopóty, dopóki funkcja `krytStopu(Zadanie)` nie zwróci wartości różnej od zera. W pętli tej wykonywana jest reprodukcja wraz z operatorami genetycznymi (`genetyka(Pt, Ot);`), po której następuje sukcesja (`sukcesja(Pnowa, Pt, Ot);`). Po zainicjowaniu populacji bazowej, jak również każdorazowo po zakończeniu reprodukcji i operacji genetycznych, dokonuje się ewaluacji wszystkich nowo wygenerowanych osobników (`popEwaluacja(Pt);`).

Oprócz czynności wynikających bezpośrednio ze schematu algorytmu ewolucyjnego, dodane są elementy specyficzne dla implementacji. Funkcje `popMonitor(Pt);` i `zadMonitor(t,Pt);` uaktualniają stan monitorów populacji (wynkorzystywanego do reprodukcji) i zadania (wykorzystywanego do kryterium zatrzymania). Stan monitorów populacji i zadania jest wyprowadzany na standardowe wyjście przez funkcje `rapPop(Pt,NULL);` i `rapZadanie(Zadanie,NULL);`.

Populacje bazowa i potomna są zdefiniowane jako zmienne lokalne `Pt` i `Ot`, zarządzaniem których zajmują się funkcje z przedrostkiem `pop`, odpowiadające za utworzenie populacji, wypełnienie jej osobnikami oraz jej opróżnienie i usunięcie.

5.3 Funkcja `char genetyka (POP Pt, POP Ot);`

Funkcja ta realizuje dwie czynności — reprodukcję i operacje genetyczne. Jej definicja pozwala w łatwy sposób wykorzystać zarówno operatory krzyżowania generujące liczbę potomstwa równą liczbie rodziców, jak i operatory nie spełniające tego warunku; podobnie, można wykorzystywać krzyżowanie dwuosobnicze, jak i wieloosobnicze. Istnieje również możliwość wprowadzenia wielu różnych operatorów tego samego typu, a także modyfikowania częstości wykonywania operatorów genetycznych w różnych fazach algorytmu ewolucyjnego.

Zmienna globalna `Operatory` (będąca wskazaniem na strukturę typu `OPERATORYS`) przechowuje informację o rodzajach operatorów genetycznych udostepnianych w systemie. W zależności od ustawienia pola `NakladanieOp` są dwa tryby działania operatorów genetycnzych. W pierwszym trybie, odpowiadających różnej od zera wartości tego pola, występują dwa rodzaje operatorów genetycznych (w domyśle — krzyżowanie i mutacja), przy czym operatory mutacyjne przyjmują tylko jednego rodzica i tworzą jednego potomka, działając na osobnikach będących wynikiem działania operatorów krzyżowania. W drugim trybie (pole ustawione na zero), wszystkie operatory działają rozłącznie. Wybór operatora dokonywany jest w obu trybach losowo, przy czym rozkład prawdopodobieństwa wyboru poszczególnych operatorów jest dany w odpowiedniej tablicy.

Funkcja `char genetyka (POP Pt, POP Ot);` sprowadza się do pętli, która jest wykonywana dopóty, dopóki nie zostanie wygenerowanych przez reprodukcję z `Pt` i operacje genetyczne tyle osobników, aby zapełnić populację potomną `Ot`. W każdym obiegu tej pętli dokonuje się najpierw wyboru operatora genetycznego. Każdy operator ma określoną liczbę rodziców. Wykonywana jest zatem funkcja `reprodukcja(Pt, Rodzice, Operatory->nRodzicow[Operator]);`, która umieszcza w tablicy `Rodzice` numery osobników populacji bazowej `Pt`, którzy zostali wybrani na rodziców. Tworzona jest następnie pomocnicza populacja osobników potomnych (tablica `Dzieci`), zaś gentoypy tych osobników są określane podczas wykonania operatora genetycznego (funkcja `operatorG(Operatory->tOperatoraC[Operator],Pt->Osobniki, Rodzice, Operatory->nRodzicow[Operator], Operatory->nDzieci[Operator], Dzieci);`). Jeśli pole `NakladanieOp` ma wartość różną od zera, to dla każdego z wygenerowanych potomków jest wybierany losowo i wykonywany operator mutacyjny. Tak utworzone potomstwo jest dodawane do populacji potomnej `Ot`.

5.4 Funkcja `reprodukcja(Pt, Rodzice, Operatory->nRodzicow[Operator]);`

Reprodukcja jest w opisywanej implementacji wykonywana „na żądanie" — wybierane jest każdorazowo tyle osobników rodzicielskich, ile jest potrzebne do wykonania operatora genetycznego. Jest to pewne odstępstwo w stosunku do schematu algorytmu ewolucyjnego, w którym reprodukcja jest dla wygody zapisu wyodrębniana jako osobny etap (co sugeruje niezależność od operatorów genetycznych). Schemat taki jest jednak nie do utrzymania jeśli dopuszcza się operatory genetyczne różniące się między sobą liczbą rodziców i potomków; stąd przyjęta w oprogramowaniu konwencja.

6 Rozbudowa oprogramowania

Oprogramowanie jest zbudowane w taki sposób, aby użytkownik mógł w stosunkowo łatwy sposób wymieniać istniejące gotowe elementy, jak również definiować własne specyficzne schematy ewolucji. Fragmenty kodu związane ze sobą tematycznie zostały zgrupowane w osobnych plikach z kodem źródłowym. Poniżej zostanie omówiona w skrócie zawartość tych plików z krótkim komentarzem o możliwościach modyfikacji.

6.1 Plik `algorytm.c`

Funkcje zawarte w tym pliku są wykorzystywane do implementacji podstawowych mechanizmów algorytmu.

6.1.1 `algorytm.h`

```
/*
+--------------------------------+
| (C) Jaroslaw Arabas, luty 1999 |
+--------------------------------+
*/

#ifndef ALGORYTM_H
#define ALGORYTM_H

#include <stdio.h>
#include "pop.h"

typedef struct{
    unsigned nPop;
    unsigned Samoadaptacja;
} ALGORYTMS;

typedef ALGORYTMS* ALGORYTM;

/* glowna petla algorytmu */
void petlaGlowna();

/* genetyka tworzy populacje O_t modyfikujac za pomoca operatorow
    genetycznych chromosomy z populacji pomocniczej */
char genetyka (POP Pom, POP Ot);


void okreslAlgorytm(FILE*, FILE*);

extern ALGORYTM Algorytm;

#endif
```

6.1.2 `algorytm.c`

```
/*
+--------------------------------+
| (C) Jaroslaw Arabas, luty 1999 |
+--------------------------------+
*/

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "pop.h"
#include "losowe.h"
#include "algorytm.h"
#include "raporty.h"
```

```
#include "genetyka.h"
#include "selekcja.h"
#include "stop.h"
#include "wewy.h"
#include "zadanie.h"

ALGORYTM Algorytm;

/* Funkcja tworzy populacje Ot z populacji Pt poprzez krzyzowanie (wraz z
reprodukcja) oraz mutacje */

char genetyka (POP Pt, POP Ot)
{
    unsigned i,j;
    unsigned nSkopiowanych;
    unsigned Operator;
    unsigned OperatorM;
    unsigned *Rodzice;
    OSOBNIK *Dzieci;

    i=0;

    Rodzice=(unsigned*)calloc(Pt->nOsobnik, sizeof (unsigned));
    Dzieci=(OSOBNIK*)calloc(Ot->nOsobnik, sizeof (OSOBNIK));

    nSkopiowanych=0;
    while (nSkopiowanych < Ot->nOsobnik)
    {
        Operator = wyborOperatora (Operatory,0);
        reprodukcja(Pt, Rodzice, Operatory->nRodzicow[Operator]);
        for(j=0; j<Operatory->nDzieci[Operator]; j++)
            Dzieci[j]=osNowy();
        operatorG(Operatory->tOperatoraC[Operator],Pt->Osobniki, Rodzice,
                    Operatory->nRodzicow[Operator],
                    Operatory->nDzieci[Operator], Dzieci);

        for (i=0; i<Operatory->nDzieci[Operator] && nSkopiowanych < Ot->nOsobnik ; i++)
        {
            if (Operatory->NakladanieOp>0)
            {
                OperatorM = wyborOperatora (Operatory,1);

                operatorG (Operatory->tOperatoraM[OperatorM],Dzieci, &i,
                            1, 1, Dzieci+i);
            }
            Ot->Osobniki[nSkopiowanych++]=Dzieci[i];
        }
    }

    free(Rodzice);
    free(Dzieci);

    return 0;
}

/* Glowna petla algorytmu ewolucyjnego */
void petlaGlowna()
{
    POP Pt, Ot, Pnowa;
    unsigned t=0;
```

```
    Pt=popNowa(Algorytm→nPop);

    popOsobniki(Pt); /*proteza*/
    popInic(Pt); /*proteza*/

    popEwaluacja(Pt);
    popMonitor(Pt);
    zadMonitor(t,Pt);

    while (!krytStopu(Zadanie))
    {
        rapPop(Pt,NULL);
        rapZadanie(Zadanie,NULL);
        Ot=popNowa(Algorytm→nPop);
        genetyka(Pt, Ot);
        popEwaluacja(Ot);
        popMonitor(Ot);
        Pnowa=popNowa(Algorytm→nPop);
        sukcesja(Pnowa, Pt, Ot);
        popUsun(Pt);
        popUsun(Ot);
        t++;
        Pt=Pnowa;
        popMonitor(Pt);
        zadMonitor(t,Pt);
    }
    rapPop(Pt,NULL);
    rapZadanie(Zadanie,NULL);

}

#define ALG_PAR 2
void okreslAlgorytm (FILE *plikKonf, FILE* plikKontr)
{

    char Wczytanych;
    char *Nazwy[ALG_PAR];
    CZYTFUN czytFun[ALG_PAR];
    void *Adresy[ALG_PAR];
    unsigned *Liczba[ALG_PAR];

    Algorytm = (ALGORYTM) malloc(sizeof(ALGORYTMS));
    assert(Algorytm!=NULL);

    Nazwy[0]="npop"; czytFun[0]=czytUnsigned; Adresy[0]=&(Algorytm→nPop); Liczba[0]=NULL;
    Nazwy[1]="samoadaptacja"; czytFun[1]=czytUnsigned; Adresy[1]=&(Algorytm→Samoadaptacja);
Liczba[1]=NULL;

    Wczytanych = wczytajSymbole (plikKonf, plikKontr, Nazwy, czytFun, Adresy, Liczba, ALG_PAR);
    /*assert (Wczytanych==ALG_PAR);*/

}
```

6.2 Plik `genetyka.c`

6.2.1 Zawartość

Plik ten zawiera definicje operatorów genetycznych. Jest w nim także zdefiniowana zmienna globalna `Operatory`, w której zapisane są informacje konfiguracyjne o operatorach (rozkład prawdopodobieństwa, liczba rodziców i potomków, nakładanie itp.).

Zdefiniowana jest większość operatorów krzyżowania i mutacji, które są wykorzystywane do przekształcania osobników kodowanych jako wektory liczb całkowitych i rzeczywistych. W przypadku operatorów mutacji zapewniony jest jeszcze opcjonalny mechanizm samoczynnej adaptacji (wymagający chromosomów wzbogaconych o wektor parametrów). Mechanizm ten jest aktywowany wówczas, gdy wektor parametrów ma liczność równą liczności wektora zmiennych niezależnych.

W definicji operatorów mutacyjnych zostało założone, że wykonywana jest ona na dokładnie jednym rodzicu i prowadzi do utworzenia jednego potomka.

6.2.2 Sposób dopisania własnego operatora genetycznego

Nagłówej funkcji definiującej operator genetyczny jest następujący

```
char <nazwa> (const OSOBNIK *Osobniki, const unsigned *Rodzice, const unsigned nRodzicow,
const unsigned nDzieci, const OSOBNIK *Dzieci)
```

Poszczególne parametry mają następujące znaczenie

- `const OSOBNIK *Osobniki` Tablica osobników znajdujących się w populacji bazowej — potencjalnych kandydatów do bycia rodzicami.
- `const unsigned *Rodzice` Tablica numerów osobników, którzy zostali wybrani na osobniki rodzicielskie.
- `const unsigned nRodzicow` Liczba rodziców.
- `const unsigned nDzieci` Liczba potomków.
- `const OSOBNIK *Dzieci` Tablica osobników potomnych, wygenerowanych w wyniku działania operatora genetycznego. Tablica ta jest wypełniana przed wywołaniem operatora genetycznego osobnikami o losowych genotypach. Operator określa ich genotypy na podstawie genotypów osobników rodzicielskich.

Po zaimplementowaniu operatora genetycznego, należy go „zarejestrować". W tym celu należy dodać odpowiednią pozycję w funkcji `operatorG`, wybierającej operator do wykonania. Jeśli operator powinien otrzymywać jakieś szczególne parametry, można mu je przekazać poprzez wzbogacenie struktury `OPERATORYS` i dodanie do funkcji wczytujacej parametry operatorów genetycznych (`okreslOperatory`) odpowiedniej sekwencji wczytującej.

6.2.3 `genetyka.h`

```
/*
+————————————–+
| (C) Jaroslaw Arabas, luty 1999 |
+————————————–+
*/
#ifndef GENETYKA_H
#define GENETYKA_H

#include "osobnik.h"

/* Definicja operatora genetycznego. Wszystkie osobniki rodzicielskie znajduja
sie w tablicy Osobniki, tablica Rodzice okresla, ktore z nich zostana
zreprodukowane. Wynikiem jest tablica Dzieci. Parametr nDzieci okresla ich
liczbe.
*/
```

```
char operatorG(unsigned Nr, const OSOBNIK *Osobniki, const unsigned *Rodzice,
                         const unsigned nRodzicow, const unsigned nDzieci,
                         const OSOBNIK *Dzieci);

/* Definicja struktury przechowujacej informacje o operatorach genetycznych.
Kazdy operator ma okreslona liczbe rodzicow i dzieci. Dodatkowo sa dane
prawdopodobienstwa uzycia poszczegolnych operatorow, a takze wskazania do
funkcji realizujacych dzialanie operatorow.
*/

typedef struct{
   unsigned nOperatorowC;
   unsigned *tOperatoraC;
   double *PstwoC;
   unsigned *nRodzicow;
   unsigned *nDzieci;
   unsigned nOperatorowM;
   unsigned *tOperatoraM;
   double *PstwoM;
   unsigned nPKrzyz;
   double TAU;
   double TAUP;
   double MinSigma;
   double pMut;
   unsigned NakladanieOp;
} OPERATORYS;

typedef OPERATORYS* OPERATORY;

/* Fukcja konfiguruje - tworzy informacje o operatorach i ustawia
odpowiednie parametry */


void okreslOperatory (FILE*, FILE*);

extern OPERATORY Operatory;

/* Funkcja zwraca numer operatora z tablicy wszystkich dostepnych. Ktore
 * okresla, czy operatory krzyzowania (bazowe), czy tez operatory mutacyjne
 * beda brane pod uwage */
unsigned wyborOperatora (const OPERATORY Operatory, unsigned ktore);

#endif
```

6.2.4 genetyka.c

```
/*
+---------------------------+
| (C) Jaroslaw Arabas, luty 1999 |
+---------------------------+
*/

#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include "osobnik.h"
#include "losowe.h"
#include "zadanie.h"
#include "pop.h"
#include "genetyka.h"
```

```
#include "wewy.h"

OPERATORY Operatory;


char kopiowanie (const OSOBNIK *, const unsigned *, const unsigned , const unsigned , const
OSOBNIK *);
char kRown (const OSOBNIK *, const unsigned *, const unsigned , const unsigned , const
OSOBNIK *);
char k1p (const OSOBNIK *, const unsigned *, const unsigned , const unsigned , const OSOBNIK
*);
char kKp (const OSOBNIK *, const unsigned *, const unsigned , const unsigned , const
OSOBNIK *);
char kUsrStd (const OSOBNIK *, const unsigned *, const unsigned , const unsigned , const
OSOBNIK *);
char kUsrMod (const OSOBNIK *, const unsigned *, const unsigned , const unsigned , const
OSOBNIK *);
char kNRown (const OSOBNIK *, const unsigned *, const unsigned , const unsigned , const
OSOBNIK *);
char kNUsrStd (const OSOBNIK *, const unsigned *, const unsigned , const unsigned , const
OSOBNIK *);
char kNUsrMod (const OSOBNIK *, const unsigned *, const unsigned , const unsigned , const
OSOBNIK *);
char mDysk (const OSOBNIK *, const unsigned *, const unsigned , const unsigned , const
OSOBNIK *);
char mProst (const OSOBNIK *, const unsigned *, const unsigned , const unsigned , const
OSOBNIK *);
char mNorm (const OSOBNIK *, const unsigned *, const unsigned , const unsigned , const
OSOBNIK *);
char mCauchy (const OSOBNIK *, const unsigned *, const unsigned , const unsigned , const
OSOBNIK *);


char operatorG (unsigned Nr, const OSOBNIK *Osobniki, const unsigned *Rodzice,
                const unsigned nRodzicow, const unsigned nDzieci,
                const OSOBNIK *Dzieci)
{
  switch (Nr)
  {
  case 0: return kopiowanie (Osobniki, Rodzice, nRodzicow, nDzieci, Dzieci);
  case 1: return kRown (Osobniki, Rodzice, nRodzicow, nDzieci, Dzieci);
  case 2: return k1p (Osobniki, Rodzice, nRodzicow, nDzieci, Dzieci);
  case 3: return kKp (Osobniki, Rodzice, nRodzicow, nDzieci, Dzieci);
  case 4: return kUsrStd (Osobniki, Rodzice, nRodzicow, nDzieci, Dzieci);
  case 5: return kUsrMod (Osobniki, Rodzice, nRodzicow, nDzieci, Dzieci);
  case 6: return kNRown (Osobniki, Rodzice, nRodzicow, nDzieci, Dzieci);
  case 7: return kNUsrStd (Osobniki, Rodzice, nRodzicow, nDzieci, Dzieci);
  case 8: return kNUsrMod (Osobniki, Rodzice, nRodzicow, nDzieci, Dzieci);
  case 9: return mDysk (Osobniki, Rodzice, nRodzicow, nDzieci, Dzieci);
  case 10: return mProst (Osobniki, Rodzice, nRodzicow, nDzieci, Dzieci);
  case 11: return mNorm (Osobniki, Rodzice, nRodzicow, nDzieci, Dzieci);
  case 12: return mCauchy (Osobniki, Rodzice, nRodzicow, nDzieci, Dzieci);
  }
}

/* Funkcja zwraca numer operatora z tablicy wszystkich dostepnych */
unsigned wyborOperatora (const OPERATORY Operatory, unsigned ktore)
{
  if (ktore==0)
  return ruletka(Operatory→PstwoC, Operatory→nOperatorowC);
```

```
    if (ktore==1)
    return ruletka(Operatory→PstwoM, Operatory→nOperatorowM);
}


char kopiowanie (const OSOBNIK *Osobniki, const unsigned *Rodzice,
                 const unsigned nRodzicow, const unsigned nDzieci,
                 const OSOBNIK *Dzieci)
{

    unsigned i;
    unsigned Rodzic;
    unsigned Dziecko;

    assert(nRodzicow==nDzieci);
    for (Dziecko=0; Dziecko<nDzieci; Dziecko++)
    {
        Rodzic=Dziecko;
        for (i=0; i< Osobniki[Rodzice[Rodzic]]→nGenow ; i++)
            Dzieci[Dziecko]→Genotyp[i] = Osobniki[Rodzice[Rodzic]]→Genotyp[i];
        for (i=0; i< Osobniki[Rodzice[Rodzic]]→nPar; i++)
            Dzieci[Dziecko]→Parametry[i] = Osobniki[Rodzice[Rodzic]]→Parametry[i];
    }
    return 0;
}
/**********************************************************************/
/* DWUOSOBNICZE KRZYZOWANIE 2->2 */
/**********************************************************************/


/*——————————————————————————————-
Krzyzowanie rownomierne.

G. Syswerda proponowal ustalic pstwo wymiary 0.5, tutaj natomiast jest ono
zmienna losowa o rozkladzie rownomiernym.

Uogolnione na przypadek samoczynnej adaptacji.
———————————————————————————————-*/
char kRown (const OSOBNIK *Osobniki, const unsigned *Rodzice,
            const unsigned nRodzicow, const unsigned nDzieci,
            const OSOBNIK *Dzieci)
{
    double PstwoZamiany;
    unsigned i;


    assert(nRodzicow==2);
    assert(nDzieci==2);

    PstwoZamiany = prostokatny(0,1);

    for (i=0; i< Osobniki[Rodzice[0]]→nGenow ; i++)
    {
        if (dwupunktowy(PstwoZamiany))
        {
            Dzieci[0]→Genotyp[i] = Osobniki[Rodzice[1]]→Genotyp[i];
            Dzieci[1]→Genotyp[i] = Osobniki[Rodzice[0]]→Genotyp[i];
            if (Osobniki[Rodzice[0]]→nPar==Osobniki[Rodzice[0]]→nGenow)
            {
                Dzieci[0]→Parametry[i] = Osobniki[Rodzice[1]]→Parametry[i];
```

```
                Dzieci[1]→Parametry[i] = Osobniki[Rodzice[0]]→Parametry[i];
            }

        }
        else
        {
            Dzieci[0]→Genotyp[i] = Osobniki[Rodzice[0]]→Genotyp[i];
            Dzieci[1]→Genotyp[i] = Osobniki[Rodzice[1]]→Genotyp[i];
            if (Osobniki[Rodzice[0]]→nPar==Osobniki[Rodzice[0]]→nGenow)
            {
                Dzieci[0]→Parametry[i] = Osobniki[Rodzice[0]]→Parametry[i];
                Dzieci[1]→Parametry[i] = Osobniki[Rodzice[1]]→Parametry[i];
            }
        }
    }

    return 0;
}

/*——————————————————————————————-
Krzyzowanie jednopunktowe. Uogolnione na przypadek samoczynnej adaptacji.
——————————————————————————————-*/

char k1p (const OSOBNIK *Osobniki, const unsigned *Rodzice,
                const unsigned nRodzicow, const unsigned nDzieci,
                const OSOBNIK *Dzieci)
{
    unsigned Punkt;
    unsigned i;

    assert(nRodzicow==2);
    assert(nDzieci==2);

    Punkt = rownomierny (Osobniki[Rodzice[0]]→nGenow -2)+1;

    for (i=0; i<Punkt; i++)
    {
        Dzieci[0]→Genotyp[i] = Osobniki[Rodzice[0]]→Genotyp[i];
        Dzieci[1]→Genotyp[i] = Osobniki[Rodzice[1]]→Genotyp[i];
    }
    if (Osobniki[Rodzice[0]]→nPar==Osobniki[Rodzice[0]]→nGenow)
        for (i=0; i<Punkt; i++)
        {
            Dzieci[0]→Parametry[i] = Osobniki[Rodzice[0]]→Parametry[i];
            Dzieci[1]→Parametry[i] = Osobniki[Rodzice[1]]→Parametry[i];
        }

    for ( i=Punkt; i< Osobniki[Rodzice[0]]→nGenow ; i++)
    {
        Dzieci[0]→Genotyp[i] = Osobniki[Rodzice[1]]→Genotyp[i];
        Dzieci[1]→Genotyp[i] = Osobniki[Rodzice[0]]→Genotyp[i];
    }
    if (Osobniki[Rodzice[0]]→nPar==Osobniki[Rodzice[0]]→nGenow)
        for ( i=Punkt; i< Osobniki[Rodzice[0]]→nGenow ; i++)
        {
            Dzieci[0]→Parametry[i] = Osobniki[Rodzice[1]]→Parametry[i];
            Dzieci[1]→Parametry[i] = Osobniki[Rodzice[0]]→Parametry[i];
        }

    return 0;
```

```
}
/*-----------------------------------------------------------------
Krzyzowanie k-punktowe. Wybierane jest losowo k punktow w taki sposob, ze
punkty nie zachodza na siebie nawzajem, tzn. jedno miejse krzyzowania nie
moze byc wielokrotnie wybrane.
Uogolnione na przypadek samoczynnej adaptacji.
-----------------------------------------------------------------*/

char kKp (const OSOBNIK *Osobniki, const unsigned *Rodzice,
                const unsigned nRodzicow, const unsigned nDzieci,
                const OSOBNIK *Dzieci)
{
    unsigned *Punkty;
    unsigned i, j;
    char Zamiana;

    assert(nRodzicow==2);
    assert(nDzieci==2);

/* przygotowanie tablicy na punkty krzyzowania */
    Punkty = (unsigned*)calloc (Operatory->nPKrzyz, sizeof(unsigned));
    kombinacja (Punkty, Osobniki[Rodzice[0]]->nGenow, Operatory->nPKrzyz);

/* wlasciwe krzyzowanie */
    Zamiana=0;
    j=0;
    for (i=0; i<Osobniki[Rodzice[0]]->nGenow; i++)
    {
        if (i==Punkty[j])
            {
            Zamiana = !Zamiana;
            j++;
            }

        if (Zamiana)
        {
            Dzieci[0]->Genotyp[i] = Osobniki[Rodzice[1]]->Genotyp[i];
            Dzieci[1]->Genotyp[i] = Osobniki[Rodzice[0]]->Genotyp[i];
            if (Osobniki[Rodzice[0]]->nPar==Osobniki[Rodzice[0]]->nGenow)
            {
                Dzieci[0]->Parametry[i] = Osobniki[Rodzice[1]]->Parametry[i];
                Dzieci[1]->Parametry[i] = Osobniki[Rodzice[0]]->Parametry[i];
            }
        }
        else
        {
            Dzieci[0]->Genotyp[i] = Osobniki[Rodzice[0]]->Genotyp[i];
            Dzieci[1]->Genotyp[i] = Osobniki[Rodzice[1]]->Genotyp[i];
            if (Osobniki[Rodzice[0]]->nPar==Osobniki[Rodzice[0]]->nGenow)
            {
                Dzieci[0]->Parametry[i] = Osobniki[Rodzice[0]]->Parametry[i];
                Dzieci[1]->Parametry[i] = Osobniki[Rodzice[1]]->Parametry[i];
            }
        }
    }

/* czynnosci porzadkowe */
    free (Punkty);

    return 0;
```

```
}

/*———————————————————————————————-
Krzyzowanie usredniajace — standardowe (wzdluz odcinka).
Uogolnione na przypadk samoczynnej adaptacji - rownomierne tutaj.
———————————————————————————————-*/

char kUsrStd (const OSOBNIK *Osobniki, const unsigned *Rodzice,
              const unsigned nRodzicow, const unsigned nDzieci,
              const OSOBNIK *Dzieci)
{
    double a;
    unsigned i;


    assert(nRodzicow==2);
    assert(nDzieci==2);

    a = jednostajny();

    for (i=0; i<Osobniki[Rodzice[0]]→nGenow; i++)
    {
        Dzieci[0]→Genotyp[i] = a*Osobniki[Rodzice[0]]→Genotyp[i]+
                               (1-a)*Osobniki[Rodzice[1]]→Genotyp[i];
        Dzieci[1]→Genotyp[i] = a*Osobniki[Rodzice[1]]→Genotyp[i]+
                               (1-a)*Osobniki[Rodzice[0]]→Genotyp[i];
    }
    for (i=0; i<Osobniki[Rodzice[0]]→nPar; i++)
    {
        if (dwupunktowy(a))
        {
            Dzieci[0]→Parametry[i] = Osobniki[Rodzice[0]]→Parametry[i];
            Dzieci[1]→Parametry[i] = Osobniki[Rodzice[1]]→Parametry[i];
        }
        else
        {
            Dzieci[0]→Parametry[i] = Osobniki[Rodzice[1]]→Parametry[i];
            Dzieci[1]→Parametry[i] = Osobniki[Rodzice[0]]→Parametry[i];
        }
    }
    return 0;
}

/*———————————————————————————————-
Krzyzowanie usredniajace — wariant zmodyfikowany (w przestrzeni).
Uogolnione na przypadek samocyznnej adaptacji (musi byc jednakowa liczba
genow i parametrow).
———————————————————————————————-*/

char kUsrMod (const OSOBNIK *Osobniki, const unsigned *Rodzice,
              const unsigned nRodzicow, const unsigned nDzieci,
              const OSOBNIK *Dzieci)
{
    double ai;
    unsigned i;

    assert(nRodzicow==2);
    assert(nDzieci==2);

    for (i=0; i<Osobniki[Rodzice[0]]→nGenow; i++)
```

```
    {
        ai = jednostajny();
        Dzieci[0]→Genotyp[i] = ai*Osobniki[Rodzice[0]]→Genotyp[i]+
                                (1-ai)*Osobniki[Rodzice[1]]→Genotyp[i];
        Dzieci[1]→Genotyp[i] = ai*Osobniki[Rodzice[1]]→Genotyp[i]+
                                (1-ai)*Osobniki[Rodzice[0]]→Genotyp[i];
        if (Osobniki[Rodzice[0]]→nPar==Osobniki[Rodzice[0]]→nGenow)
            if (dwupunktowy(ai))
            {
                Dzieci[0]→Parametry[i] = Osobniki[Rodzice[0]]→Parametry[i];
                Dzieci[1]→Parametry[i] = Osobniki[Rodzice[1]]→Parametry[i];
            }
            else
            {
                Dzieci[0]→Parametry[i] = Osobniki[Rodzice[1]]→Parametry[i];
                Dzieci[1]→Parametry[i] = Osobniki[Rodzice[0]]→Parametry[i];
            }

    }
    return 0;
}

/**********************************************************************/
/* WIELOOSOBNICZE KRYZOWANIE N->1 */
/**********************************************************************/


/*-------------------------------------------------------------------
Krzyzowanie rownomierne.
Uogolnione na przypadek samoczynnej adaptacji.
--------------------------------------------------------------------*/

char kNRown (const OSOBNIK *Osobniki, const unsigned *Rodzice,
                const unsigned nRodzicow, const unsigned nDzieci,
                const OSOBNIK *Dzieci)
{
    unsigned *RozkladZamiany;
    unsigned i;

    assert(nRodzicow>=2);
    assert(nDzieci==1);

    RozkladZamiany=(unsigned*)calloc (nRodzicow, sizeof(unsigned));
    dyskNWym(RozkladZamiany, nRodzicow, nRodzicow);


    for (i=0; i<Osobniki[Rodzice[0]]→nGenow ; i++)
    {
        Dzieci[0]→Genotyp[i] = Osobniki[Rodzice[ RozkladZamiany[i] ]]→Genotyp[i];
        if (Osobniki[Rodzice[0]]→nPar==Osobniki[Rodzice[0]]→nGenow)
              Dzieci[0]→Parametry[i] = Osobniki[Rodzice[RozkladZamiany[i]]]→Parametry[i];
    }

    free (RozkladZamiany);
    return 0;
}

/*-------------------------------------------------------------------
Kombinacja wypukla — wariant standardowy.
Uogolnione na przypadk samoczynnej adaptacji - rownomierne tutaj.
```

```
——————————————————————————————————-*/

char kNUsrStd (const OSOBNIK *Osobniki, const unsigned *Rodzice,
                const unsigned nRodzicow, const unsigned nDzieci,
                const OSOBNIK *Dzieci)
{
    double *TabWsp;
    unsigned i,j, a;

    assert(nRodzicow>=2);
    assert(nDzieci==1);

/* przygotowanie tablicy wspolczynnikow usredniania */
    TabWsp = (double*)calloc (nRodzicow, sizeof(double));
    normJednNWym (TabWsp, nRodzicow);

    for (i=0; i<Osobniki[Rodzice[0]]->nGenow; i++)
    {
        Dzieci[0]->Genotyp[i]=0;
        for (j=0; j<nRodzicow; j++)
            Dzieci[0]->Genotyp[i] += TabWsp[j]*Osobniki[Rodzice[j]]->Genotyp[i];

    }
    for (i=0; i<Osobniki[Rodzice[0]]->nPar; i++)
    {
        a=ruletka (TabWsp, nRodzicow);
          Dzieci[0]->Parametry[i] = Osobniki[Rodzice[a]]->Parametry[i];
    }

    free (TabWsp);
    return 0;
}


/*——————————————————————————————————-
Kombinacja wypukla — wariant zmodyfikowany.
Uogolnione na przypadk samoczynnej adaptacji - rownomierne tutaj.
——————————————————————————————————-*/

char kNUsrMod (const OSOBNIK *Osobniki, const unsigned *Rodzice,
                const unsigned nRodzicow, const unsigned nDzieci,
                const OSOBNIK *Dzieci)
{
    double *TabWsp;
    unsigned i, j, ai;


    assert(nRodzicow>=2);
    assert(nDzieci==1);

/* przygotowanie tablicy wspolczynnikow usredniania */
    TabWsp = (double*)calloc (nRodzicow, sizeof(double));

    for (i=0; i<Osobniki[Rodzice[0]]->nGenow ; i++)
    {
        normJednNWym (TabWsp, nRodzicow);
        Dzieci[0]->Genotyp[i]=0;
        for (j=0; j<nRodzicow; j++)
            Dzieci[0]->Genotyp[i] += TabWsp[j]*Osobniki[Rodzice[j]]->Genotyp[i];
```

```
        if (Osobniki[Rodzice[0]]→nPar==Osobniki[Rodzice[0]]→nGenow)
        {
            ai=ruletka (TabWsp, nRodzicow);
               Dzieci[0]→Parametry[i] = Osobniki[Rodzice[ai]]→Parametry[i];
        }
    }

    free (TabWsp);
    return 0;
}


/*************************************************************************/
/* MUTACJE */
/*************************************************************************/


/*---------------------------------
Mutacja polegajaca na zamianie litery genotypu na wartosc losowa ze zbioru
liter alfabetu uzywanych do kodowania.
----------------------------------*/

char mDysk (const OSOBNIK *Osobniki, const unsigned *Rodzice,
                const unsigned nRodzicow, const unsigned nDzieci,
                const OSOBNIK *Dzieci)
{
    unsigned i;

    assert(nRodzicow==1);
    assert(nDzieci==1);

    for (i=0; i<Osobniki[Rodzice[0]]→nGenow ; i++)
        if (dwupunktowy(Operatory→pMut))
            Dzieci[0]→Genotyp[i]=rownomierny (Zadanie→nLiter);
    return 0;
}


/*---------------------------------
Mutacja z rozkladem prostokatnym - znudzilem sie
----------------------------------*/

char mProst (const OSOBNIK *Osobniki, const unsigned *Rodzice,
                const unsigned nRodzicow, const unsigned nDzieci,
                const OSOBNIK *Dzieci)
{
    unsigned i;

    assert(nRodzicow==1);
    assert(nDzieci==1);

    for (i=0; i<Osobniki[Rodzice[0]]→nGenow; i++)
        if (dwupunktowy(Operatory→pMut))
            Dzieci[0]→Genotyp[i] =
                            prostokatny(Zadanie→DolneX[i], Zadanie→GorneX[i]);
    return 0;
}

/*---------------------------------
 Mutacja ze strategii ewolucyjnej, nieskorelowana, rozkladem normalnym.
```

```
-----------------------------------------------------------------*/
char mNorm (const OSOBNIK *Osobniki, const unsigned *Rodzice,
            const unsigned nRodzicow, const unsigned nDzieci,
            const OSOBNIK *Dzieci)
{
   unsigned i;
   double N;

   assert(nRodzicow==1);
   assert(nDzieci==1);

   N=normstd();

   for (i=0; i<Osobniki[Rodzice[0]]->nGenow; i++)
   {
      if (Osobniki[Rodzice[0]]->nPar>0)
      {
         Dzieci[0]->Parametry[i] = Osobniki[Rodzice[0]]->Parametry[i]
                      *exp(Operatory->TAUP*N + Operatory->TAU*normstd() );
         if (Dzieci[0]->Parametry[i] < Operatory->MinSigma)
            Dzieci[0]->Parametry[i] = Operatory->MinSigma;
         Dzieci[0]->Genotyp[i] = Osobniki[Rodzice[0]]->Genotyp[i] +
                                 normstd()*Dzieci[0]->Parametry[i];
      }
      else
      {
         Dzieci[0]->Genotyp[i] = Osobniki[Rodzice[0]]->Genotyp[i] +
                                 normstd()*Operatory->MinSigma;
      }

   }
   return 0;
}

/*-----------------------------------------------------------------
 Mutacja ze strategii ewolucyjnej, nieskorelowana, rozkladem Cauchy'ego
-----------------------------------------------------------------*/

char mCauchy (const OSOBNIK *Osobniki, const unsigned *Rodzice,
            const unsigned nRodzicow, const unsigned nDzieci,
            const OSOBNIK *Dzieci)
{
   unsigned i;
   double N;

   assert(nRodzicow==1);
   assert(nDzieci==1);

   N=normstd();

   for (i=0; i<Osobniki[Rodzice[0]]->nGenow; i++)
   {
      if (Osobniki[Rodzice[0]]->nPar>0)
      {
         Dzieci[0]->Parametry[i] = Osobniki[Rodzice[0]]->Parametry[i]
                      *exp(Operatory->TAUP*N + Operatory->TAU*normstd() );
         if (Dzieci[0]->Parametry[i] < Operatory->MinSigma)
            Dzieci[0]->Parametry[i] = Operatory->MinSigma;
         Dzieci[0]->Genotyp[i] = Osobniki[Rodzice[0]]->Genotyp[i] +
                                 cauchy()*Dzieci[0]->Parametry[i];
```

```
        }
        else
        {
            Dzieci[0]->Genotyp[i] = Osobniki[Rodzice[0]]->Genotyp[i] +
                                    cauchy()*Operatory->MinSigma;
        }
    }
    return 0;
}
```

```
#define GEN_PAR 14
void okreslOperatory (FILE* plikKonf, FILE *plikKontr)
{
    char Wczytanych;
    char *Nazwy[GEN_PAR];
    CZYTFUN czytFun[GEN_PAR];
    void *Adresy[GEN_PAR];
    unsigned *Liczba[GEN_PAR];

    Operatory=(OPERATORY)malloc(sizeof(OPERATORYS));
    assert(Operatory!=NULL);

    Nazwy[0]="noperatorow"; czytFun[0]=czytUnsigned; Adresy[0]=&(Operatory->nOperatorowC);
Liczba[0]=NULL;
    Nazwy[1]="npkrzyz"; czytFun[1]=czytUnsigned; Adresy[1]=&(Operatory->nPKrzyz);
Liczba[1]=NULL;
    Nazwy[2]="tau"; czytFun[2]=czytDouble; Adresy[2]=&(Operatory->TAU); Liczba[2]=NULL;
    Nazwy[3]="taup"; czytFun[3]=czytDouble; Adresy[3]=&(Operatory->TAUP); Liczba[3]=NULL;
    Nazwy[4]="minsigma"; czytFun[4]=czytDouble; Adresy[4]=&(Operatory->MinSigma);
Liczba[4]=NULL;
    Nazwy[5]="pmut"; czytFun[5]=czytDouble; Adresy[5]=&(Operatory->pMut); Liczba[5]=NULL;
    Nazwy[6]="pstwo"; czytFun[6]=czytNDouble; Adresy[6]=&(Operatory->PstwoC);
Liczba[6]=&(Operatory->nOperatorowC);
    Nazwy[7]="nrodzicow"; czytFun[7]=czytNUnsigned; Adresy[7]=&(Operatory->nRodzicow);
Liczba[7]=&(Operatory->nOperatorowC);
    Nazwy[8]="ndzieci"; czytFun[8]=czytNUnsigned; Adresy[8]=&(Operatory->nDzieci);
Liczba[8]=&(Operatory->nOperatorowC);
    Nazwy[9]="toperatora"; czytFun[9]=czytNUnsigned; Adresy[9]=&(Operatory->tOperatoraC);
Liczba[9]=&(Operatory->nOperatorowC);
    Nazwy[10]="pstwom"; czytFun[10]=czytNDouble; Adresy[10]=&(Operatory->PstwoM);
Liczba[10]=&(Operatory->nOperatorowM);
    Nazwy[11]="toperatoram"; czytFun[11]=czytNUnsigned; Adresy[11]=&(Operatory->tOperatoraM);
Liczba[11]=&(Operatory->nOperatorowM);
    Nazwy[12]="nakladanieop"; czytFun[12]=czytUnsigned; Adresy[12]=&(Operatory->NakladanieOp);
Liczba[12]=NULL;
    Nazwy[13]="noperatorowm"; czytFun[13]=czytUnsigned; Adresy[13]=&(Operatory->nOperatorowM);
Liczba[13]=NULL;

    Wczytanych = wczytajSymbole (plikKonf, plikKontr, Nazwy, czytFun, Adresy, Liczba, GEN_PAR);
/* assert (Wczytanych==GEN_PAR);*/

}
```

6.3 Plik `glowny.c`

Plik ten zawiera funkcję `main`, zarządzającą wykonaniem programu, oraz `okreslParametry`, która wywołuje funkcje odczytujące zawartość plików konfiguracyjnych i ustawiające odpowiednie strutury konfiguracyjne.

6.3.1 glowny.c

```
/*
+-----------------------------+
| (C) Jaroslaw Arabas, luty 1999 |
+-----------------------------+
*/

#include <assert.h>
#include <stdio.h>

#include "algorytm.h"
#include "genetyka.h"
#include "losowe.h"
#include "ogranicz.h"
#include "raporty.h"
#include "selekcja.h"
#include "stop.h"
#include "zadanie.h"

void okreslParametry();

int main()
{
    okreslParametry();
    initLosowe();
    petlaGlowna();
    return 0;
}

void okreslParametry()
{
    FILE *plikiPar[8];
    FILE *plikLog;

    plikLog=fopen("gabi.log","w");

    plikiPar[0]=fopen("ogranicz.par","r");
    assert(plikiPar[0]!=NULL);
    okreslOgraniczenia(plikiPar[0], plikLog);
    fclose(plikiPar[0]);

    plikiPar[1]=fopen("algorytm.par","r");
    assert(plikiPar[1]!=NULL);
    okreslAlgorytm(plikiPar[1], plikLog);
    fclose(plikiPar[1]);

    plikiPar[2]=fopen("raporty.par","r");
    assert(plikiPar[2]!=NULL);
    okreslRaporty(plikiPar[2], plikLog);
    fclose(plikiPar[2]);

    plikiPar[3]=fopen("selekcja.par","r");
    assert(plikiPar[3]!=NULL);
    okreslSelekcje(plikiPar[3], plikLog);
    fclose(plikiPar[3]);

    plikiPar[4]=fopen("stop.par","r");
    assert(plikiPar[4]!=NULL);
    okreslStop(plikiPar[4], plikLog);
```

```
    fclose(plikiPar[4]);

    plikiPar[5]=fopen("losowe.par","r");
    assert(plikiPar[5]!=NULL);
    okreslLosowe(plikiPar[5], plikLog);
    fclose(plikiPar[5]);

    plikiPar[6]=fopen("genetyka.par","r");
    assert(plikiPar[6]!=NULL);
    okreslOperatory(plikiPar[6], plikLog);
    fclose(plikiPar[6]);

    plikiPar[7]=fopen("zadanie.par","r");
    assert(plikiPar[7]!=NULL);
    okreslZadanie(plikiPar[7]);
    fclose(plikiPar[7]);


    fclose(plikLog);
}
```

6.4 Plik `losowe.c`

W tym pliku zawarte są implementacje losowości w algorytmie ewolucyjnym. Generatory rozkładu jednostajnego pochodzą z monografii Wieczorkowskiego i Zielińskiego. Generatory innych rozkładów są zaimplementowane zgodnie ze wskazówkami zawartymi w tej pracy jak również w monografii Pacuta. Pochodzenie i przeznaczenie generatorów są opisane w kodzie źródłowym.

6.4.1 `losowe.h`

```
/*
+-----------------------------+
| (C) Jaroslaw Arabas, luty 1999 |
+-----------------------------+
*/

#ifndef LOSOWE_H
#define LOSOWE_H

#include <stdio.h>

/* Struktura przechowujaca informacje o tym, ktore generatory sa uzywane.
*/

typedef struct{
unsigned tJednostajny;
unsigned tNormalny;
unsigned tInicjacji;

unsigned InitZiarno;
} LOSOWES;

typedef LOSOWES* LOSOWE;

/* rozklad jednostajny - na przedziale [0..1], bazowy dla
pozostalych rozkladow */
```

```
double jednostajny ();

/*
Zmienna losowa o rozkladzie normalnym standaryzowanym
*/
double normstd();



/* inicjacja procedur pseudolosowych */

void initLosowe();

/*
Zmienna losowa o rozkladzie podanym jako argument, okreslona na zbiorze liczb
calkowitych nieujemnych [0..n-1].
*/

unsigned ruletka (double* rozklad, unsigned n);

/*
Zmienna losowa o rozkladzie dyskretnym, rownomiernym na zbiorze liczb
calkowitych nieujemnych z zakresu [0..n-1]
*/
unsigned rownomierny (unsigned n);

/*
Zmienna losowa o rozkladzie normalnym, o podanej wartosci oczekiwanej i
standardowym odchyleniu
*/
void normalny (double *X, const double *Mu, const double *Sigma, const unsigned n);

/*
Zmienna losowa o rozkladzie projstokatnym, tj jednostajnym na zbiorze liczb
rzeczywistych z zakresu [dolne .. gorne]
*/
double prostokatny (double dolne, double gorne);

/* Losowa permutacja zbioru n kolejnych liczb nieujemnych z zakresu [0..n-1]
W tablicy permutacja jest umieszczona wynikowa permutacja
*/

void permutacja (unsigned* permutacja, unsigned n);


/*
Zmienna losowa o rozkladzie dwupunktowym, zwracajaca
1 z pstwem jeden i 0 z pstwem 1-jeden
*/
char dwupunktowy (double jeden);

/*
Zmienna losowa o n-wymiarowym rozkladzie jednostajnym, kazdy z wymiarow
ma zmiennosc od 0 do 1
*/
void jednNWym (double *wektor, unsigned n);

/*
Zmienna losowa o n-wymiarowym rozkladzie jednostajnym.
Suma wartosci jest rowna 1.
```

```
*/
void normJednNWym (double *wektor, unsigned n);



/*
Zmienna losowa o n-wymiarowym rozkladzie jednostajnym, kazdy z wymiarow
ma zmiennosc od 0 do n
*/
void dyskNWym (unsigned *wektor, unsigned n, unsigned k);

void kombinacja (unsigned* Wynik, unsigned n, unsigned k);

/* Rozklad Cauchy'ego standaryzowany */
double cauchy();

void okreslLosowe(FILE*,FILE*);

extern LOSOWE Losowe;

#endif
```

6.4.2 losowe.c

```
/*
+----------------------------+
| (C) Jaroslaw Arabas, luty 1999 |
+----------------------------+
*/

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <time.h>
#include "defs.h"
#include "losowe.h"
#include "wewy.h"

LOSOWE Losowe;



void t1Randomize(unsigned s);
void t2Randomize();
void t1Init();
void t2Init();
void t3Init();
double t1Jedn(void);
double t2Jedn(void);
double t3Jedn(void);
double t1Norm();
double t2Norm();
double t3Norm();
double t4Norm();


void Randomize()
```

```
{
    switch (Losowe→tInicjacji)
    {
    case 1: t1Randomize(Losowe→InitZiarno);
            break;
    case 2: t2Randomize();
            break;
    default: t2Randomize();
            break;
    }
}

double jednostajny ()
{
    switch (Losowe→tJednostajny)
    {
    case 1: return t1Jedn();
    case 2: return t2Jedn();
    case 3: return t3Jedn();
    default: return 0;
    }
}

double normstd()
{
    switch (Losowe→tNormalny)
    {
    case 1: return t1Norm();
    case 2: return t2Norm();
    case 3: return t3Norm();
    case 4: return t4Norm();
    default: return t1Norm();
    }
}

void initLosowe()
{
    switch (Losowe→tJednostajny)
    {
    case 1: t1Init();
            break;
    case 2: t2Init();
            break;
    case 3: t3Init();
            break;
    default:
            break;
    }
}

void t1Randomize(unsigned s)
{
    srand(s);
}

void t2Randomize()
{
    srand((unsigned)time(0));
}
```

```
/* Generator uniwersalny o okresie 2^144, opis - por Wieczorkowski, Zielinski
   "Komputerowe generatory liczb losowych", WNT, Warszawa, 1997 str. 39 */

static double uu[97];
static int ip=97;
static int jp=33;
static double cc=362436.0/16777216.0;
static double cd=7654321.0/16777216.0;
static double cm=16777213.0/16777216.0;

void rstart(int i, int j, int k, int l)
{
    int ii, jj, m, wi, wj, wk, wl;
    double s,t;

    wi=i; wj=j; wk=k; wl=l;
    for (ii=0; ii<97; ii++)
    {
        s=0; t=0.5;
        for(jj=1; jj<=24; jj++)
        {
            m=(((wi*wj)%179)*wk)%179;
            wi=wj; wj=wk; wk=m;
            wl=(53*wl+1)%169;
            if ( (wl*m)%64 >= 32 )
                s+=t;
            t* =0.5;
        }
        uu[ii]=s;
    }
}

void t1Init()
{
    Randomize();
    rstart(rand()%178+1,rand()%178+1,rand()%178+1,rand()%169);
}


double t1Jedn(void)
{
    double pom;

    pom=uu[ip-1]-uu[jp-1];
    if (pom < 0.0)
        pom+=1;
    uu[ip-1]=pom;
    ip--;
    if (ip==0)
        ip=97;
    jp--;
    if (jp==0)
        jp=97;
    cc-=cd;
    if (cc<0.0)
        cc+=cm;
    pom-=cc;
    if (pom<0.0)
```

```
        pom+=1;
    return pom;

}

/* Ogolny generator liniowy o okresie rzedu 2∧96, opis - por Wieczorkowski, Zielinski
    "Komputerowe generatory liczb losowych", WNT, Warszawa, 1997 str. 37.
    W oryginale blad - omawiany tam generator daje liczby z przedzialu [-0.5,0.5] */
static double a,b,c;

void init_ecng (int ia, int ib, int ic)
{
    a=ia; b=ib; c=ic;
}

void t2Init()
{
    Randomize();
    init_ecng(rand()%100, rand()%100, rand()%100);
}

double t2Jedn()
{
    static int n;
    static double d,x;

    d=(a+b+c)*8192;
    x=fmod(d,4294967291.0);
    a=b; b=c; c=x;
    if (x < (float)2147483648.)
        n= (int) x;
    else
        n= (int)(x-4294967296.);
    return (n*2.3283064365e-10 + 0.5);
}


/* Rozklad bazowy. Zmienna losowa o rozkladzie ciaglym jednostajnym
    na przedziale [0,1]. Wersja naiwna. */

void t3Init()
{
    Randomize();
}

double t3Jedn ()
{
    return rand()/(double)RAND_MAX;
}


/*
Zmienna losowa o rozkladzie podanym jako argument, okreslona na zbiorze liczb
calkowitych nieujemnych [0..n-1].
*/

unsigned ruletka (double* rozklad, unsigned n)
{
    double pom=0, suma;
    unsigned i;
```

```
    pom=jednostajny();
    for (i=0, suma=0; i<n-1; i++)
    {
        suma+=rozklad[i];
        if (pom < suma)
            return i;
    }
    return (n-1);
}

/*
Zmienna losowa o rozkladzie dyskretnym, rownomiernym na zbiorze liczb
calkowitych nieujemnych z zakresu [0..n-1]
*/
unsigned rownomierny (unsigned n)
{
    double pom=0, suma;
    unsigned i;

    if (n<=1)
    return 0;
    pom=jednostajny();
    for (i=0, suma=1.0/n; i<n; i++)
        if (pom < suma)
            return i;
        else
            suma +=1.0/n;
    return (n-1);

}

/* Generator rozkladu normalnego, opis - por Wieczorkowski, Zielinski
    "Komputerowe generatory liczb losowych", WNT, Warszawa, 1997 str. 78, algorytm 3.30. */

double t1Norm()
{
    double u,v, zakres, x, xx;


    zakres=sqrt(2/exp(1));

    while (1)
    {
        while (!(u=jednostajny()) );
        v=2*zakres*jednostajny()-zakres;
        x=v/u;
        xx=x*x;
        if (xx <= 2*(3-u*(4+u))) return x;
        else if (xx <= 2/u-2*u)
            if (xx <= -4*log(u) ) return x;
    }
    return 0;



}
/* Generator rozkladu normalnego, opis - por Wieczorkowski, Zielinski
    "Komputerowe generatory liczb losowych", WNT, Warszawa, 1997 str. 42. */
```

```
double g(double u)
{
    double t;

    t=sqrt(-2*log(u));
    return (t-
        (((((0.0000453642210148*t)+0.0204231210245)*t+0.342242088547)*t+1)*t+0.322232431088)/
        (((((0.0038560700634*t)+0.103537752850)*t+0.531103462366)*t+0.588581570495)*t+0.0993484626060)
        );
}
double t2Norm()
{
    double u;


    u=jednostajny();
    if (u<0.5 && u>1e-20)
        return g(u);
    else
        return (-g(1-u));

}
/* Generator rozkladu normalnego Box-Mueller, opis - por Wieczorkowski, Zielinski
    "Komputerowe generatory liczb losowych", WNT, Warszawa, 1997 str. 93. */

double t3Norm()
{
    double u,v;


    while(!(u=jednostajny()));
    v=jednostajny();
    return (sqrt(-2*log(u))*cos(2*3.141592667597*v));

}
/* Zmienna losowa o rozkladzie normalnym standaryzowanym. Opis - por. Pacut
    "Prawdopodobienstwo - Teoria - Modelowanie probabilistyczne w technice",
    WNT Warszawa, 1985, str. 514. */

#define N 12
double t4Norm ()
{
    unsigned i;
    double suma;

    for (i=0; i<N ; i++)
        suma+=sqrt(12/N)*(jednostajny()-0.5);
    return suma;
}

//double normalny(double mu, double sigma)
//{
// return (normstd()*sigma+mu);
//}
/*
void normalny(double* X, const double* Mu, const double* Sigma, const unsigned n)
{
    double *Kier, Suma, Ssigma, N;
    unsigned i;
```

```
    Kier=(double*) calloc(n, sizeof(double));
    for (i=0, Suma=0; i<n; i++)
    {
        Kier[i]=normstd();
        Suma+=(Kier[i])*(Kier[i]);
    }
    Suma=sqrt(Suma);
    for (i=0; i<n; i++)
        X[i]=Mu[i];

    if (Suma>1e-10)
    {
        for (i=0, Ssigma=0; i<n; i++)
        {
            Kier[i]/=Suma;
            Ssigma+=Sigma[i]*Sigma[i]*Kier[i]*Kier[i];
        }
        Ssigma=sqrt(Ssigma);
        N=Ssigma*normstd();
        for (i=0; i<n; i++)
            X[i]+=N*Kier[i];
    }
    free(Kier);
}
*/
void normalny(double* X, const double* Mu, const double* Sigma, const unsigned n)
{
    unsigned i;

    for (i=0; i<n; i++)
        X[i]=Mu[i]+Sigma[i]*normstd();
}


/*
Zmienna losowa o rozkladzie prostokatnym na zbiorze liczb
rzeczywistych z zakresu [a .. b]
*/
double prostokatny (double a, double b)
{
    return (jednostajny()*(b-a) + a);
}

/* Losowa permutacja zbioru n kolejnych liczb nieujemnych z zakresu [0..n-1]
W tablicy permutacja jest umieszczona wynikowa permutacja
*/

void permutacja (unsigned* permutacja, unsigned n)
{
    unsigned i,j;
    unsigned pom;

    for(i=0; i<n; i++)
        permutacja[i]=i;
    for(i=n-1; i>0; i--)
    {
        j=rownomierny(i);
        if (i!=j)
        {
            pom=permutacja[j];
```

```
            permutacja[j]=permutacja[i];
            permutacja[i]=pom;
        }
    }

}


/*
Zmienna losowa o rozkladzie dwupunktowym, zwracajaca
1 z pstwem jeden i 0 z pstwem 1-jeden
*/
char dwupunktowy (double jeden)
{
    if (jednostajny() < jeden )
        return 1;
    else
        return 0;

}



/*
Zmienna losowa o n-wymiarowym rozkladzie jednostajnym, kazdy z wymiarow
ma zmiennosc od 0 do 1
*/
void jednNWym (double *wektor, unsigned n)
{
    unsigned i;
    for (i=0; i<n; i++)
        wektor[i]=jednostajny();
}

/*
Zmienna losowa o n-wymiarowym rozkladzie jednostajnym.
Suma wartosci jest rowna 1.
*/
void normJednNWym (double *wektor, unsigned n)
{
    unsigned i;
    double suma;

    do{
        for (i=0, suma=0; i<n; i++)
            suma+=(wektor[i]=jednostajny());
    }while(suma==0);

    for (i=0; i<n; i++)
        wektor[i]/=suma;
}



/*
Zmienna losowa o n-wymiarowym rozkladzie jednostajnym, kazdy z wymiarow
ma zmiennosc od 0 do k
*/
void dyskNWym (unsigned *wektor, unsigned n, unsigned k)
{
```

```
    unsigned i;
    for (i=0; i<n; i++)
        wektor[i]=rownomierny(k);
}

/* stara wersja
void kombinacja (unsigned* Wynik, unsigned n, unsigned k)
{
    unsigned i,j;
    unsigned Pom;
    unsigned *Permutacja;

    Permutacja = (unsigned*) calloc(n,sizeof(unsigned));

    for(i=0; i<n; i++)
        Permutacja[i]=i;
    for(i=n-1; i>n-k; i–)
    {
        j=rownomierny(i);
        if (i!=j)
        {
      Pom=Permutacja[j];
      Permutacja[j]=Permutacja[i];
      Permutacja[i]=Pom;
        }
    }
    memccpy(Wynik, Permutacja+n-k, k, sizeof(unsigned));
    free(Permutacja);
}
*/

/*
void kombinacja (unsigned* Wynik, unsigned n, unsigned k)
{
    unsigned i,j,l;
    unsigned Pom;
    unsigned *Uzyte;

    Uzyte = (unsigned*) calloc(n,sizeof(unsigned));

    for(i=0; i<n; i++)
        Uzyte[i]=0;
    for (i=0, j=0; i<k; i++)
    {
        l=rownomierny(n)+1;
        do{
    j++;
    while (Uzyte[j]) j=(j+1)%n;
    l–;
        }while (l>0);
        Uzyte[j]=1;
        Wynik[i]=j;
    }
    free(Uzyte);
}

*/
void kombinacja (unsigned* Wynik, unsigned n, unsigned k)
{
    unsigned i,j,l;
```

```
    unsigned Pom;
    unsigned *Uzyte;

    Uzyte = (unsigned*) calloc(n,sizeof(unsigned));

    for(i=0; i<n; i++)
        Uzyte[i]=i;
    for (j=0; j<k; j++)
    {
        l=rownomierny(n-j);
       Wynik[j]=Uzyte[l];
       Pom=Uzyte[n-1-j];
        Uzyte[n-1-j]=Uzyte[l];
        Uzyte[l]=Pom;
    }
    free(Uzyte);
}

/* Generator rozkladu Cauchy'ego, opis - por Wieczorkowski, Zielinski
    "Komputerowe generatory liczb losowych", WNT, Warszawa, 1997 str. 91. */

double cauchy()
{
    double u,x;

    x=jednostajny()*2-1;
    u=jednostajny();
    if ((u+0.27324)*(1+x*x) > 1.27324)
        x=jednostajny()*2-1;
    if (jednostajny()>0.5)
        return x;
    else
        if (x!=0)
            return (1/x);
        else
            return MAXDOUBLE;
}

#define LOS_PAR 4
void okreslLosowe (FILE* plikKonf, FILE* plikKontr)
{
    char Wczytanych;
    char *Nazwy[LOS_PAR];
    CZYTFUN czytFun[LOS_PAR];
    void *Adresy[LOS_PAR];
    unsigned *Liczba[LOS_PAR];

    Losowe = (LOSOWE )malloc(sizeof(LOSOWES));
    assert(Losowe!=NULL);

    Nazwy[0]="tjednostajny"; czytFun[0]=czytUnsigned; Adresy[0]=&(Losowe->tJednostajny);
Liczba[0]=NULL;
    Nazwy[1]="tnormalny"; czytFun[1]=czytUnsigned; Adresy[1]=&(Losowe->tNormalny);
Liczba[1]=NULL;
    Nazwy[2]="tinicjacji"; czytFun[2]=czytUnsigned; Adresy[2]=&(Losowe->tInicjacji);
Liczba[2]=NULL;
    Nazwy[3]="initziarno"; czytFun[3]=czytUnsigned; Adresy[3]=&(Losowe->InitZiarno);
Liczba[3]=NULL;

    Wczytanych = wczytajSymbole (plikKonf, plikKontr, Nazwy, czytFun, Adresy, Liczba, LOS_PAR);
```

```
/* assert (Wczytanych==LOS_PAR);*/

}
```

6.5 Plik `monitor.c`

Plik ten zawiera funkcje wykorzystywane w monitorze populacji do oceniania stopnia zróżnicowania populacji. Są to następujące funkcje;

- `double popDyskrepancja(POP Pop)`

  Funkcja ta oblicza przybliżoną miarę zróżnicowania wykorzystującą nieważoną dyskrepancję chromosomów osobników znajdujacych się w populacji podanej jako argument wywołania. Zakłada się kodowanie rzeczywistoliczbowe. Miara ta jest równa dyskrepancji populacji podzielonej przez dyskrepancję jednego punktu — centroidu populacji. Wymagane jest, aby były zdefiniowane ograniczenia kostkowe. Metoda polega na generowaniu losowych prostokątów wycinających fragment z prostopadłościanu określonego ograniczeniami kostkowymi i określeniu jakości przybliżenia miary tego podzbioru poprzez stosunek liczby chromosomów należących do jego wnętrza do liczby wczystkich chromosomów.

- `double popSchematy(POP Pop)` Funkcja ta oblicza przybliżenie ilości informacji zawartej w populacji z całkowitoliczbowym kodowaniem. Metoda polega na generowaniu losowych schematów i obliczaniu liczby osobników do nich pasujących. Różnica między stosunkiem tej liczby do liczby wszystkich osobników oraz prawdopodobieństwem wygenerowania losowego osobnika pasującego do tego schematu jest sumowana i świadczy o ilości informacji zwartej w populacji bazowej.

6.5.1 `monitor.h`

```
/*
+----------------------------+
| (C) Jaroslaw Arabas, listopad 1998 |
+----------------------------+
*/

#ifndef MONITOR_H
#define MONITOR_H
#include "osobnik.h"

typedef struct{
    unsigned Maksimum;
    unsigned Minimum;
    OSOBNIK Sredni;
    double Zroznicowanie;
} MONPOP;

typedef struct{
    OSOBNIK Rozwiazanie;
    double Poprawa;
    unsigned LPrzystosowan;
    unsigned Generacja;
    double *OkienkoGen;
    double *OkienkoPrzyst;

} MONZAD;


MONPOP* inicjujMonPop();

void usunMonPop(MONPOP*);
```

```
MONZAD* inicjujMonZad();

void usunMonZad (MONZAD*);
#endif
```

6.5.2 monitor.c

```
#include <math.h>
#include <assert.h>
#include <stdlib.h>

#include "defs.h"
#include "pop.h"
#include "losowe.h"
#include "monitor.h"
#include "stop.h"
#include "zadanie.h"

/*ziarnistosc z jaka przyblizamy dyskrepancje - ile pomiarow w
  proporcji do liczby Zmiennych*/

#define DYSK_DYSK 2000

#ifdef CIAGLE

/*dyskrepancja populacji osobnikow kodowanych rzeczywistoliczbowo*/
double popZroznicowanie(POP Pop)
{
    unsigned i,j,k;
    double *t, *centroid;
    double Wynik, Wynik2;
    double Pom ;
    char Wewnatrz, Wewnatrz2;
    double Objetosc;

    t=(double*)calloc(Zadanie->nZmiennych, sizeof(double));
    centroid=(double*)calloc(Zadanie->nZmiennych, sizeof(double));

    for (i=0; i<Zadanie->nZmiennych; i++)
        centroid[i]=0;
    for (k=0; k<Pop->nOsobnik; k++)
        for (i=0; i<Zadanie->nZmiennych; i++)
            centroid[i]+=Pop->Osobniki[k]->Genotyp[i];
    for (i=0; i<Zadanie->nZmiennych; i++)
        centroid[i]/=Pop->nOsobnik;

    Wynik=0; Wynik2=0;//Pom2=0;
    for (j=0; j<DYSK_DYSK*Zadanie->nZmiennych; j++)
    {
        Objetosc=1;
        for (i=0; i<Zadanie->nZmiennych; i++)
        {
            t[i]=prostokatny(Zadanie->DolneX[i], Zadanie->GorneX[i]);
            Objetosc* =(t[i]-Zadanie->DolneX[i])/(Zadanie->GorneX[i]-Zadanie->DolneX[i]);
        }
        for (Pom=0,k=0; k<Pop->nOsobnik; k++)
        {
            Wewnatrz=1;
            for (i=0; i<Zadanie->nZmiennych && Wewnatrz; i++)
```

```
                if (Pop→Osobniki[k]→Genotyp[i]>t[i])
                    Wewnatrz=0;
            if (Wewnatrz)
                Pom+=1;
        }
        Wewnatrz2=1;
        for (i=0; i<Zadanie→nZmiennych && Wewnatrz2; i++)
            if (centroid[i]>t[i])
                Wewnatrz2=0;
        Pom/=Pop→nOsobnik;
        Pom=Objetosc-Pom;
        Wynik+=Pom*Pom;
        if (Wewnatrz2)
            Wynik2+=((Objetosc-1)*(Objetosc-1));
        else
            Wynik2+=Objetosc*Objetosc;
    }
    free (t);
    free (centroid);
    //Wynik=log(sqrt(Wynik/(DYSK_DYSK*Zadanie-
>nZmiennych))/sqrt(Pom2/(DYSK_DYSK*Zadanie->nZmiennych)));
    Wynik=sqrt(Wynik/(DYSK_DYSK*Zadanie→nZmiennych));
    Wynik2=sqrt(Wynik2/(DYSK_DYSK*Zadanie→nZmiennych));
    Wynik=log(Wynik/Wynik2);

    return Wynik;
}

#else

/* cos na ksztalt entropii populacji osobnikow kodowanych dyskretnie
metoda polega na zliczaniu schematow*/

double entropia (unsigned* Tablica, unsigned N)
{
    unsigned i;
    double Suma, Wynik, Pom;

    for (i=0, Suma=0; i<N; i++)
        Suma+=Tablica[i];
    for (i=0, Wynik=0; i<N; i++)
    {
        if (Suma>0)
        {
            Pom=Tablica[i]/Suma;
            Wynik+=(Pom*log(Pom));
        }
    }
    return Wynik;
}

double popZroznicowanie(POP Pop)
{
    unsigned i,j,k;
    unsigned Istotne;
    unsigned *Znaki;
    unsigned *Schemat;
    double *Entropia;
    double Wynik1, Wynik;
```

```
    Schemat=(unsigned*)calloc(Zadanie->nZmiennych, sizeof(unsigned));
    Znaki=(unsigned*)calloc(Zadanie->nLiter, sizeof(unsigned));
    Entropia=(double*)calloc(Zadanie->nZmiennych, sizeof(double));


    for (j=0, Wynik=0; j<DYSK_DYSK*Zadanie->nZmiennych; j++)
    {
        for (i=0; i<Zadanie->nZmiennych; i++)
            Schemat[i]=rownomierny(Zadanie->nLiter+1);
        for (i=0, Wynik1=0, Istotne=0;i<Zadanie->nZmiennych; i++)
        {
            if (Schemat[i]!=Zadanie->nLiter)
            {
                for (k=0; k<Zadanie->nLiter; k++)
                    Znaki[k]=0;
                for (k=0; k<Pop->nOsobnik; k++)
                    Znaki[Pop->Osobniki[k]->Genotyp[i]]+=1;
                Wynik1+=entropia(Znaki, Zadanie->nLiter);
                Istotne++;
            }
        }

        if (Istotne!=0)
            Wynik+=Wynik1/Istotne; /* entropia istotnych miejsc schematu */
    }
    free (Schemat);
    free (Znaki);
    free (Entropia);
    return Wynik;
}


#endif
MONPOP* inicjujMonPop()
{
    MONPOP* Wynik;
    Wynik=(MONPOP*)malloc(sizeof(MONPOP));
    assert(Wynik!=NULL);
    Wynik->Sredni = osNowy();
    return Wynik;
}

void usunMonPop(MONPOP* Usuwany)
{
    osUsun(&(Usuwany->Sredni));
    free (Usuwany);
}


MONZAD* inicjujMonZad()
{
    MONZAD* Wynik;

    Wynik=(MONZAD*)malloc(sizeof(MONZAD));
    assert(Wynik!=NULL);
    Wynik->Rozwiazanie=NULL;
    Wynik->LPrzystosowan=0;
    Wynik->OkienkoGen = (double*)calloc(Stop->SzerStop, sizeof(double));
    Wynik->OkienkoPrzyst= (double*)calloc(Stop->SzerStop, sizeof(double));
    return Wynik;
```

```
}

void usunMonZad (MONZAD* Usuwany)
{
    free(Usuwany->OkienkoGen);
    free(Usuwany->OkienkoPrzyst);
    free(Usuwany);
}
```

6.6 Plik `ogranicz.c`

Plik ten zawiera implementacje metod uwzględniania ograniczeń w algorytmie ewolucyjnym. Przyjeto zostało, że ewaluacja każdego osobnika jest wykonywana przez funkcję uwzględniającą ograniczenia, która dopiero wywołuje funkcję celu.

Zaimplementowane zostały następujące metody

- brak ograniczeń

  W tym przypadku funkcja uwzględniająca ograniczenia jest „przezroczysta" — sprowadza się do obliczenia funkcji celu i sprowadzeniu jej do funkcji przystosowania (wymnożenie przez -1 w przypadku, gdy funkcja celu podlega minimalizacji).

- ograniczenia uwzględniane przez zewnętrzną funkcję kary

  Przystosowanie osobnika jest obliczane jako ważona suma wartości funkcji celu oraz wartości funkcji kary. W wersji podstawowej została zaimplementowana liniowa funkcja kary.

- ograniczenia uwzględniane przez metodę naprawy

  W tej metodzie dokonuje się sprawdzenia, czy osobnik ma genotyp znajdujący się w obszarze dopuszczalnym. Jeśli nie, uruchamiana jest funkcja naprawy, która tworzy nowy fenotyp. Następnie podejmowana jest decyzja o sposobie wykorzystania tego fenotypu. Z prawdopodobieństwem równym `Lamarkizm` (nazwa pola struktury OGRANICZENIAS) fenotyp ten jest przekazywany osobnikowi (i będzie podlegał dziedziczeniu); zdarzeniem przeciwnym jest pominięcie tego fenotypu (ewolucja Darwinowska z efektem Baldwina). W obu przypdakach, wartością funkcji przystosowania osobnika staje się wartośc obliczona dla „naprawionego" fenotypu.

6.6.1 `ogranicz.h`

```
/*
+——————————————-+
| (C) Jaroslaw Arabas, luty 1999 |
+——————————————-+
*/

#ifndef OGRANICZ_H
#define OGRANICZ_H

typedef struct{
unsigned tOgraniczen;
unsigned tFKary;

double Lamarkizm;
double WagaOgr;
} OGRANICZENIAS;

typedef OGRANICZENIAS* OGRANICZENIA;

void okreslOgraniczenia(FILE*,FILE*);

extern OGRANICZENIA Ograniczenia;

#endif
```

6.6.2 ogranicz.c

```
/*
+--------------------------------+
| (C) Jaroslaw Arabas, listopad 1998 |
+--------------------------------+
*/

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

#include "zadanie.h"
#include "osobnik.h"
#include "losowe.h"
#include "ogranicz.h"
#include "wewy.h"


OGRANICZENIA Ograniczenia;

double ogrBrak (OSOBNIK Osobnik);
double fKary (double Przekroczenie);
double ogrZKara (OSOBNIK Osobnik);
double ogrNaprawa (OSOBNIK Osobnik);

double fKLin (double Przekroczenie);


double fPrzystosowania (OSOBNIK Osobnik)
{
    switch (Ograniczenia->tOgraniczen)
    {
    case 1: return ogrBrak(Osobnik);
    case 2: return ogrZKara(Osobnik);
    case 3: return ogrNaprawa(Osobnik);
    default: return -1;
    }
}

/* brak ograniczen - funkcja przystosowania tozsama z funkcja celu (z dokladoscia
do znaku */
double ogrBrak (OSOBNIK Osobnik)
{
    Osobnik->Przystosowanie=-Zadanie->Minimalizacja*(*(Zadanie->fcelu))(Osobnik->Genotyp,
Osobnik->nGenow);
    return Osobnik->Przystosowanie;
}


/* Zewnetrzna funkcja kary - zalezna od wartosci przekroczen, nie odleglosci od
    obszaru dopuszczalnego*/

double fKary (double Przekroczenie)
{
    switch (Ograniczenia->tFKary)
    {
    case 1: return fKLin (Przekroczenie);
    default: return 0;
    }
```

```
}

/* Liniowa funkcja */
double fKLin (double Przekroczenie)
{
    return Przekroczenie ;
}

double ogrZKara (OSOBNIK Osobnik)
{
    unsigned i;
    double Przekroczenie;
    Osobnik->Przystosowanie=-Zadanie->Minimalizacja*(*(Zadanie->fcelu))(Osobnik->Genotyp,
Osobnik->nGenow);
    for (i=0; i<Zadanie->nNierown; i++)
    {
        Przekroczenie= (*(Zadanie->nierownosciowe[i]))(Osobnik->Genotyp, Osobnik->nGenow);
        if (Przekroczenie>0)
            Osobnik->Przystosowanie-=Ograniczenia->WagaOgr*fKary(Przekroczenie);
    }
/* if (Osobnik->Przystosowanie<0)
        Osobnik->Przystosowanie=0;*/
    return Osobnik->Przystosowanie;
}


/* Funkcja naprawy - wyrzucenie z plecaka losowych przedmiotow */
void naprawa (TGEN *plecak, unsigned n, FUNKCJA *ograniczenie, unsigned nOgraniczen)
{
    unsigned i, j;

    while ((*ograniczenie[0])(plecak, n)>0)
    {
        j=rownomierny(n);
        for (i=0; j>0; j>0 ? i=(++i)%n : i)
            if (plecak[i]!=0)
                j--;
        plecak[i]=0;
    }
}

double ogrNaprawa (OSOBNIK Osobnik)
{

    TGEN *Fenotyp;
    unsigned i;

    Fenotyp=(TGEN*)calloc(Osobnik->nGenow, sizeof (TGEN));
    for(i=0; i<Osobnik->nGenow; i++)
        Fenotyp[i]=Osobnik->Genotyp[i];

    naprawa(Fenotyp, Osobnik->nGenow, Zadanie->nierownosciowe, Zadanie->nNierown);
    Osobnik->Przystosowanie=-Zadanie->Minimalizacja*(*(Zadanie->fcelu))(Fenotyp, Osobnik->nGenow);

    if (dwupunktowy(Ograniczenia->Lamarkizm))
    {
        free (Osobnik->Genotyp);
        Osobnik->Genotyp=Fenotyp;
    }
    else
```

```
        free (Fenotyp);
    return Osobnik→Przystosowanie;
}
#define OGR_PAR 4
void okreslOgraniczenia(FILE* plikKonf,FILE* plikKontr)
{
    char Wczytanych;
    char *Nazwy[OGR_PAR];
    CZYTFUN czytFun[OGR_PAR];
    void *Adresy[OGR_PAR];
    unsigned *Liczba[OGR_PAR];

    Ograniczenia = (OGRANICZENIA )malloc(sizeof(OGRANICZENIAS));
    assert(Ograniczenia!=NULL);

    Nazwy[0]="tograniczen"; czytFun[0]=czytUnsigned; Adresy[0]=&(Ograniczenia→tOgraniczen);
Liczba[0]=NULL;
    Nazwy[1]="tfkary"; czytFun[1]=czytUnsigned; Adresy[1]=&(Ograniczenia→tFKary);
Liczba[1]=NULL;
    Nazwy[2]="lamarkizm"; czytFun[2]=czytDouble; Adresy[2]=&(Ograniczenia→Lamarkizm);
Liczba[2]=NULL;
    Nazwy[3]="wagaogr"; czytFun[3]=czytDouble; Adresy[3]=&(Ograniczenia→WagaOgr);
Liczba[3]=NULL;

    Wczytanych = wczytajSymbole (plikKonf, plikKontr, Nazwy, czytFun, Adresy, Liczba, OGR_PAR);

/* assert (Wczytanych==OGR_PAR);*/
}
```

6.7 Plik `osobnik.c`

W tym pliku znajdują się funkcje pomocnicze zarządzające osobnikiem.

6.7.1 `osobnik.h`

```
/*
+—————————————-+
| (C) Jaroslaw Arabas, luty 1999 |
+—————————————-+
*/

#ifndef OSOBNIK_H
#define OSOBNIK_H

#include "defs.h"

typedef struct
    {
        unsigned nGenow;
        TGEN* Genotyp;
        unsigned nPar;
        double* Parametry;
        double Przystosowanie;
        unsigned MaxWiek;

        /*dla celow monitorowania*/
        unsigned Wiek;
        unsigned LKopii;
    } OSOBNIKS;
```

```
typedef OSOBNIKS* OSOBNIK;

/* Funkcje pomocnicze - zarzadzanie pamiecia OSOBNIKa
*/
OSOBNIK osPrzesun(OSOBNIK *Osobnik);
OSOBNIK osKopiuj (OSOBNIK Osobnik);
OSOBNIK osKopiuj2 (OSOBNIK Osobnik);
void osUsun (OSOBNIK* Osobnik);
OSOBNIK osNowy ();
char wKostkowych(OSOBNIK Osobnik);

#endif
```

6.7.2 osobnik.c

```
/*
+——————————————-+
| (C) Jaroslaw Arabas, luty 1999 |
+——————————————-+
*/

#include <math.h>
#include <memory.h>
#include <stdlib.h>

#include "algorytm.h"
#include "losowe.h"
#include "osobnik.h"
#include "zadanie.h"


OSOBNIK osPrzesun(OSOBNIK *Osobnik)
{
    OSOBNIK Wynik = *Osobnik;

    *Osobnik = NULL;
    return Wynik;
}

OSOBNIK osKopiuj (OSOBNIK Osobnik)
{
    OSOBNIK Nowy = NULL;
    unsigned i;

    Nowy = osNowy();
    for(i=0; i<Osobnik->nGenow; i++)
        Nowy->Genotyp[i]=Osobnik->Genotyp[i];
    if (Osobnik->nPar >0)
        for(i=0; i<Osobnik->nPar; i++)
            Nowy->Genotyp[i]=Osobnik->Parametry[i];
    Nowy->Przystosowanie=Osobnik->Przystosowanie;
    Osobnik->LKopii++;
    return Nowy;
}


void osUsun (OSOBNIK* Osobnik)
```

```
{
    free ((*Osobnik)->Genotyp);
    if ((*Osobnik)->nPar>0)
        free ((*Osobnik)->Parametry);
    free (*Osobnik);
    *Osobnik=NULL;

}

OSOBNIK osNowy ()
{
    OSOBNIK Nowy;
    unsigned i;

    Nowy = (OSOBNIK)malloc (sizeof(OSOBNIKS));
    Nowy->nGenow = Zadanie->nZmiennych;
    if (Algorytm->Samoadaptacja)
    {
        Nowy->nPar = Zadanie->nZmiennych;
        Nowy->Parametry = (double*)calloc (Nowy->nPar, sizeof(double));
        for (i=0; i<Nowy->nPar; i++)
            Nowy->Parametry[i]=prostokatny(0,(Zadanie->GorneX[i]-
Zadanie->DolneX[i])/(3*Zadanie->nZmiennych));
    }
    else
        Nowy->nPar = 0;
    Nowy->Genotyp = (TGEN*)calloc (Nowy->nGenow, sizeof(TGEN));
    Nowy->Wiek = 0;
    Nowy->LKopii=0;
    return Nowy;
}


char wKostkowych(OSOBNIK Osobnik)
{
    unsigned i;
    if (!Zadanie->saKostkowe)
        return 1;
    else
        for (i=0; i<Osobnik->nGenow; i++)
        {
            if (Osobnik->Genotyp[i] > Zadanie->GorneX[i])
                return 0;
            if (Osobnik->Genotyp[i] < Zadanie->DolneX[i])
                return 0;
        }
    return 1;

}

OSOBNIK osKopiuj2 (OSOBNIK Osobnik)
{
    OSOBNIK Wynik;
    Wynik=osKopiuj(Osobnik);
    Osobnik->LKopii--;
    return Wynik;
}
```

6.8 Plik `pop.c`

Plik ten zawiera funckje pomocnicze zarządzające populacją osobników. Znajdują się tu również implementacje funkcji uaktualniających zawartości monitorów — `void popMonitor(POP Pop)` oraz `void zadMonitor(unsigned t, POP Pt)`.

6.8.1 `pop.h`

```
/*
+-------------------------------+
| (C) Jaroslaw Arabas, listopad 1998 |
+-------------------------------+
*/

#ifndef POP_H
#define POP_H

#include "monitor.h"
#include "osobnik.h"

/* Populacja jest reprezentowana jako wskaznik do struktury
*/
typedef struct
{
    unsigned nOsobnik;
    OSOBNIK* Osobniki;
    MONPOP* Monitor;
} POPS;

typedef POPS* POP;

/* sortowanie populacji - wedlug kryterium optymalizacji */
void popSortuj (POP Populacja);

/* permutowanie populacji - ustawienie osobnikow w porzadku losowym */
void popPermutuj (POP Populacja);

/* ewaluacja populacji - obliczenie wartosci przystosowania dla wszystkich
    osobnikow populacji */
void popEwaluacja (POP Populacja);

/* inicjacja populacji - utworzenie i wypelnienie jej nowymi osobnikami */
void popInic(POP Populacja);

/* inicjacja populacji - utworzenie i wypelnienie jej nowymi osobnikami */
void popOsobniki(POP Populacja);

/* usuniecie osobnikow z populacji */
void popCzysc(POP Populacja);

POP popNowa(unsigned Licznosc);

void popUsun(POP Populacja);

void popMonitor(POP Populacja);

double popZroznicowanie(POP Pop);

void zadMonitor(unsigned t,POP Populacja);
```

```
#endif
```

6.8.2 pop.c

```
/*
+---------------------------------+
| (C) Jaroslaw Arabas, listopad 1998 |
+---------------------------------+
*/

#include <assert.h>
#include <stdlib.h>
#include <math.h>
//#include <values.h>
#include "losowe.h"
#include "osobnik.h"
#include "pop.h"
#include "stop.h"
#include "zadanie.h"
#include "monitor.h"

/* sortowanie populacji - wedlug kryterium optymalizacji */

int porownaj (const void *C1, const void *C2)
{
    if ( (*(OSOBNIK*)C2)->Przystosowanie - (*(OSOBNIK*)C1)->Przystosowanie > 0 )
        return 1;
    else
        return -1;
}

void popSortuj (POP Populacja)
{
    qsort(Populacja->Osobniki, Populacja->nOsobnik, sizeof(OSOBNIK), porownaj);
}

/* permutowanie populacji - ustawienie osobnikow w porzadku losowym */
void popPermutuj (POP Populacja)
{
    unsigned *Permut;
    OSOBNIK *Ustawienie;
    unsigned i;

    Permut = (unsigned*)calloc (Populacja->nOsobnik, sizeof(unsigned));
    permutacja (Permut, Populacja->nOsobnik);
    Ustawienie = (OSOBNIK*)calloc (Populacja->nOsobnik, sizeof(OSOBNIK));
    for (i=0; i<Populacja->nOsobnik; i++)
        Ustawienie[i] = Populacja->Osobniki[Permut[i]];
    free (Populacja->Osobniki);
    Populacja->Osobniki=Ustawienie;
    free (Permut);
}



/* ewaluacja populacji - obliczenie wartosci przystosowania dla wszystkich
    osobnikow populacji */
void popEwaluacja (POP Populacja)
{
```

```
    unsigned i;

    for (i=0; i<Populacja→nOsobnik; i++)
    {
        if (wKostkowych(Populacja→Osobniki[i]))
        {
            fPrzystosowania (Populacja→Osobniki[i]);
            Zadanie→Monitor→LPrzystosowan++;
        }
        else
            Populacja→Osobniki[i]→Przystosowanie=SMIERC;
    }


}

POP popNowa(unsigned nOsobnik)
{
    POP Wynik;

    Wynik=(POP)malloc(sizeof(POPS));
    Wynik→nOsobnik=nOsobnik;
    Wynik→Osobniki=(OSOBNIK*)calloc(nOsobnik, sizeof(OSOBNIK));
    Wynik→Monitor = inicjujMonPop();
    return Wynik;
}


/* inicjacja populacji - wypelnienie jej nowymi osobnikami */
void popOsobniki(POP Populacja)
{
    unsigned i;

    for(i=0; i<Populacja→nOsobnik; i++)
        Populacja→Osobniki[i]=osNowy();

}

/* inicjacja populacji - wypelnienie jej nowymi osobnikami */
void popInic (POP Populacja)
{
    unsigned i;
    TGEN **Genotypy;

    Genotypy=(TGEN**)calloc(Populacja→nOsobnik, sizeof(TGEN*));
    assert(Genotypy!=NULL);
    for(i=0; i<Populacja→nOsobnik; i++)
        Genotypy[i]=Populacja→Osobniki[i]→Genotyp;
    zadInicjuj(Genotypy, Populacja→nOsobnik);
    free(Genotypy);
}

/* usuniecie osobnikow z populacji */
void popCzysc(POP Populacja)
{
    unsigned i;

    for (i=0;i<Populacja→nOsobnik; i++)
        if (Populacja→Osobniki[i] !=NULL)
            osUsun(&(Populacja→Osobniki[i]));
```

```
}

void popUsun(POP Populacja)
{
    popCzysc(Populacja);
    free(Populacja->Osobniki);
    usunMonPop(Populacja->Monitor);
    free(Populacja);
}

void popMonitor(POP Pop)
{
    unsigned i,j;
    unsigned Zywi;

    /* Wykonanie statystyk populacji */
    Pop->Monitor->Maksimum=0;
    Pop->Monitor->Minimum=0;
    Pop->Monitor->Sredni->Przystosowanie=0;
    Pop->Monitor->Zroznicowanie=0;
#ifdef CIAGLE
    for (j=0; j<Pop->Monitor->Sredni->nGenow; j++)
        Pop->Monitor->Sredni->Genotyp[j]=0;
#endif
    for (j=0; j<Pop->Monitor->Sredni->nPar; j++)
        Pop->Monitor->Sredni->Parametry[j]=0;
    for (i=0, Zywi=0; i<Pop->nOsobnik; i++)
    {

#ifdef CIAGLE
        for (j=0; j<Pop->Monitor->Sredni->nGenow; j++)
            Pop->Monitor->Sredni->Genotyp[j]+=Pop->Osobniki[i]->Genotyp[j];
#endif

            for (j=0; j<Pop->Monitor->Sredni->nPar; j++)
            Pop->Monitor->Sredni->Parametry[j]+=Pop->Osobniki[i]->Parametry[j];

        if (Pop->Osobniki[i]->Przystosowanie!=SMIERC)
        {
            if (Pop->Osobniki[i]->Przystosowanie >
Pop->Osobniki[Pop->Monitor->Maksimum]->Przystosowanie)
                Pop->Monitor->Maksimum=i;
            if (Pop->Osobniki[i]->Przystosowanie <
Pop->Osobniki[Pop->Monitor->Minimum]->Przystosowanie)
                Pop->Monitor->Minimum=i;
            Pop->Monitor->Sredni->Przystosowanie+=Pop->Osobniki[i]->Przystosowanie;
            Zywi++;
        }

    }

#ifdef CIAGLE
    for (j=0; j<Pop->Monitor->Sredni->nGenow; j++)
        Pop->Monitor->Sredni->Genotyp[j]/=Pop->nOsobnik;
#endif

    for (j=0; j<Pop->Monitor->Sredni->nPar; j++)
        Pop->Monitor->Sredni->Parametry[j]/=Pop->nOsobnik;
```

```
    if (Zywi!=0)
    {
        Pop→Monitor→Sredni→Przystosowanie/=Zywi;
/* Pop->Monitor->Zroznicowanie=popZroznicowanie(Pop);*/
    }
    else
        Pop→Monitor→Sredni→Przystosowanie=SMIERC;
}

void zadMonitor(unsigned t, POP Pt)
{
    unsigned i;
    /* Wykonanie statystyk algorytmu */
    Zadanie→Monitor→Generacja=t;
    for (i=1; i<Stop→SzerStop; i++)
        Zadanie→Monitor→OkienkoGen[i-1]=Zadanie→Monitor→OkienkoGen[i];


    if (Zadanie→Monitor→OkienkoGen[Stop→SzerStop-1]<
        Pt→Osobniki[Pt→Monitor→Maksimum]→Przystosowanie || Zadanie→Monitor→Generacja ==0)
    {
        if (Zadanie→Monitor→Rozwiazanie!=NULL)
            osUsun(&(Zadanie→Monitor→Rozwiazanie));
        Zadanie→Monitor→Rozwiazanie=osKopiuj2(Pt→Osobniki[Pt→Monitor→Maksimum]);
        Zadanie→Monitor→OkienkoGen[Stop→SzerStop-
1]=Zadanie→Monitor→Rozwiazanie→Przystosowanie;
    }
    else
        Zadanie→Monitor→Rozwiazanie→Wiek++;


}
```

6.9 Plik `raporty.c`

W tym pliku zaimplementowane zostały funkcje generujące raporty osobników, populacji i zadania.

6.9.1 `raporty.h`

```
/*
+————————————————-+
| (C) Jaroslaw Arabas, listopad 1998 |
+————————————————-+
*/

#ifndef RAPORTY_H
#define RAPORTY_H

#include "osobnik.h"
#include "pop.h"
#include "zadanie.h"
typedef struct{
char *RapGen;
/*RAPORT POPULACJI*/
```

```
/* dopuszczalne wartosci
o - Osobniki
z - Zroznicowanie
n - miNimum
s - Srednia
x - maXimum
*/
char* RapPop;
//#define POPULACJA "o\n—\n"

/*RAPORT KAZDEGO OSOBNIKA - DOTYCZY TAKZE
SREDNIEJ, MAKSIMUM, MINIMUM, ROZWIAZANIA*/
/* dopuszczalne wartosci
x - zmienne niezalezne X
p - Parametry
w - Wiek
k - liczba Kopii
f - wartosc Funkcji przystosowania
*/
char *RapOs;
//#define SREDNI "xpf"
char *RapSredni;
//#define MINIMUM "f"
char *RapMin;
//#define MAKSIMUM "f"
char *RapMax;

char *RapZad;
//#define ROZWIAZANIE "f"
char *RapRozw;
} RAPORTYS;

typedef RAPORTYS* RAPORTY;

extern RAPORTY Raporty;

void okreslRaporty(FILE*,FILE*);

void rapPop(POP Pop, char* Parametry);
void rapOsobnik(OSOBNIK Osobnik, char* Parametry);
void rapZadanie(ZADANIE Zadanie, char* Parametry);

#endif
```

6.9.2 raporty.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include "osobnik.h"
#include "pop.h"
#include "raporty.h"
#include "wewy.h"

RAPORTY Raporty;

char znakSpecjalny(char Nastepny)
{
```

```
    switch (Nastepny)
    {
        case 'n': return '\n';
        case '
': return '
';
        case 't': return '\t';
        default: return 0;
    }
}
void rapPop(POP Pop, char* ParZewn)
{
    unsigned i,j;
    char *Parametry;
    char Specjalny;

    if (ParZewn==NULL)
        Parametry=Raporty→RapPop;
    else
        Parametry=ParZewn;

    j=0;
    while (Parametry[j]!=0)
    {
        switch (Parametry[j])
        {
        case 'o':
        case 'O':
            for (i=0; i<Pop→nOsobnik; i++)
                rapOsobnik(Pop→Osobniki[i],Raporty→RapOs);
            break;
        case 'z':
        case 'Z':
            printf("%lf ", Pop→Monitor→Zroznicowanie);
            break;
        case 's':
        case 'S':
            rapOsobnik(Pop→Monitor→Sredni, Raporty→RapSredni);
            break;
        case 'x':
        case 'X':
            rapOsobnik(Pop→Osobniki[Pop→Monitor→Maksimum], Raporty→RapMax);
            break;
        case 'n':
        case 'N':
            rapOsobnik(Pop→Osobniki[Pop→Monitor→Minimum], Raporty→RapMin);
            break;
        case '~':
            printf(" ");
            break;
        case '
':
            Specjalny=znakSpecjalny(Parametry[++j]);
            if (Specjalny!=0)
                printf("%c",Specjalny);
            break;
        default:
            printf("%c",Parametry[j]);
            break;
        }
```

```
        j++;
    }

    fflush(stdout);
}
//void rapOsobnik(OSOBNIK Osobnik, const char* Parametry);

void rapOsobnik(OSOBNIK Osobnik, char* ParZewn)
{
    unsigned i,j;
    char *Parametry;
    char Specjalny;

    if (ParZewn==NULL)
        Parametry="";
    else
        Parametry=ParZewn;

    j=0;
    while (Parametry[j]!=0)
    {
        switch (Parametry[j])
        {
        case 'x':
        case 'X':
            for (i=0; i<Osobnik->nGenow; i++)
            {
                printf(Raporty->RapGen, Osobnik->Genotyp[i]);
                printf(" ");
             }
            break;
        case 'p':
        case 'P':
            for (i=0; i<Osobnik->nPar; i++)
                printf("%lf ", Osobnik->Parametry[i]);
            break;
        case 'w':
        case 'W':
            printf("%u ", Osobnik->Wiek);
            break;
        case 'k':
        case 'K':
            printf("%u ", Osobnik->LKopii);
            break;
        case 'f':
        case 'F':
            printf("%lf ", Osobnik->Przystosowanie);
            break;
        case '~':
            printf(" ");
            break;
        case '
':
            Specjalny=znakSpecjalny(Parametry[++j]);
            if (Specjalny!=0)
                printf("%c",Specjalny);
            break;
        default:
            printf("%c",Parametry[j]);
            break;
```

```
        }
        j++;
    }

    fflush(stdout);
}

void rapZadanie(ZADANIE Zadanie, char* ParZewn)
{
    unsigned j;
    char *Parametry;
    char Specjalny;

    if (ParZewn==NULL)
        Parametry=Raporty->RapZad;
    else
        Parametry=ParZewn;

    j=0;
    while (Parametry[j]!=0)
    {
        switch (Parametry[j])
        {
        case 't':
        case 'T':
            printf("%u ",Zadanie->Monitor->Generacja);
            break;
        case 'r':
        case 'R':
            rapOsobnik(Zadanie->Monitor->Rozwiazanie, Raporty->RapRozw);
            break;
        case '~':
            printf(" ");
            break;
        case '
':
            Specjalny=znakSpecjalny(Parametry[++j]);
            if (Specjalny!=0)
                printf("%c",Specjalny);
            break;
        default:
            printf("%c",Parametry[j]);
            break;
        }
        j++;
    }
    fflush(stdout);

}

#define RAP_PAR 8
void okreslRaporty(FILE* plikKonf,FILE* plikKontr)
{
    char Wczytanych;
    char *Nazwy[RAP_PAR];
    CZYTFUN czytFun[RAP_PAR];
    void *Adresy[RAP_PAR];
    unsigned *Liczba[RAP_PAR];

    Raporty=(RAPORTY)malloc(sizeof(RAPORTYS));
```

```
    assert(Raporty!=NULL);

    Nazwy[0]="rapgen"; czytFun[0]=czytLancuch; Adresy[0]=&(Raporty->RapGen); Liczba[0]=NULL;
    Nazwy[1]="rappop"; czytFun[1]=czytLancuch; Adresy[1]=&(Raporty->RapPop); Liczba[1]=NULL;
    Nazwy[2]="rapos"; czytFun[2]=czytLancuch; Adresy[2]=&(Raporty->RapOs); Liczba[2]=NULL;
    Nazwy[3]="rapsredni"; czytFun[3]=czytLancuch; Adresy[3]=&(Raporty->RapSredni);
Liczba[3]=NULL;
    Nazwy[4]="rapmin"; czytFun[4]=czytLancuch; Adresy[4]=&(Raporty->RapMin); Liczba[4]=NULL;
    Nazwy[5]="rapmax"; czytFun[5]=czytLancuch; Adresy[5]=&(Raporty->RapMax); Liczba[5]=NULL;
    Nazwy[6]="rapzad"; czytFun[6]=czytLancuch; Adresy[6]=&(Raporty->RapZad); Liczba[6]=NULL;
    Nazwy[7]="raprozw"; czytFun[7]=czytLancuch; Adresy[7]=&(Raporty->RapRozw); Liczba[7]=NULL;

    Wczytanych = wczytajSymbole (plikKonf, plikKontr, Nazwy, czytFun, Adresy, Liczba, RAP_PAR);

/* assert (Wczytanych==RAP_PAR);*/
}
```

6.10 Plik `selekcja.c`

Plik ten zawiera implementacje podstawowych metod reprodukcji i sukcesji opisywanych w książce. Rangi są nadawane podczas wywołania funkcji `void wyznaczRangi (unsigned *Rangi, unsigned *MaxRanga, unsigned nOsobnik, OSOBNIK *Osobniki);`. Wartość wskazywana przez parametr `MaxRanga` jest ustawiana przez funkcję nadającą rangi, gdyż niektóre metody reprodukcji rangowej (np. progowa) mogą wymagać podania tej wartości. Istnieje możliwość uogólnienia tej funkcji na przypadek optymalizacji wielokryterialnej.

6.10.1 `selekcja.h`

```
/*
+---------------------------+
| (C) Jaroslaw Arabas, luty 1999 |
+---------------------------+
*/

#ifndef SELEKCJA_H
#define SELEKCJA_H

typedef struct{
    unsigned tReprodukcji;
    unsigned tSukcesji;

    unsigned nTurnieju;
    double nElity;
    unsigned tRang;
    double obciecie;
    double fRWykladnik;
    unsigned maxWiek;
} SELEKCJAS;

typedef SELEKCJAS* SELEKCJA;

/* reprodukcja tworzy populacje pomocnicza z populacji P_t */
char reprodukcja(POP Pt, unsigned *Wybrani, const unsigned nRodzicow);
/* sukcesja tworzy populacje P_{t+1} z populacji O_t i P_t */
char sukcesja (POP Pnowa, POP Pt, POP Ot);

extern SELEKCJA Selekcja;

void okreslSelekcje(FILE*,FILE*);
```

```
#endif
```

6.10.2 selekcja.c

```
/*
+---------------+
| (C) Jaroslaw Arabas, luty 1999 |
+---------------+
*/

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

#include "pop.h"
#include "losowe.h"
#include "algorytm.h"
#include "raporty.h"
#include "selekcja.h"
#include "wewy.h"

SELEKCJA Selekcja;

char rPropMod (POP Pt, unsigned *Wybrani, unsigned nWybranych );
char rTurnZwr (POP Pt, unsigned *Wybrani, unsigned nWybranych );
char rRang (POP Pt, unsigned *Wybrani, unsigned nWybranych );
char rProsta (POP Pt, unsigned *Wybrani, unsigned nWybranych );
char rPropStd (POP Pt, unsigned *Wybrani, unsigned nWybranych );
char rTurnBZ (POP Pt, unsigned *Wybrani, unsigned nWybranych );

char sProsta (POP Pnowa, POP Pt, POP Ot);
char sElitaStd (POP Pnowa, POP Pt, POP Ot);
char sElitaCZ (POP Pnowa, POP Pt, POP Ot);

double fRWykl (unsigned Ranga, unsigned MaxRanga);
double fRProg (unsigned Ranga, unsigned MaxRanga);

char reprodukcja(POP Pt, unsigned *Rodzice, const unsigned nRodzicow)
{
   switch (Selekcja->tReprodukcji)
   {
   case 1: return rPropStd(Pt, Rodzice, nRodzicow);
   case 2: return rPropMod(Pt, Rodzice, nRodzicow);
   case 3: return rTurnZwr(Pt, Rodzice, nRodzicow);
   case 4: return rTurnBZ(Pt, Rodzice, nRodzicow);
   case 5: return rRang(Pt, Rodzice, nRodzicow);
   case 6: return rProsta(Pt, Rodzice, nRodzicow);
   default: return 1;
   }
}


char sukcesja (POP Pnowa, POP Pt, POP Ot)
{
   switch (Selekcja->tSukcesji)
   {
   case 1: return sProsta (Pnowa, Pt, Ot);
   case 2: return sElitaStd (Pnowa, Pt, Ot);
   case 3: return sElitaCZ (Pnowa, Pt, Ot);
   default: return 1;
```

```
    }

}

double fRang (unsigned Ranga, unsigned MaxRanga)
{
    switch (Selekcja→tRang)
    {
    case 1: return fRWykl (Ranga, MaxRanga);
    case 2: return fRProg (Ranga, MaxRanga);
    default: return 1;
    }
}
/* Reprodukcja proporcjonalna (ruletkowa) z przesuwaniem wartosci
    funkcji przystosowania. Dzieki temu, wartosci te sa zawsze nie mniejsze
    od zera */

char rPropMod (POP Pt, unsigned *Wybrani, unsigned nWybranych )
{
    double *RozkladSel; /*rozklad p-stwa selekcji*/
    unsigned i;
    double SumaPrzyst;

/* utworzenie dystrybuanty rozkladu p-stwa selekcji */

    RozkladSel =(double*) calloc (Pt→nOsobnik, sizeof(double));

    SumaPrzyst=(Pt→Monitor→Sredni→Przystosowanie-
        Pt→Osobniki[Pt→Monitor→Minimum]→Przystosowanie )*Pt→nOsobnik;
    if (SumaPrzyst!=0)
        for(i=0; i<Pt→nOsobnik; i++)
            RozkladSel[i] = (Pt→Osobniki[i]→Przystosowanie -
Pt→Osobniki[Pt→Monitor→Minimum]→Przystosowanie) / SumaPrzyst;
    else
        for(i=0; i<Pt→nOsobnik; i++)
            RozkladSel[i] = 1;


/* reprodukcja przez wielokrotny wybor ze zwracaniem i kopiowanie */
/* wylosowanego osobnika */

    for(i=0; i<nWybranych; i++)
        Wybrani[i] = ruletka (RozkladSel, Pt→nOsobnik);

/* zwolnienie zbednej pamieci */
    free (RozkladSel);

    return 0;
}
unsigned zwyciezca (OSOBNIK* Konkurenci, unsigned* Turniej, unsigned nTurnieju)
{
    unsigned i, Zwyciezca;

    for(i=1, Zwyciezca=Turniej[0]; i<nTurnieju; i++)
        if (Konkurenci[Turniej[i]]→Przystosowanie > Konkurenci[Zwyciezca]→Przystosowanie)
            Zwyciezca=Turniej[i];
    return Zwyciezca;
}
```

```
/* Reprodukcja turniejowa. Do turnieju wchodza osobniki ze zwracaniem,
moze sie wiec zdarzyc, ze jeden osobnik zostanie wybrany wiecej niz
jednokrotnie. Wielkosc turnieju jest zdefiniowana jako zmienna globalna
*/

char rTurnZwr (POP Pt, unsigned *Wybrani, unsigned nWybranych )
{
    unsigned Zwyciezca;
    unsigned *Turniej;
    unsigned i,j;

/* utworzenie miejsca na turniej */
    Turniej = (unsigned*)calloc (Selekcja->nTurnieju, sizeof(unsigned));

/* wielokrotne - przygotowanie i rozegranie turnieju */
    for(i=0; i<nWybranych; i++)
    {
        do{
        for (j=0; j<Selekcja->nTurnieju; j++)
            Turniej [j] = rownomierny (Pt->nOsobnik);

            Zwyciezca=zwyciezca (Pt->Osobniki, Turniej, Selekcja->nTurnieju);
        }while(Pt->Osobniki[Zwyciezca]->Przystosowanie == SMIERC);
        Wybrani[i]=Zwyciezca;
    }

/* czynnosci porzadkowe */
    free (Turniej);

    return 0;
}


#define EPS_TOL 0.0
// nieefektywna metoda sortowania
void wyznaczRangi (unsigned *Rangi, unsigned *MaxRanga,
                   unsigned nOsobnik, OSOBNIK *Osobniki)
{
    unsigned i,j;
    double Prototyp;
// double Tolerancja;
    unsigned Pom;
    unsigned *Sort;
    unsigned Ranga;

    Sort=(unsigned*)calloc(nOsobnik, sizeof(unsigned));

    //sortowanie numerow osobnikow - metoda brutalna
    for (i=0; i<nOsobnik; i++)
        Sort[i]=i;
    for (i=0, Prototyp=0; i<nOsobnik-1; i++)
        for(j=i+1; j<nOsobnik; j++)
        {
            if (Osobniki[Sort[i]]->Przystosowanie < Osobniki[Sort[j]]->Przystosowanie)
            {
                Pom=Sort[i];
                Sort[i]=Sort[j];
                Sort[j]=Pom;
            }
```

```
    }

  //przypisanie rang osobnikom
  Rangi[Sort[0]]=Ranga=1;
  for (i=1, Prototyp=Osobniki[Sort[0]]→Przystosowanie; i<nOsobnik; i++)
  {
    if (Prototyp-Osobniki[Sort[i]]→Przystosowanie <= EPS_TOL)
      Rangi[Sort[i]]=Ranga;
    else
    {
      Prototyp=Osobniki[Sort[i]]→Przystosowanie;
      Rangi[Sort[i]]=++Ranga;
    }
  }
  *MaxRanga=Ranga;

  free(Sort);

}

/* funkcja rang - wykladnicza*/
double fRWykl(unsigned Ranga, unsigned MaxRanga)
{
  double w;

  return (pow((MaxRanga - Ranga),Selekcja→fRWykladnik));
}

double fRProg (unsigned Ranga, unsigned MaxRanga)
{
  unsigned mu;

  mu=MaxRanga * Selekcja→obciecie;
  if (mu <1)
    mu=1;
  if (Ranga <= mu)
    return (1.0/mu);
  else
    return 0;
}

void wyznaczRozklad(double *Rozklad, unsigned *Rangi, unsigned MaxRanga,
                    unsigned nOsobnik)
{
  unsigned i;
  double Suma;
  double Mnoznik;


  for (i=0,Suma=0; i<nOsobnik; i++)
  {
    Rozklad[i]=fRang(Rangi[i], MaxRanga);
    Suma+=Rozklad[i];
  }
// Mnoznik=nOsobnik/Suma;
  for (i=0; i<nOsobnik; i++)
    Rozklad[i]/=Suma;
  // Rozklad[i]*=Mnoznik;
}
```

```
/* Reprodukcja rangowa niedeterministyczna.
*/

char rRang (POP Pt, unsigned *Wybrani, unsigned nWybranych )
{
    unsigned *Rangi;
    double *Rozklad;
    unsigned i,j;
    unsigned MaxRanga;

/* przygotowanie tablicy przekodowania rang na liczbe kopii */
    Rangi = (unsigned*)calloc (Pt->nOsobnik, sizeof(unsigned));
    Rozklad = (double*)calloc (Pt->nOsobnik, sizeof(double));

    wyznaczRangi(Rangi, &MaxRanga, Pt->nOsobnik, Pt->Osobniki);

// MaxRanga=Pt->nOsobnik; //atrapa
    wyznaczRozklad(Rozklad, Rangi, MaxRanga, Pt->nOsobnik);

/* wlasciwa selekcja rangowa */

    for (i=0; i<nWybranych ; i++)
    {
        j=ruletka(Rozklad, Pt->nOsobnik);
        Wybrani[i] = j;
    }

/* czynnosci porzadkowe */
    free (Rozklad);
    free (Rangi);

    return 0;


}

/* Reprodukcja prosta - nie ma zroznicowania prawdopodobienstw. */

char rProsta (POP Pt, unsigned *Wybrani, unsigned nWybranych )
{
    unsigned i;

/* reprodukcja przez wielokrotny wybor ze zwracaniem i kopiowanie */
/* wylosowanego osobnika - wybor z jednakowym prawdopodobienstwem */

    for(i=0; i<nWybranych; i++)
        Wybrani[i] = rownomierny(Pt->nOsobnik);

    return 0;
}

/* Reprodukcja proporcjonalna (ruletkowa) bez przesuwania wartosci
    funkcji przystosowania. */


char rPropStd (POP Pt, unsigned *Wybrani, unsigned nWybranych )
{
    double *RozkladSel; /*rozklad p-stwa selekcji*/
    unsigned i;
    double SumaPrzyst;
```

```
/* utworzenie dystrybuanty rozkladu p-stwa selekcji */

    RozkladSel =(double*) calloc (Pt→nOsobnik, sizeof(double));

    SumaPrzyst=(Pt→Monitor→Sredni→Przystosowanie)*Pt→nOsobnik;
    if (SumaPrzyst!=0)
        for(i=0; i<Pt→nOsobnik; i++)
            RozkladSel[i] = (Pt→Osobniki[i]→Przystosowanie ) / SumaPrzyst;
    else
        for(i=0; i<Pt→nOsobnik; i++)
            RozkladSel[i] = 1;


/* reprodukcja przez wielokrotny wybor ze zwracaniem i kopiowanie */
/* wylosowanego osobnika */

    for(i=0; i<nWybranych; i++)
        Wybrani[i] = ruletka (RozkladSel, Pt→nOsobnik);

/* zwolnienie zbednej pamieci */
    free (RozkladSel);

    return 0;
}

/* Reprodukcja turniejowa. Do turnieju wchodza osobniki bez zwracania,
Wielkosc turnieju jest zdefiniowana jako zmienna globalna.
*/

char rTurnBZ (POP Pt, unsigned *Wybrani, unsigned nWybranych )
{
    unsigned Zwyciezca;
    unsigned *Turniej;
    unsigned i;

/* utworzenie miejsca na turniej */
    Turniej = (unsigned*)calloc (Selekcja→nTurnieju, sizeof(unsigned));

/* wielokrotne - przygotowanie i rozegranie turnieju */
    for(i=0; i<nWybranych ; i++)
    {
        do{
            kombinacja (Turniej, Pt→nOsobnik, Selekcja→nTurnieju);

            Zwyciezca=zwyciezca (Pt→Osobniki, Turniej, Selekcja→nTurnieju);
        }while (Pt→Osobniki [Zwyciezca ]→Przystosowanie == SMIERC);
        Wybrani [i] = Zwyciezca;
    }

/* czynnosci porzadkowe */
    free (Turniej);

    return 0;
}


/* Sukcesja nieelitarna — z calkowitym zastepowaniem*/

char sProsta (POP Pnowa, POP Pt, POP Ot)
```

```
{
    unsigned i;

    for (i=0; i<Pnowa->nOsobnik; i++)
        Pnowa->Osobniki [i] = osPrzesun (&(Ot->Osobniki[i]));
    return 0;
}

/* Sukcesja elitarna */
char sElitaStd (POP Pnowa, POP Pt, POP Ot)
{
    unsigned i, j, k;

    popSortuj (Pt);
    popSortuj (Ot);

/* Wybieranie najlepszych osobnikow ze zlaczenia Pt i Ot do nastepnej
populacji bazowej. Pierwszenstwo (przy jednakowych wartosciach funkcji
przystosowania) maja nowo wygenerowane Osobniki z Pt
*/

    j=0; k=0;
    for (i=0; i<Pnowa->nOsobnik; i++)
    {
/* jesli nie ma juz osobnikow w Pt albo Ot - koniec petli*/
        if (k>=Ot->nOsobnik || j>=Selekcja->nElity*Algorytm->nPop) break;

/* wybor lepszego do skopiowania */
        if (Ot->Osobniki[k]->Przystosowanie > Pt->Osobniki[j]->Przystosowanie)
            Pnowa->Osobniki [i] = osPrzesun (&(Ot->Osobniki[k++]));
        else if (Ot->Osobniki[k]->Przystosowanie == Pt->Osobniki[j]->Przystosowanie)
            Pnowa->Osobniki [i] = osPrzesun (&(Pt->Osobniki[j++]));
        else
            Pnowa->Osobniki [i] = osPrzesun (&(Pt->Osobniki[j++]));
    }

/* uzupelnienie ewentualnych brakow osobnikami z tej populacji, ktora
nie zostala wyczerpana */
    if (i<Pnowa->nOsobnik)
    {
        if (k>=Ot->nOsobnik)
            for (; i<Pnowa->nOsobnik; i++)
                Pnowa->Osobniki [i] = osPrzesun (&(Pt->Osobniki[j++]));
        else
            for (; i<Pnowa->nOsobnik; i++)
                Pnowa->Osobniki [i] = osPrzesun (&(Ot->Osobniki[k++]));
    }

/* czynnosci porzadkowe */
    return 0;
}


/* Sukcesja elitarna z maksymalnym czasem zycia */
char sElitaCZ (POP Pnowa, POP Pt, POP Ot)
{

    unsigned i, j, k;

    for (i=0 ; i<Pt->nOsobnik ; i++)
```

```
        if (Pt→Osobniki[i]→Wiek > Selekcja→maxWiek)
            Pt→Osobniki[i]→Przystosowanie = SMIERC;

    popSortuj (Pt);
    popSortuj (Ot);

/* Wybieranie najlepszych osobnikow ze zlaczenia Pt i Ot do nastepnej
populacji bazowej. Pierwszenstwo (przy jednakowych wartosciach funkcji
przystosowania) maja nowo wygenerowane Osobniki z Pt
*/

    j=0; k=0;
    for (i=0; i<Pnowa→nOsobnik; i++)
    {
/* jesli nie ma juz osobnikow w Pt albo Ot - koniec petli*/
        if (k>=Ot→nOsobnik || j>=Selekcja→nElity*Algorytm→nPop) break;

/* wybor lepszego do skopiowania */
        if (Ot→Osobniki[k]→Przystosowanie > Pt→Osobniki[j]→Przystosowanie)
            Pnowa→Osobniki [i] = osPrzesun (&(Ot→Osobniki[k++]));
        else if (Ot→Osobniki[k]→Przystosowanie == Pt→Osobniki[j]→Przystosowanie)
            Pnowa→Osobniki [i] = osPrzesun (&(Pt→Osobniki[j++]));
        else
            Pnowa→Osobniki [i] = osPrzesun (&(Pt→Osobniki[j++]));
    }

/* uzupelnienie ewentualnych brakow osobnikami z tej populacji, ktora
nie zostala wyczerpana */
    if (i<Pnowa→nOsobnik)
    {
        if (k>=Ot→nOsobnik)
            for (; i<Pnowa→nOsobnik; i++)
                Pnowa→Osobniki [i] = osPrzesun (&(Pt→Osobniki[j++]));
        else
            for (; i<Pnowa→nOsobnik; i++)
                Pnowa→Osobniki [i] = osPrzesun (&(Ot→Osobniki[k++]));
    }

/* czynnosci porzadkowe */
    return 0;
}

#define SEL_PAR 8
void okreslSelekcje(FILE *plikKonf, FILE *plikKontr)
{
    char Wczytanych;
    char *Nazwy[SEL_PAR];
    CZYTFUN czytFun[SEL_PAR];
    void *Adresy[SEL_PAR];
    unsigned *Liczba[SEL_PAR];

    Selekcja = (SELEKCJA )malloc(sizeof(SELEKCJAS));
    assert(Selekcja!=NULL);

    Nazwy[0]="treprodukcji"; czytFun[0]=czytUnsigned; Adresy[0]=&(Selekcja→tReprodukcji);
Liczba[0]=NULL;
    Nazwy[1]="tsukcesji"; czytFun[1]=czytUnsigned; Adresy[1]=&(Selekcja→tSukcesji);
Liczba[1]=NULL;
    Nazwy[2]="nturnieju"; czytFun[2]=czytUnsigned; Adresy[2]=&(Selekcja→nTurnieju);
Liczba[2]=NULL;
```

```
    Nazwy[3]="nelity"; czytFun[3]=czytDouble; Adresy[3]=&(Selekcja→nElity); Liczba[3]=NULL;
    Nazwy[4]="trang"; czytFun[4]=czytUnsigned; Adresy[4]=&(Selekcja→tRang); Liczba[4]=NULL;
    Nazwy[5]="maxwiek"; czytFun[5]=czytUnsigned; Adresy[5]=&(Selekcja→maxWiek); Liczba[5]=NULL;
    Nazwy[6]="obciecie"; czytFun[6]=czytDouble; Adresy[6]=&(Selekcja→obciecie); Liczba[6]=NULL;
    Nazwy[7]="frwykladnik"; czytFun[7]=czytDouble ; Adresy[7]=&(Selekcja→fRWykladnik);
Liczba[7]=NULL;

    Wczytanych = wczytajSymbole (plikKonf, plikKontr, Nazwy, czytFun, Adresy, Liczba, SEL_PAR);

/* assert (Wczytanych==SEL_PAR);*/

}
```

6.11 Plik stop.c

Zwartość tego pliku stanowią implementacje różnych kryteriów zatrzymania algorytmu. Każde z nich wykorzystuje przynajmniej jeden z monitorów (zadania lub populacji).

6.11.1 stop.h

```
/*
+——————————————-+
| (C) Jaroslaw Arabas, luty 1999 |
+——————————————-+
*/

#ifndef STOP_H
#define STOP_H

#include <stdio.h>
#include "zadanie.h"

typedef struct{
    unsigned tStopu;
    unsigned SzerStop;
    double WysStop;
    unsigned MaxGen;
    unsigned MaxPrzyst;
    double MinPrzyst;
} STOPS;

typedef STOPS* STOP;


char krytStopu (ZADANIE Zadanie);

extern STOP Stop;

void okreslStop(FILE*,FILE*);

#endif
```

6.11.2 stop.c

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
```

```
#include "stop.h"
#include "wewy.h"

STOP Stop;

char stopCierpG (ZADANIE Zadanie);
char stopMaxG (ZADANIE Zadanie);
char stopCierpF (ZADANIE Zadanie);
char stopMaxF (ZADANIE Zadanie);
char stopMinPrzyst (ZADANIE Zadanie);


char krytStopu (ZADANIE Zadanie)
{
    switch(Stop->tStopu)
    {
    case 1: return stopCierpG (Zadanie);
    case 2: return stopMaxG (Zadanie);
    case 3: return stopCierpF (Zadanie);
    case 4: return stopMaxF (Zadanie);
    case 5: return stopMinPrzyst (Zadanie);
    default: return stopMinPrzyst (Zadanie);
    }
}

/* kryterium cierpliwosciowe - wykorzystujace liczbe generacji jako
 miare szybkosci */
char stopCierpG (ZADANIE Zadanie)
{

    if (Zadanie->Monitor->Generacja <Stop->SzerStop)
        return 0;
    if (Zadanie->Monitor->Rozwiazanie->Wiek>Stop->SzerStop)
        return 1;
    else if (Zadanie->Monitor->OkienkoGen[Stop->SzerStop-1]-Zadanie->Monitor->OkienkoGen[0]<=
Stop->WysStop/((double)Stop->SzerStop))
        return 1;
    else
        return 0;
}

/*kryterium na maksymalna liczbe generacji
*/
char stopMaxG (ZADANIE Zadanie)
{

    if (Zadanie->Monitor->Generacja > Stop->MaxGen)
        return 1;
    else
        return 0;
}

/* kryterium mnimalnej szybkosci zbieznosci wykorzystujace liczbe
 obliczen wartosci funkcji przystosowania
*/
char stopCierpF (ZADANIE Zadanie)
{

    if (Zadanie->Monitor->LPrzystosowan <Stop->SzerStop)
        return 0;
```

```
    if (Zadanie→Monitor→Rozwiazanie→Wiek>Stop→SzerStop)
        return 1;
    else if (Zadanie→Monitor→OkienkoPrzyst[Stop→SzerStop-1]-Zadanie→Monitor→OkienkoPrzyst[0]<=
Stop→WysStop/((double)Stop→SzerStop))
        return 1;
    else
        return 0;
}
/* kryterium na maksymalna liczbe obliczen wartosci funkcji przystosowania
*/
char stopMaxF(ZADANIE Zadanie)
{

    if (Zadanie→Monitor→LPrzystosowan > Stop→MaxPrzyst)
        return 1;
    else
        return 0;
}

/* kryterium na minimalna wartosc funkcji przystosowania
*/
char stopMinPrzyst (ZADANIE Zadanie)
{

    if (Zadanie→Monitor→Rozwiazanie→Przystosowanie >= Stop→MinPrzyst)
        return 1;
    else if (Zadanie→Monitor→LPrzystosowan > Stop→MaxPrzyst)
        return 1;
    else
        return 0;
}

#define STOP_PAR 6
void okreslStop(FILE* plikKonf, FILE *plikKontr)
{
    char Wczytanych;
    char *Nazwy[STOP_PAR];
    CZYTFUN czytFun[STOP_PAR];
    void *Adresy[STOP_PAR];
    unsigned *Liczba[STOP_PAR];

    Stop = (STOP )malloc(sizeof(STOPS));
    assert(Stop!=NULL);

    Nazwy[0]="tstopu"; czytFun[0]=czytUnsigned; Adresy[0]=&(Stop→tStopu); Liczba[0]=NULL;
    Nazwy[1]="szerstop"; czytFun[1]=czytUnsigned; Adresy[1]=&(Stop→SzerStop); Liczba[1]=NULL;
    Nazwy[2]="wysstop"; czytFun[2]=czytDouble; Adresy[2]=&(Stop→WysStop); Liczba[2]=NULL;
    Nazwy[3]="maxgen"; czytFun[3]=czytUnsigned; Adresy[3]=&(Stop→MaxGen); Liczba[3]=NULL;
    Nazwy[4]="maxprzyst"; czytFun[4]=czytUnsigned; Adresy[4]=&(Stop→MaxPrzyst); Liczba[4]=NULL;
    Nazwy[5]="minprzyst"; czytFun[5]=czytDouble; Adresy[5]=&(Stop→MinPrzyst); Liczba[5]=NULL;

    Wczytanych = wczytajSymbole (plikKonf, plikKontr, Nazwy, czytFun, Adresy, Liczba, STOP_PAR);

/* assert (Wczytanych==STOP_PAR);*/
}
```

6.12 Plik wewy.c

Plik ten zawiera pomocnicze funkcje potzrebne do obsługi wejścia/wyjścia podczas parametryzacji algorytmu za pomocą plików konfiguracyjnych.

6.12.1 wewy.h

```
#ifndef WEWY_H
#define WEWY_H

#include <stdio.h>

typedef char (*CZYTFUN)(FILE*, FILE*, void*, unsigned);

char wczytajSymbole (FILE* Zrodlo, FILE *Kontrola, char *Nazwy[ ],
                     const CZYTFUN *czytFun,
                     void** Adresy, unsigned *Liczba[ ], const unsigned IleMozliwosci);

char czytDouble (FILE* Zrodlo, FILE* Kontrola, void* miejsce, unsigned liczba);
char czytNDouble (FILE* Zrodlo, FILE* Kontrola, void* miejsce, unsigned liczba);
char czytUnsigned (FILE* Zrodlo, FILE* Kontrola, void* miejsce, unsigned liczba);
char czytNUnsigned (FILE* Zrodlo, FILE* Kontrola, void* miejsce, unsigned liczba);
char czytLancuch (FILE* Zrodlo, FILE* Kontrola, void* miejsce, unsigned liczba);
char czytOperator (FILE* Zrodlo, FILE* Kontrola, void* miejsce, unsigned liczba);

#endif
```

6.12.2 wewy.c

```
#include "wewy.h"
#include "genetyka.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

char czytNUnsigned(FILE* Zrodlo, FILE* Kontrola, void* miejsce, unsigned Liczba)
{
    unsigned *Dokad;
    unsigned i;
    unsigned nWczytanych=0;

    Dokad = (unsigned*)calloc(Liczba, sizeof(unsigned));
    assert (Dokad!=NULL);
    for (i=0; i<Liczba ; i++)
    {
        if ( fscanf(Zrodlo, "%u", Dokad+i) == EOF )
            return nWczytanych;
        nWczytanych++;
        fprintf(Kontrola, "%u ", Dokad[i]);
    }
    *((unsigned**)miejsce)=Dokad;
    return nWczytanych;
}

char czytNDouble(FILE* Zrodlo, FILE* Kontrola, void* miejsce, unsigned Liczba)
{
    double *Dokad;
    unsigned i;
    unsigned nWczytanych=0;
```

```
    Dokad = (double*)calloc(Liczba , sizeof(double));
    assert (Dokad!=NULL);
    for (i=0; i<Liczba ; i++)
    {
        if ( fscanf(Zrodlo, "%lf", Dokad+i) == EOF )
            return nWczytanych;
        fprintf(Kontrola, "%lf ", Dokad[i]);
    }
    *((double**)miejsce)=Dokad;
    return nWczytanych;
}

char czytDouble(FILE* Zrodlo, FILE* Kontrola, void* miejsce, unsigned Liczba)
{
    double Dokad;
    assert (Liczba==1);
    if ( fscanf(Zrodlo, "%lf", &Dokad) == EOF )
        return EOF;
    fprintf(Kontrola, "%lf ", Dokad);
    *((double*)miejsce)=Dokad;
    return 1;
}

char czytUnsigned(FILE* Zrodlo, FILE* Kontrola, void* miejsce, unsigned Liczba)
{
    unsigned Dokad;
    assert (Liczba==1);
    if ( fscanf(Zrodlo, "%u", &Dokad) == EOF )
        return EOF;
    fprintf(Kontrola, "%u ", Dokad);
    *((unsigned*)miejsce)=Dokad;
    return 1;
}

#define MAX_ZNAK 100
/*char czytLancuch(FILE* Zrodlo, FILE* Kontrola, void* miejsce, unsigned Liczba)
{
    char Bufor[MAX_ZNAK+1];
    char* Buf2;
    assert (Liczba==1);
    fgets(Bufor, MAX_ZNAK, Zrodlo);
    Buf2 = strdup(Bufor);
    *(char**)miejsce = Buf2;
    fprintf(Kontrola, "%s ", Bufor);
    return 1;
}
*/
char czytLancuch(FILE* Zrodlo, FILE* Kontrola, void* miejsce, unsigned Liczba)
{
    char Bufor[MAX_ZNAK+1];
    char* Buf2;
    assert (Liczba==1);
    if ( fscanf(Zrodlo, "%s", Bufor) == EOF )
        return EOF;
    Buf2 = strdup(Bufor);
    *(char**)miejsce = Buf2;
    fprintf(Kontrola, "%s ", Bufor);
    return 1;
}
```

```
/*void czytOperator(char* Zrodlo, FILE* Kontrola, void* miejsce)
{
    OPERATOR Operator;
    static unsigned liczba=0;

    Operator = (OPERATOR*) miejsce;
    fprintf(Kontrola, "operator nr %u: ", liczba+1);
    sscanf(Zrodlo, "%u", &(Operator->tOperatora));
    fprintf(Kontrola, "%u ", (Operator->tOperatora));
    sscanf(Zrodlo, "%lf", &(Operator->Pstwo));
    fprintf(Kontrola, "%lf ", (Operator->Pstwo));
    sscanf(Zrodlo, "%u", &(Operator->nRodzicow));
    fprintf(Kontrola, "%u ", (Operator->nRodzicow));
    sscanf(Zrodlo, "%u", &(Operator->nDzieci));
    fprintf(Kontrola, "%u ", (Operator->nDzieci));
}
*/
char wyborZListy (char *Nazwy[ ], char *Bufor, unsigned Liczba)
{
    unsigned i;

   // strtol(Bufor);
    for (i=0; i<Liczba; i++)
        if (!strcmp(Bufor, Nazwy[i]))
            return i;
    return -1;
}

char wczytajSymbole (FILE* Plik, FILE *Kontrola, char *Nazwy[ ],
                            const CZYTFUN *czytFun,
                            void** Adresy, unsigned *Liczba[ ], const unsigned IleMozliwosci)
{
    char Nazwa[101];
    char Wczytany, nWczytanych;
    char Symbol;


    nWczytanych=0;
    Wczytany = fscanf(Plik, "%s", Nazwa);

    while(Wczytany!=EOF)
    {
        Symbol=wyborZListy(Nazwy, Nazwa, IleMozliwosci);
        if (Symbol>=0 && Symbol <IleMozliwosci)
        {
            fprintf (Kontrola, "%s = ", Nazwa);
            if (Liczba[Symbol]!=NULL)
                Wczytany = (czytFun[Symbol])(Plik, Kontrola, Adresy[Symbol], *(Liczba[Symbol]));
            else
                Wczytany = (czytFun[Symbol])(Plik, Kontrola, Adresy[Symbol], 1);
             if (Wczytany == EOF)
                 return nWczytanych;
           nWczytanych++;
           fprintf (Kontrola, "\n");
           fflush (Kontrola);
        }
        Wczytany = fscanf(Plik, "%s", Nazwa);
    }
    return nWczytanych;
```

```
}
```

6.13 Plik `zadanie.c`

W tym pliku zdefiniowane jest zadanie, stanowiąe środowisko, do którego algorytm ewolucyjny się adaptuje (np. poprzez optymalizację funkcji celu w obszarze dopuszczalnym). Użytkowk, chcąc zdefiniować zadanie, powinien zmodyfikować następujące fragmenty kodu:

1. funkcja `void okreslZadanie(FILE* plikKonf)`

   Funkcja ta definiuje parametry zadania, na podstawie którego obliczana jest wartość funkcji przystosowania. W przypadku zadań ciągłych należy zdefiniowac ew. ograniczenia kostkowe, podać wskazanie na funkcję obliczająca wartość funkcji celu oraz wskazania na funkcje implementujące ograniczenia równościowe i nierównościowe, określic wymiarowość zadania.

2. funkcja `void zadInicjuj(TGEN** Punkty, unsigned Liczba)` Jej nazwa może się wydawac z początku myląca. Zadaniem tej funkcji jest wygenerowanie punktów należących do bszaru dopuszczalnego, stanowiących genotypy osobników pierwszej populacji bazowej. Można przyjać że będą one generowane z rozkładem równomiernym, lecz w gre mogą wchodzić również inne (np.deterministyczne) metody inicjacji, takie jak zbiór punktów na siatce prostokątnej lub zbiór punktów o niskiej dyskrepancji.

6.13.1 `zadanie.h`

```
/*
+----------------------------------+
| (C) Jaroslaw Arabas, luty 1999 |
+----------------------------------+
*/

#ifndef ZADANIE_H
#define ZADANIE_H

#include <stdio.h>
#include "osobnik.h"
#include "monitor.h"

/* Funkcja - typ uzywany w zadaniach optymalizacji jednokryterialnej*/
typedef double (*FUNKCJA)(const TGEN*, unsigned);

/* funkcje — celu (rowniez wielokryterialnie),
                ograniczen nierownosciowych,
                ograniczen rownosciowych
            obszaru zainteresowania - ograniczen kostkowych
*/

typedef struct{
        unsigned nLiter; /* uzywane jedynie do przypadkow dysk. */
        unsigned nZmiennych;
#ifdef WIELOKRYT
        unsigned nCelow;
        FUNKCJA* fcelu;
        char *Minimalizacja;
#else
        FUNKCJA fcelu;
        char Minimalizacja;
#endif
        unsigned nNierown;
        FUNKCJA *nierownosciowe;
        unsigned nRown;
        FUNKCJA *rownosciowe;
```

```
        MONZAD *Monitor;
        unsigned saKostkowe;
        TGEN *GorneX;
        TGEN *DolneX;

        char* plikDanych;
        unsigned tInic;
        double* Rozklad;
    } ZADANIES;

typedef ZADANIES* ZADANIE;



/* funkcja przystosowania oblicza przystosowanie osobnika na podstawie jego
   funkcji celu i ograniczen, zapisuje wyliczona wartosc wartosc oraz zwraca
   ja jako wynik */
double fPrzystosowania (OSOBNIK Osobnik);

/* krytStopu okresla czy zostal spelniony warunek zatrzymania */
char krytStopu (ZADANIE Zadanie);

/*sluzy do definicji zadania optymalizacji - wymiarowosci, funkcji celu i
ograniczen */

void okreslZadanie(FILE*);

void zadInicjuj(TGEN**, unsigned);

extern ZADANIE Zadanie;

#endif
```

6.13.2 zadanie.c

```
/*
+---------------------------------+
| (C) Jaroslaw Arabas, listopad 1998 |
+---------------------------------+
*/

#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "defs.h"
#include "losowe.h"
#include "osobnik.h"
#include "wewy.h"
#include "zadanie.h"


ZADANIE Zadanie;

double max(double a, double b)
{
   if (a>b)
```

```
        return a;
    else
        return b;
}
double sqr(double x)
{
    return (x*x);
}
double Gauss(double x, double mu, double sigma)
{
    return (exp(-sqr((x-mu)/sigma)));
}

double Galar(const double *x, unsigned n)
{
    assert (n==1);
    return (max(1.1*Gauss(x[0],-1, 0.2),Gauss(x[0],1,1)));

}

void inicjujCiagle()
{
    FILE *PlikKonf;
    int wynik;
    unsigned i;

    PlikKonf = fopen(Zadanie->plikDanych,"r");
    assert(PlikKonf!=NULL);
    wynik=fscanf(PlikKonf,"%d", &Zadanie->nZmiennych);
    assert(wynik==1);
    Zadanie->DolneX=(double*)calloc(Zadanie->nZmiennych, sizeof(double));
    Zadanie->GorneX=(double*)calloc(Zadanie->nZmiennych, sizeof(double));
    assert(Zadanie->DolneX!=NULL);
    assert(Zadanie->GorneX!=NULL);
    for(i=0; i<Zadanie->nZmiennych; i++)
    {
        wynik=fscanf(PlikKonf, "%lf%lf", &(Zadanie->DolneX[i]), &(Zadanie->GorneX[i]));
        assert(wynik==2);
    }
    fclose(PlikKonf);
}

#define ZAD_PAR 4
void okreslZadanie(FILE* plikKonf)
{
    char Wczytanych;
    char *Nazwy[ZAD_PAR];
    CZYTFUN czytFun[ZAD_PAR];
    void *Adresy[ZAD_PAR];
    unsigned *Liczba[ZAD_PAR];
    FILE* plikKontr;

    plikKontr=fopen("zadanie.log", "wt");
    assert(plikKontr!=NULL);

    Zadanie = (ZADANIE)malloc (sizeof(ZADANIES));
    assert(Zadanie!=NULL);


    Nazwy[0]="plikdanych"; czytFun[0]=czytLancuch ;Adresy[0]=&(Zadanie->plikDanych);
```

```
Liczba[0]=NULL;
    Nazwy[1]="rozklad"; czytFun[1]=czytNDouble ;Adresy[1]=&(Zadanie→Rozklad);
Liczba[1]=&(Zadanie→nLiter);
    Nazwy[2]="tinic"; czytFun[2]=czytUnsigned ;Adresy[2]=&(Zadanie→tInic); Liczba[2]=NULL;
    Nazwy[3]="nliter"; czytFun[3]=czytUnsigned ;Adresy[3]=&(Zadanie→nLiter); Liczba[3]=NULL;

    Wczytanych = wczytajSymbole (plikKonf, plikKontr, Nazwy, czytFun, Adresy, Liczba, ZAD_PAR);



    Zadanie→Monitor=inicjujMonZad();

    Zadanie→Minimalizacja=-1;
    Zadanie→nRown=0;
    Zadanie→nNierown=0;
    inicjujCiagle();
    Zadanie→fcelu =Galar;

    fclose(plikKontr);
}

void zadInicjuj(TGEN** Punkty, unsigned Liczba)
{
    unsigned i,j;

    for(i=0; i<Liczba; i++)
        for (j=0; j<Zadanie→nZmiennych; j++)
            (Punkty[i])[j]=prostokatny(Zadanie→DolneX[j], Zadanie→GorneX[j]);
}
```