Niekonwencjonalne metody obliczeniowe

Algorytmy genetyczne - ćwiczenia

Jakub Wróblewski

Warszawa, 1996

Spis treści

1. Problem optymalizacyjny ... 3
2. Zasada działania algorytmu genetycznego ... 6
3. Implementacja prostego algorytmu genetycznego ... 11
4. Problem reprezentacji ... 15
5. Modyfikacje i ulepszenia niezależne od problemu ... 19
  5.1. Skalowanie funkcji celu ... 19
  5.2. Modyfikacje algorytmu selekcji ... 21
  5.3. Bariery reprodukcyjne - modyfikacja algorytmu krzyżowania ... 23
  5.4. Diploidalny kod genetyczny ... 24
6. Strategie ewolucyjne ... 26
7. Przykłady zastosowań algorytmów genetycznych ... 33
  7.1. Algorytmy genetyczne poszukujące strategii ... 33
  7.2. Kodowanie zmiennoprzecinkowe ... 35
  7.3. Kodowanie permutacji ... 37
  7.4. Programowanie genetyczne ... 42
Bibliografia ... 45

1. Problem optymalizacyjny

Przez **problem optymalizacyjny** w dalszej części książki będziemy rozumieli problem sformułowany następująco:

> "Dana jest przestrzeń X. Znaleźć $x \in X$ taki, że x spełnia określone warunki."

W szczególności do klasy problemów optymalizacyjnych należą problemy typu:

> "Dana jest przestrzeń X i **funkcja celu** f: $X \rightarrow \mathbf{R}$. Znaleźć $x \in X$ taki, że f(x) jest maksimum (minimum) funkcji f na zbiorze X."

Większość problemów rozwiązywanych przez komputery jest sformułowana na jeden z powyższych sposobów. Część z tych problemów ma znane, typowe rozwiązania korzystające z ich specyfiki - na przykład maksymalizacja funkcji - wielomianów na przestrzeni będącej przedziałem liczbowym nie stanowi trudnego zadania. Niestety, nie wszystkie praktyczne problemy mają proste i jednoznaczne przepisy (algorytmy) na znajdowanie rozwiązań.

Ważnym parametrem mającym wpływ na to, czy problem należy uznać za trudny, jest moc (wielkość) przestrzeni stanów. Zauważmy, że każdy rozwiązywalny przez komputer problem musi mieć przestrzeń stanów skończoną, ze względu na ograniczoną pamięć maszyny. Nawet wtedy, gdy komputer operuje na pozornie nieskończonej przestrzeni, np. szukając liczby rzeczywistej x takiej, że $3-x^2$ jest maksymalne, tak naprawdę liczba różnych rozwiązań, które komputer jest w stanie podać, jest ograniczona przez np. dokładność reprezentacji liczb zmiennoprzecinkowych. Teoretycznie więc każde rozwiązalne przy pomocy komputera zadanie można rozwiązać metodą **przeglądania przestrzeni stanów**: skoro możliwych rozwiązań jest skończenie wiele, przejrzyjmy je wszystkie, a w końcu trafimy na właściwe. Dzięki wciąż rosnącej szybkości współczesnych komputerów jest to metoda użyteczna dla wielu zadań - praktycznie wszystkich o przestrzeni stanów mniejszej, niż milion (albo i 100 milionów - jeśli mamy więcej czasu) elementów. Takich problemów również nie będziemy nazywali trudnymi.

Przykład 1.1. Sortowanie.
Zadanie posortowania zbioru wartości liczbowych można formalnie zapisać następująco:
Dany jest zbiór n wartości liczbowych $x_1, \ldots x_n$. Znaleźć taką permutację σ elementów tego zbioru, by:

$$\forall\ 1 \le i < j \le n\ \ x_{\sigma(i)} > x_{\sigma(j)}$$

Przestrzenią stanów tego zadania jest przestrzeń wszystkich permutacji. Moc przestrzeni (n!) wyklucza stosowanie metody przeglądania wszystkich możliwych rozwiązań. Z drugiej strony, zadanie to nie jest zaliczane do trudnych - istnieją specjalne algorytmy pozwalające posortować duże zbiory wykonując przy tym stosunkowo niewiele - rzędu n^2, a nawet tylko n log (n) operacji.

Przykład 1.2. Problem komiwojażera.
Jest to znany i często przytaczany przykład problemu o prostym sformułowaniu, ale trudnego do rozwiązania. Mamy dane n miast połączonych drogami o znanej długości. Naszym zadaniem

jest znalezienie trasy przechodzącej przez wszystkie miasta dokładnie raz, przy czym całkowita długość trasy ma być możliwie najmniejsza.

Szukanym rozwiązaniem jest pewna trasa. Przestrzenią stanów będzie więc przestrzeń wszystkich możliwych tras. Moc tej przestrzeni można łatwo obliczyć zauważając, że każdą trasę da się jednoznacznie opisać podając kolejność odwiedzania miast. Tras jest więc tyle, ile permutacji n-elementowego zbioru miast, czyli n!. Oznacza to, że już dla n=15 sprawdzenie długości wszystkich tras jest praktycznie niemożliwe.

Co więcej, nie są znane metody szybkiego (tzn. o złożoności najwyżej n^k) rozwiązania tego zadania. Problem ten należy do klasy problemów NP-trudnych, o których sądzi się, że nie mają rozwiązania znacząco prostszego, niż przeglądanie wszystkich możliwych rozwiązań.

Inną klasą problemów trudnych są zagadnienia związane z optymalizowaniem funkcji wielu zmiennych, nieróżniczkowalnych i nieciągłych. W takich sytuacjach konwencjonalne metody analityczne zawodzą. Nawet różniczkowalne funkcje mogą sprawiać kłopoty, jeżeli mają dużo maksimów lokalnych, a nas interesuje maksimum globalne. Również funkcje określone na przestrzeniach dyskretnych mogą być trudne do analizy.

Metodą radzenia sobie z problemami trudnymi jest zastosowanie jednej z **technik przybliżonych**. Działają one na następującej zasadzie: skoro nie możemy znaleźć rozwiązania optymalnego, znajdźmy rozwiązanie jak najbliższe optymalnemu, ewentualnie znajdźmy rozwiązanie, które na 90% jest rozwiązaniem optymalnym. W wielu przypadkach tego typu rozwiązania nas satysfakcjonują.

Najprostszym przykładem algorytmu dającego dobry wynik z pewnym prawdopodobieństwem jest **błądzenie losowe**: z przestrzeni stanów wybieramy kolejne losowe punkty i sprawdzamy, czy stanowią one rozwiązanie. Po dostatecznie (nieskończenie) długim czasie trafimy na rozwiązanie z prawdopodobieństwem 1. Algorytm ten możemy ulepszyć: jeżeli podejrzewamy, że rozwiązanie leży w jakimś obszarze przestrzeni stanów, możemy zamiast losowania jednostajnego zwiększyć szanse wylosowania punktów z tego obszaru.

Przedstawione poniżej metody przybliżone stosuje się do zadań z funkcją celu i wymagają pewnych dodatkowych informacji o zadaniu. Zakładamy mianowicie, że na przestrzeni stanów da się wprowadzić strukturę sąsiedztwa, tzn. dla każdego elementu tej przestrzeni potrafimy wskazać jednoznacznie jego sąsiadów. Dodatkowo sama funkcja celu powinna mieć pewne cechy regularności.

Pierwszą z tych metod jest metoda **wspinaczki** (*hill climbing*). Oto algorytm postępowania, zakładając, że chcemy znaleźć maksimum funkcji f(x):

1. Startujemy z punktu $x \in X$ wybranego losowo.
2. Wybieramy losowo jednego z sąsiadów punktu x, oznaczamy go x'.
3. Porównujemy wartości funkcji celu w punkcie x i x'.
4. Jeżeli f(x') > f(x), czyli nowy punkt daje lepszą wartość, przenosimy się do tego punktu, tzn. przypisujemy x = x' i przechodzimy do kroku 2.
5. Jeżeli nie, zostajemy w punkcie x. Jeżeli punkt x ma jeszcze jakiegoś nie sprawdzonego sąsiada, wybieramy go jako nowy x' i przechodzimy do kroku 3.
6. Jeżeli sprawdziliśmy wszystkich sąsiadów, kończymy program. Jako wynik podajemy punkt x.

Zgodnie z warunkami wyjścia z programu, otrzymany punkt x jest lokalnym optimum funkcji f, tzn. żaden sąsiad x nie ma lepszej wartości funkcji, niż on. Oczywistą wadą tej metody jest to, że często optimum lokalne ma dużo niższą wartość, niż optimum globalne.

Druga metoda częściowo radzi sobie z tym problemem. Metoda ta, zwana **wychładzaniem** (lub wyżarzaniem, *simulated annealing*) ma schemat podobny, do poprzedniej:

1. Startujemy z punktu $x \in X$ wybranego losowo.

2. Wybieramy losowo jednego z sąsiadów punktu x, oznaczamy go x'.
3. Porównujemy wartości funkcji celu w punkcie x i x'.
4. Jeżeli f(x') > f(x), czyli nowy punkt daje lepszą wartość, przenosimy się do tego punktu, tzn. przypisujemy x = x' i przechodzimy do kroku 2.
5. Jeżeli nie, wykonujemy dodatkowe losowanie i z prawdopodobieństwem p(x,x') przenosimy się do nowego punktu, tzn. przypisujemy x = x' i przechodzimy do kroku 2.
6. Jeżeli losowanie się nie powiedzie, zostajemy w x i przechodzimy do kroku 2.
7. Program kończymy po ustalonej liczbie kroków. Jako wynik podajemy najlepszy znaleziony punkt x.

Algorytm działa na zasadzie: idź w górę, a czasem (z prawdopodobieństwem p(x,x')) w dół. Taka strategia daje nadzieje na to, że program nie zatrzyma się na maksimum lokalnym i będzie kontynuował przeglądanie przestrzeni stanów. Prawdopodobieństwo p(x,x') zależy od tego, o ile nowy punkt x' jest gorszy od x:

$$p(x, x') = e^{-\frac{f(x)-f(x')}{c}}$$

Prawdopodobieństwo przejścia jest tym większe, im różnica między punktami jest mniejsza. Dodatkowy parametr c (zwany temperaturą) reguluje "tolerancją" algorytmu: im ma wyższą wartość, tym częściej program zezwala na pogorszenie funkcji celu. Wartość c bliska zeru przekształca algorytm wychładzania w zwykły algorytm wspinaczki. Parametr c jest przeważnie zmienny w czasie: na początku ma wysoką wartość, w miarę upływu czasu zbiega niemal do zera.

Trzecia grupa metod, również oparta na zadaniach optymalizacji funkcji celu, to algorytmy genetyczne. Pozostałe rozdziały poświęcone będą opisowi ich działania i przykładów zastosowań.

2. Zasada działania algorytmu genetycznego

Podstawy działania algorytmów genetycznych wzorowane są na zjawisku doboru naturalnego. Ewolucja naturalna jest również pewnym procesem optymalizacyjnym - gatunki starają jak najlepiej "dopasować się" do środowiska, w jakim żyją, by optymalnie wykorzystać jego zasoby. Cechy pozwalające na skuteczną eksploatację danego środowiska powstają w sposób losowy - przez krzyżowanie się osobników i mutacje. To, że cechy korzystne mają szansę na szerokie rozpowszechnienie, jest zasługą zasady doboru naturalnego. Osobniki wyposażone w takie cechy mają większe szanse w walce o byt i szybciej się rozmnażają, co powoduje dalsze rozpowszechnianie się cechy. W końcu osobniki pozbawione cechy dającej przewagę zostaną wyparte przez inne i wyginą.

Załóżmy, że udało nam się zasymulować zjawisko ewolucji na osobnikach - komiwojażerach, przy czym osobniki przechodzące krótszą trasę mają większe szanse w walce o byt. Przez analogię do procesów naturalnych, po pewnym czasie w populacji złożonej z różnych komiwojażerów powinny zacząć dominować te osobniki, których trasy są jak najkrótsze. Przy odrobinie szczęścia powinniśmy doczekać się "wyhodowania" osobnika optymalnego - przechodzącego przez miasta po trasie globalnie najkrótszej. Taki osobnik powinien szybko opanować całą populację.

Na takich właśnie założeniach działają algorytmy genetyczne. Typowy schemat postępowania przedstawia się następująco:

1. Mamy zadanie optymalizacyjne: przestrzeń stanów i funkcję celu, którą chcemy zmaksymalizować. Każdy punkt przestrzeni stanów, czyli każde potencjalne rozwiązanie zadania, kodujemy w postaci "kodu genetycznego", przeważnie w alfabecie binarnym. Naszymi osobnikami będą właśnie te zakodowane punkty przestrzeni stanów.
2. Tworzymy populację osobników przez wylosowanie pewnej liczby punktów z przestrzeni stanów i zakodowanie ich do postaci osobników.
3. Populacja osobników podlega losowym mutacjom i procesowi krzyżowania - wymiany między losowymi osobnikami części materiału genetycznego.
4. Każdy osobnik populacji jest odkodowywany, a następnie liczona jest jego wartość funkcji celu.
5. Populacja jest poddawana selekcji: osobniki o dużej wartości funkcji celu rozmnażają się, pozostałe giną.
6. Wracamy do punktu 3. Proces ewolucji powtarzamy tak długo, aż uzyskamy osobnika spełniającego nasze wymagania.

Oto słownik podstawowych pojęć używanych w teorii i praktyce algorytmów genetycznych. Pochodzą one najczęściej od pojęć biologicznych, jednak procesy naturalne przeważnie są znacznie bardziej złożone, niż sugeruje to poniższy opis:

funkcja celu	- funkcja, którą optymalizujemy; inne nazwy: funkcja kosztu, funkcja straty, funkcja użyteczności, *objective function*;
funkcja przystosowania	- funkcja określająca stopień przystosowania osobnika do środowiska, na podstawie której odbywa się proces selekcji osobników; jest ściśle związana z funkcją celu, ale nie zawsze jest z nią identyczna; funkcja dopasowania, *fitness function*;
osobnik	- najmniejsza jednostka przenosząca informację genetyczną, odpowiada punktowi w przestrzeni stanów, czyli potencjalnemu rozwiązaniu zadania;

populacja	- zbiór osobników - punktów przestrzeni stanów, żyjących w jednym środowisku i konkurujących w walce o przetrwanie;
gen	- najmniejsza jednostka przenosząca informację genetyczną, można ją utożsamić z konkretnym miejscem (locus) w chromosomie;
chromosom	- uporządkowany zbiór genów, kodujący wszystkie informacje o osobniku;
allele	- odmiany wartości genu; jeżeli chromosom jest np. ciągiem binarnym, gen pojedyńczą zmienną binarną, to allelami są cyfry 0 i 1;
genotyp	- wartość chromosomu; w populacji mogą istnieć osobniki o tym samym genotypie (w przyrodzie np. bliźnięta jednojajowe);
fenotyp	- "wygląd" osobnika, jego rzeczywiste cechy określające stopień przystosowania do środowiska; mogą istnieć osobniki o tym samym fenotypie, ale różnych genotypach (ale nie odwrotnie!); o ile genotyp określa osobnika w języku wartości genów, o tyle fenotyp jest opisem osobnika w języku zadania - jako punktu w przestrzeni stanów;
krzyżowanie	- zjawisko wymiany części materiału genetycznego między osobnikami w procesie rozmnażania; *crossing-over, crossover*;
mutacje	- niewielkie, losowe zmiany w genotypie osobników, zapewniające zróżnicowanie genetyczne populacji;
nisza ekologiczna	- obszar przestrzeni stanów o specyficznych, dogodnych warunkach; odpowiada lokalnym maksimom funkcji celu;

Siła algorytmów genetycznych leży w tym, że w zasadzie nie musimy wiedzieć nic o funkcji celu: może to być "czarna skrzynka", do której wrzucamy punkt z przestrzeni stanów i otrzymujemy wynik. Funkcja celu może też mieć wiele maksimów lokalnych, może być funkcją wielu zmiennych, jej wartości mogą nawet być obarczone błędem losowym (jak przy pomiarach wartości fizycznych). Funkcji celu może nawet w ogóle nie być: będziemy rozpatrywali algorytmy genetyczne, w których jedyną rzeczą, jaką potrafimy powiedzieć o punktach przestrzeni stanów jest to, który z dwóch punktów jest lepszy.

Przykład 2.1. Niech dany będzie wytwórca lodów dążący do maksymalizacji swych dochodów. Wytwórca może decydować o smaku lodów, ich wielkości i obecności różnych dodatków (orzechy, polewa czekoladowa itp.). Przyjmijmy dla uproszczenia, że w grę wchodzą cztery smaki lodów: waniliowe, cytrusowe, czekoladowe, karmelowe; ponadto lody mogą mieć, lub nie mieć dodatków, oraz mogą być duże lub małe. Wszystkie te parametry mogą wpłynąć zarówno na cenę lodów, jak i na późniejszy popyt. Jakie lody powinien produkować wytwórca, by jego zysk (liczony jako iloczyn ceny i popytu) był maksymalny?

Przypuśćmy, że koszt wyprodukowania małych lodów o dowolnym smaku jest stały i wynosi 10. Koszt dodatków do lodów jest również stały i wynosi 5. Cena dużych lodów jest dwukrotnie wyższa.

Załóżmy ponadto, że dysponujemy badaniami określającymi zależności popytu na lody od ich smaku. W umownych jednostkach popytu wyniki tych badań przedstawiają się następująco:
Lody waniliowe: 6, z dodatkami: 6
Lody cytrusowe: 10, z dodatkami: 6
Lody czekoladowe: 8, z dodatkami: 12
Lody karmelowe: 10, z dodatkami: 12

Ze względu na wyższą cenę, lody duże cieszą się dwukrotnie mniejszym zainteresowaniem.

Dysponując powyższym kompletem informacji, możemy przystąpić do rozwiązywania problemu za pomocą algorytmu genetycznego. W pierwszym rzędzie należy zastanowić się, co będzie naszymi "osobnikami". Oczywiście, będą nimi lody, a właściwie ich rodzaje. Każdemu rodzajowi lodów (fenotypowi), odpowiadał będzie pewien zespół opisujących go cech (genów).

Tymi cechami będą: smak lodów (4 możliwości), ich wielkość (2 możliwości) i obecność dodatków (2 możliwości). Mamy więc łącznie 16 kombinacji cech lodów.

W klasycznym, najprostszym modelu, geny kodowane są za pomocą cyfr 0 i 1. W przypadku lodów, każdy ich rodzaj będzie opisywany przez cztery takie cyfry: dwie opisujące ich smak, jedna obecność dodatków i jedna wielkość. Przyjmijmy następującą metodę kodowania cech:

Lody waniliowe:		00	
Lody cytrusowe:		01	
Lody czekoladowe:		10	
Lody karmelowe:		11	
Nie ma dodatków:	0,	są dodatki:	1.
Lody małe:	0,	lody duże:	1.

Np.: 0011 - lody waniliowe w polewie czekoladowej, duże.

Następnym problemem do rozstrzygnięcia jest skonstruowanie funkcji przystosowania (fitness). Powinna to być funkcja przypisująca każdej kombinacji genów - każdemu rodzajowi lodów - pewną nieujemną wielkość, tym większą, im bardziej dany osobnik nam odpowiada. W naszym przykładzie, najbardziej naturalną miarą jakości lodów jest zysk z ich sprzedaży - iloczyn popytu i ceny.

Sposób eliminacji osobników słabszych określał będzie tzw. **algorytm "koła ruletki"**: w fazie reprodukcji szanse na przeżycie będą proporcjonalne do wartości funkcji przystosowania.

Do poprawnego działania algorytmu genetycznego potrzebujemy jeszcze określić sposób działania operatorów genetycznych: mutacji i krzyżowania. W przypadku kodowania binarnego, możemy użyć mutacji polegającej na losowej zamianie wartości pojedynczego bitu na przeciwny:

Np.: 0 0 0 0 0 1 0 0

Operacja krzyżowania jest trochę bardziej złożona. Najpierw wybieramy losowo parę rodziców, następnie losujemy punkt przecięcia - jeden z trzech punktów pomiędzy genami - jednakowy dla obojga rodziców, następnie konstruujemy potomka z początkowego fragmentu pierwszego rodzica i końcowego fragmentu drugiego. W końcu potomek zastępuje jednego z rodziców.

Np.:

Rodzic 1: 0 0 0 | 0

Potomek: 0 0 0 1

Rodzic 2: 1 1 1 | 1

Krzyżowanie będziemy przeprowadzać z prawdopodobieństwem 50%, mutację: 5% (dotyczy to każdego bitu z osobna).

Jesteśmy gotowi do uruchomienia algorytmu genetycznego. Ustalmy wielkość populacji na 4 osobniki. Populacja początkowa będzie składała się z osobników wybranych losowo:

Np.: 0111, 1000, 1000, 0100

Teraz powinniśmy zastosować operatory genetyczne. Jednak, ponieważ populacja składa się na razie z osobników losowych, nie ma to sensu - możemy przejść od razu do fazy reprodukcji.

Faza reprodukcji zaczyna się od policzenia wartości funkcji przystosowania każdego osobnika. Następnie, aby móc zastosować algorytm "koła ruletki", sumujemy wartości funkcji wszystkich osobników i wartości te dzielimy przez otrzymaną sumę - w ten sposób otrzymujemy procentowy

“udział” każdego osobnika w sumie. Wartości te sumujemy następnie narastająco - tworzymy dystrybuantę rozkładu osobników.

Nr	Osobnik	Wartość przystosowania	% udział	suma udziałów, narastająco
1	0111	90	25%	25%
2	1000	80	23%	48%
3	1000	80	23%	71%
4	0100	100	29%	100%
	Suma:	350	100%	

Występujące w tabeli wartości fitness - funkcji przystosowania obliczyliśmy na podstawie danych wejściowych: np. lody 0111 mają cenę 30 (gdyż mają dodatki i są duże), a popyt na nie wynosi 3 (cytrusowe z dodatkami: 6; lody są duże, więc popyt dzielony przez 2), tak więc zysk wyniesie $30\times3=90$.

Aby wybrać osobniki do następnej populacji, losujemy cztery liczby z zakresu [0,1) i w ostatniej rubryce sprawdzamy, któremu osobnikowi one odpowiadają. Wylosowaliśmy liczby: 0.22, 0.01, 0.15, 0.88. Odpowiadają one osobnikom: 1, 1, 1 i 4 (znajdujemy najmniejszą liczbę większą od wylosowanej i wybieramy odpowiadającego jej osobnika). Nowa populacja składa się zatem z osobników:

0111, 0111, 0111, 0100

Zgodnie z przewidywaniem, do następnej populacji dostały się osobniki najlepiej przystosowane.

Po fazie reprodukcji przychodzi czas na krzyżowanie. Dla każdego osobnika losujemy więc (z prawdopodobieństwem 50%), czy zostanie na nim zastosowany operator krzyżowania. Załóżmy, że wylosowaliśmy osobnika 1 i 4. Losujemy następnie, z którymi osobnikami mają się one skrzyżować - wylosowaliśmy, że parą dla 1 będzie osobnik 2, a parą dla 4 będzie 1.

W przypadku pierwszej pary wylosowaliśmy, że punktem krzyżowania będzie przerwa między trzecim i czwartym genem:

Osobnik 1: 0 1 1 | 1

Potomek: 0 1 1 1

Osobnik 2: 0 1 1 | 1

Ponieważ osobniki 1 i 2 mają identyczny zestaw genów (opisują ten sam rodzaj lodów), skrzyżowanie ich nie prowadzi do niczego nowego - jest to podstawowa cecha wszelkich operatorów krzyżowania. Inaczej będzie w drugim przypadku:

Osobnik 4: 0 1 0 | 0

Potomek: 0 1 0 1

Osobnik 1: 0 1 1 | 1

Wylosowaliśmy ten sam punkt krzyżowania, co poprzednio. Widzimy, że w populacji pojawił się nowy rodzaj lodów: po osobniku 4 odziedziczył on brak dodatków, po osobniku 1 - wielkość. Po zastąpieniu pierwszych rodziców potomkami, populacja składa się z osobników:

0111, 0111, 0111, 0101

Nadszedł czas na mutacje. Zwykle jest to operator stosowany rzadziej, niż krzyżowanie - w naszym przypadku prawdopodobieństwo wynosi 5%, czyli zmutowany zostanie średnio mniej, niż jeden gen w całej populacji (liczącej 16 genów). Tym razem jednak, losując kolejno dla każdego bitu liczbę z przedziału [0,1), za siódmym razem wylosowaliśmy liczbę mniejszą, niż 0.05. Zmutowany zostanie więc trzeci gen drugiego osobnika:

0111, 0101, 0111, 0101

W efekcie mutacji powstał taki sam osobnik, jak po krzyżowaniu czwartego osobnika z pierwszym. Ostatecznie, po drugim cyklu ewolucji skład populacji przedstawia się jak powyżej. Wartości funkcji przystosowania wynoszą odpowiednio 90, 100, 90 i 100 - są więc średnio wyższe, niż początkowe. Lody ewoluują w dobrym kierunku.

Po mutacji algorytm przechodzi do fazy reprodukcji, krzyżowania, mutacji, reprodukcji itd. Po kilku (pięciu? dziesięciu?) cyklach otrzymamy zestaw składający się z lodów przynoszących wytwórcy godziwe zyski - może nawet będą wśród nich lody idealne (łatwo sprawdzić, że najwyższą wartość funkcji przystosowania - 180 - uzyskują lody czekoladowe z orzechami lub karmelowe w polewie, dowolnej wielkości). Wytwórca może uruchomić seryjną produkcję i w krótkim czasie zbić fortunę.

Uwaga: Przedstawione w powyższym przykładzie parametry pracy algorytmu genetycznego odpowiadają występującym w praktycznych zastosowaniach. Jedynie wielkość populacji została zmniejszona do 4 (inaczej analiza byłaby niepotrzebnie skomplikowana), zwykle wynosi ona od kilkunastu do (w niektórych zastosowaniach) kilku tysięcy osobników. Poza tym prawdopodobieństwa mutacji i krzyżowania, a także sama konstrukcja operatorów genetycznych i sposób reprodukcji mogą być bez zmian używane w sytuacjach, gdy przestrzeń stanów jest na tyle duża, że uzasadnia stosowanie niekonwencjonalnych metod obliczeniowych.

Zadanie 2.1. Co w przykładzie 2.1 jest przeszukiwaną przestrzenią stanów? Jaka jest moc tej przestrzeni?

Zadanie 2.2. Załóżmy, że dodatkowym parametrem wpływającym na wielkość sprzedaży jest kolor opakowania - jeden z 8 możliwych. Ile rodzajów lodów należałoby wówczas rozpatrywać? Ile genów potrzeba, aby opisać tak rozszerzone pojęcie rodzaju lodów?

Zadanie 2.3. Czy możliwe jest, by po fazie reprodukcji całą populację opanował osobnik najgorzej przystosowany?

3. Implementacja prostego algorytmu genetycznego

Populację osobników będziemy reprezentować jako dwuwymiarową tablicę typu `int`: pierwsza współrzędna określała będzie numer osobnika w populacji, druga - wartość określonego genu. Ze względów technicznych będziemy potrzebowali dwóch takich tablic. Rozmiar tablic, czyli wielkość populacji i liczbę genów w chromosomie, przechowują stałe:

```
#define POP_SIZE   20
#define CHROM_SIZE 10

int pop[POP_SIZE][CHROM_SIZE];     // tablica osobników
int temp[POP_SIZE][CHROM_SIZE];    // pomocnicza tablica osobników
```

Taka metoda implementacji jest oczywiście dosyć rozrzutna: do przechowywania wartości 0 lub 1 używamy zmiennej typu `int`. O wyborze takim zadecydowały łatwość implementacji i elastyczność, ponadto w algorytmach genetycznych wielkość pamięci rzadko jest liczącym się ograniczeniem.

Oto funkcje realizujące podstawowe operacje genetyczne: losową inicjalizację osobnika, mutację i krzyżowanie:

```
void init(int t[],int n)
{
  for(int i=0;i<n;i++) t[i] = rand()%2;      // inicjalizacja losowa: 0 lub 1
};

void mutation(int t[],int n)
{
  int position = rand()%n;               // losowy gen osobnika
  t[position] = 1 - t[position];         // mutacja: 0 na 1 lub 1 na 0
};
void crossover(int t1[],int t2[],int n)
{
  int position = rand()%(n-1) + 1;
  for(int i=position;i<n;i++)
  {                                    // od losowego miejsca
    int p = t1[i];                     // zamieniamy miejscami geny osobników
    t1[i] = t2[i];
    t2[i] = p;
  };
};
```

Wszystkie te funkcje wymagają jako parametrów: wskaźnika na początek chromosomu i jego wielkości. Dzięki opisanej wyżej metodzie implementacji wywołanie tych funkcji może mieć postać np.: `mutation(pop[0],CHROM_SIZE)`, co oznacza mutację pierwszego osobnika w populacji.

Podobną składnię będzie miała funkcja przystosowania osobników. Przyjmijmy, że nasze zadanie polega na znalezieniu 10-elementowego ciągu bitów, którego kwadrat wartości średniej elementów jest maksymalny.

```
double fitness(int t[],int n)          // funkcja przystosowania
{
  double s=0;
  for(int i=0;i<n;i++) s+=t[i];
  s/=n;                               // przykład: zmaksymalizować kwadrat
  return s*s;                         //           średniej wartości genów
};
```

Będziemy potrzebowali jeszcze paru funkcji pomocniczych: kopiowanie osobnika w inne miejsce tablicy, wyświetlanie na ekranie zawartości chromosomu osobnika wraz z jego funkcją celu, oraz funkcji symulującej rzut "niesymetryczną monetą": zwracającej 1 lub 0 z ustalonym prawdopodobieństwem:

```
void copy(int dest[],int source[],int n)
{
  for(int i=0;i<n;i++) dest[i] = source[i];
};

void print(int t[],int n)     // funkcja pomocnicza: wypisz geny osobnika
{
  for(int i=0;i<n;i++) cout << t[i] << " ";
  cout << " = " << fitness(t,n) << endl;   //...i wartość jego funkcji celu
};

int flip(double p)      // pomocnicza: zwraca 1 z prawdopodobieństwem p
{
  return p<rand()/(double)(RAND_MAX+1);
};
```

Ostatnią funkcją pomocniczą będzie implementacja algorytmu "koła ruletki":

```
void RouletteWheel(double fitness[],int result[],int n,int m=-1)
{
  double *sum = new double[n];      // dystrybuanta rozkładu: kolejne sumy

  if(m==-1) m = n;            // domyślnie: m = n

  sum[0]=( fitness[0]>0 ? fitness[0] : 0 );
  for(int i=1;i<n;i++) sum[i] = sum[i-1]+( fitness[i]>0 ? fitness[i] : 0 );

  if(sum[n-1]>0)      // sum[n-1] przechowuje sumę wartości całej populacji
  {
    for(i=0;i<n;i++) sum[i]/=sum[n-1];   // normalizujemy rozkład do sumy 1
    for(i=0;i<m;i++)
    {
      double r=rand()/(double)(RAND_MAX+1); // r - losowe z przedziału [0,1)
      for(int j=0;sum[j]<r;j++);
      result[i] = j;            // j - numer pierwszego elementu tablicy sum
                                //     większego lub równego r
    };
  } else for(i=0;i<m;i++) result[i] = 0;
  delete [] sum;
};
```

Parametrami funkcji są kolejno: wskaźnik do tablicy z wartościami funkcji kosztu osobników, wskaźnik do tablicy, w której zapisane zostaną numery wylosowanych przez algorytm osobników, wielkość tablicy osobników i wielkość tablicy wynikowej. Funkcja została zaprojektowana z myślą o możliwie największej uniwersalności: co prawda zwykle wielkości tych tablic są jednakowe (taka też jest wartość domyślna ostatniego parametru), jednak nic nie stoi na przeszkodzie, żeby wylosować mniej lub więcej osobników, niż było dotychczas w populacji (np. gdyby wielkość populacji miała się zmienić). Wartości funkcji przystosowania są sumowane kolejno w pomocniczej tablicy `sum`, następnie tablica ta jest normalizowana, tzn. dzielona przez taką liczbę, by ostatnia komórka zawierała wartość 1,0. Jeżeli potraktujemy wartości funkcji przystosowania jako gęstość rozkładu prawdopodobieństwa, to w znormalizowanej tablicy `sum` znajduje się jego dystrybuanta. Następnie losowana jest wartość `r` z przedziału [0,1) i porównywana z kolejnymi wartościami tablicy `sum`. Pierwszy element tej tablicy większy od `r` wskazuje na osobnika podawanego jako wynik.

Jest to znana metoda odwracania dystrybuanty - przydatna, gdy chcemy wygenerować liczbę losową (w tym przypadku: numer osobnika) ze znanego rozkładu prawdopodobieństwa. Losowanie powtarzane jest wielokrotnie, a wyniki trafiają do tablicy `result`.

Czas na właściwy algorytm genetyczny, zawarty w funkcji `main()`:

```
void main()
{
  randomize();
                          // inicjacja losowa osobników:
  for(int i=0;i<POP_SIZE;i++) init(pop[i],CHROM_SIZE);

  const double mutation_prob = 0.1;  // prawdopodobieństwo mutacji osobnika
  const double crossover_prob = 0.5; // prawdopodobieństwo krzyżowania pary

  for(int t=0;t<100;t++)    // ograniczenie pętli - liczba pokoleń
  {
    for(i=0;i<POP_SIZE;i++)    // operatory genetyczne
    {
      if(flip(mutation_prob)) mutation(pop[i],CHROM_SIZE);
      if(flip(crossover_prob))
      {
        int second = rand()%POP_SIZE;         // losowy wybór partnera
        crossover(pop[i],pop[second],CHROM_SIZE);
      };
    };
    double f[POP_SIZE];        // wartości funkcji przystosowania osobników
    int selected[POP_SIZE];    // numery osobników wybranych "kołem ruletki"

    for(i=0;i<POP_SIZE;i++) f[i] = fitness(pop[i],CHROM_SIZE);

    RouletteWheel(f,selected,POP_SIZE);

    for(i=0;i<POP_SIZE;i++) copy(temp[i],pop[selected[i]],CHROM_SIZE);
            // kopiowanie wybranych osobników do tablicy pomocniczej
    for(i=0;i<POP_SIZE;i++) copy(pop[i],temp[i],CHROM_SIZE);
            // kopiowanie zawartości tablicy pomocniczej do podstawowej
  };
  cout << "Wynik - populacja końcowa:" << endl;
  for(i=0;i<POP_SIZE;i++) print(pop[i],CHROM_SIZE);
};
```

Przed rozpoczęciem pracy tablica osobników inicjowana jest losowo. Następnie program przechodzi do cyklu mutacji - krzyżowania - liczenia funkcji przystosowania - selekcji, zakończonego kopiowaniem wyselekcjonowanych przez funkcję `RouletteWheel` osobników do tablicy pomocniczej, a następnie kopiowaniem tablicy pomocniczej z powrotem do tablicy osobników. Korzystanie z tablicy pomocniczej jest konieczne, gdyż do końca procesu selekcji powinniśmy mieć możliwość dostępu do wszystkich osobników poprzedniej populacji.

Program wykona 100 kroków ewolucji (graniczna wartość pętli `for(int t…)`) i zatrzyma się, wyświetlając zawartość ostatniej populacji. Wynikiem działania programu - szukanym optimum - powinien być najlepszy osobnik z ostatniej populacji. Oczywiście może się zdarzyć, że optymalny osobnik pojawi się wcześniej, ale nie dożyje do ostatniego pokolenia - wykorzystywane w praktyce algorytmy powinny więc wyszukiwać i zapamiętywać najlepszego osobnika po każdym kroku.

Do prawidłowego działania programu wystarczy dołączyć odpowiednie pliki nagłówkowe i zadbać o inicjalizację generatora liczb losowych. W przypadku większości komplilatorów C++ wystarczy na początku programu dopisać:

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

#define randomize() srand((unsigned)time(NULL)) // zbędne np. w Borland C++
```

Po uruchomieniu programu, jednym z osobników wypisanych na ekranie powinien być osobnik optymalny (składający się z samych jedynek), lub jemu bliski. Szybkość zbieżności do rozwiązania nie jest imponująca, ale można łatwo temu zaradzić skalując wartości funkcji celu (mechanizm skalowania zostanie opisany w jednym z następnych rozdziałów).

Zadanie 3.1. Porównać skuteczność algorytmu (np. liczoną jako średnia z najlepszych wyników po 50 pokoleniach) dla różnych prawdopodobieństw mutacji i krzyżowania i dla różnych wielkości populacji.

4. Problem reprezentacji

Wszystkie dotychczas rozważane przykłady bazowały na pojęciu osobnika jako ciągu bitowego. Jeżeli naszym zadaniem jest np. optymalizacja funkcji działającej z przestrzeni ciągów binarnych, jej rozwiązanie za pomocą algorytmu genetycznego jest proste i naturalne. Co jednak zrobić, gdy problem polega na znalezieniu optymalnej trójki liczb rzeczywistych, najkrótszej drogi między przeszkodami czy funkcji pasującej do zadanych punktów? Wszystkie te problemy również można rozwiązać algorytmem genetycznym, jednak wymagają one właściwego zakodowania problemu, tzn. przetłumaczenia zadania na język chromosomów i funkcji przystosowania osobników do środowiska.

Ogólny zarys postępowania powinien wyglądać następująco:

1. Wybieramy sposób reprezentacji problemu - alfabet chromosomu.
2. Tworzymy funkcję przekształcającą chromosomy na punkty przestrzeni stanów.
3. W razie potrzeby modyfikujemy operatory krzyżowania i mutacji.
4. Ustalamy funkcję przystosowania.

Wyniki teoretyczne dają nam pewne wskazówki na temat szczegółowego postępowania na każdym z powyższych kroków. Kluczowym pojęciem w teorii algorytmów genetycznych jest pojęcie **schematu**:

Definicja: *Schematem* nazywamy ciąg o długości równej długości chromosomu, złożony ze znaków { 0, 1, * }. Miejsca zajęte przez znaki 0 lub 1 nazywamy *pozycjami ustalonymi* schematu. Mówimy, że ciąg binarny *pasuje* do schematu, jeżeli jest z nim identyczny na pozycjach ustalonych.

Do schematu { * 1 0 * } pasuje zarówno ciąg { 1 1 0 0 } jak i { 0 1 0 1 }, natomiast nie pasuje { 1 1 1 1 }. Zauważmy, że każdemu schematowi odpowiada pewien podzbiór ciągów binarnych (chociaż nie odwrotnie), o liczności 2^k, gdzie k jest liczbą jego nieustalonych pozycji.

Definicja: *Rzędem* o(S) schematu nazywamy liczbę jego pozycji ustalonych. *Rozpiętością* δ(S) schematu nazywamy odległość między skrajnymi pozycjami ustalonymi.

Załóżmy, że interesujemy się losem pojedynczego schematu podczas ewolucji. Przystosowanie takiego schematu można zdefiniować jako średnią wartość funkcji przystosowania pasujących do niego osobników.

Podstawowe twierdzenie teorii algorytmów genetycznych - **twierdzenie o schematach** - mówi, że podczas ewolucji liczba reprezentantów schematów dobrze przystosowanych, niskiego rzędu i niskiej rozpiętości rośnie wykładniczo. Schematy takie często nazywane są **cegiełkami**.

Intuicyjnie twierdzenie to nie wydaje się zaskakujące: schematy dobrze przystosowane mają większe szanse na wylosowanie w procesie selekcji, ich niski rząd zmniejsza prawdopodobieństwo zniszczenia wskutek mutacji, a niska rozpiętość zapobiega "rozcięciu" przez krzyżowanie. Kilka takich schematów, złożonych ze sobą w jednym osobniku, powinno dać optymalne rozwiązanie problemu - założenie takie nazywane jest **hipotezą o cegiełkach**.

Czas na praktyczne wykorzystanie wniosków płynących z tych rozważań. Prześledźmy kolejne kroki przekształcania zadania na język algorytmów genetycznych.

1. **Wybór alfabetu.**

Z hipotezy o cegiełkach wynika, że optymalne rozwiązanie składane jest ze schematów wygenerowanych przez algorytm genetyczny. Zależałoby nam więc na tym, by tych schematów składowych było jak najwięcej. Okazuje się, że wybór kodu binarnego w dotychczasowych rozważaniach nie był przypadkowy: to właśnie kod dwójkowy charakteryzuje się największą liczbą możliwych schematów w porównaniu z liczbą możliwych osobników.

Nie oznacza to jednak, że jedynym używanym w algorytmach genetycznych alfabetem jest kod binarny. Często sformułowanie problemu nasuwa inny, bardziej naturalny sposób kodowania. W dalszych rozdziałach zostanie przedstawionych wiele takich przykładów.

2. **Sposób reprezentacji przestrzeni stanów.**

Jest to często najtrudniejszy etap rozwiązywania problemu. Z teorii wynika, że powinniśmy się starać, by struktura problemu znalazła swe odbicie w sposobie reprezentacji. Przynajmniej niektóre schematy powinny mieć sensowną interpretację w języku zadania, a dodatkowo powinny one być schematami niskiego rzędu i niskiej rozpiętości - tak, by mogły stać się "cegiełkami", z których zbudowane zostanie rozwiązanie. W praktyce oznacza to, że jeżeli optymalizacji podlega kilka parametrów (a zwykle tak właśnie jest), odpowiadające im grupy bitów powinny leżeć obok siebie.

Zakodowanie pojedynczego parametru jest stosunkowo proste. Jeżeli jest to zmienna rzeczywista, dzielimy zakres jej zmienności na 2^l przedziałów i reprezentujemy je jako ciąg l cyfr binarnych. Jeżeli jest to parametr dyskretny o ustalonej liczbie wartości, przypisujemy im kolejne liczby naturalne i kodujemy na najmniejszej możliwej liczbie bitów. Np. jeżeli pewien parametr przyjmuje 6 różnych wartości, kodujemy go na trzech bitach (w dowolny sposób), przy czym dwie wartości nie będą wykorzystane. Jeżeli zdarzy się (np. na skutek mutacji), że na tych miejscach chromosomu pojawi się wartość niemożliwa do zinterpretowania, algorytm powinien takiego osobnika odrzucić (np. przez przypisanie funkcji przystosowania wartości 0).

Po zakodowaniu wszystkich parametrów, odpowiadające im ciągi łączymy w jeden chromosom.

Czasem zdarza się, że na kształt przestrzeni stanów wpływają dodatkowe ograniczenia (tzw. **więzy**): nierówności wiążące wartości zmiennych, dodatkowe warunki itp. W takiej sytuacji możliwe są dwa wyjścia. Można próbować przekształcić przestrzeń stanów tak, by mimo wszystko dało się opisać ją za pomocą iloczynu kartezjańskiego przedziałów. Jeżeli np. mamy zoptymalizować funkcję dwu zmiennych na kole jednostkowym, można przedstawić ją we współrzędnych biegunowych i kodować genetycznie wartość kąta i promienia. Drugim sposobem jest wprowadzenie tzw. **systemu kar**: jeżeli rozwiązanie kodowane przez dany chromosom narusza więzy, wartość funkcji przystosowania jest znacząco zmniejszane. Zmniejszenie to powinno być odpowiednio wyważone: jeżeli będzie zbyt duże, spowoduje to odrzucanie wszystkich osobników nie spełniających więzów. Nie jest to właściwe, ponieważ osobniki takie mogą być nosicielami genetycznie pożytecznych cech. Z drugiej strony nie można dopuścić, aby osobniki kodujące błędne rozwiązania zdominowały te, które nie naruszają więzów - jest to sytuacja niebezpieczna, gdyż może się okazać, że globalne optimum leży na obszarze "zabronionym".

3. **Dostosowanie operatorów mutacji i krzyżowania do sposobu reprezentacji.**

Jeżeli zrezygnujemy z kodowania binarnego, pociąga to za sobą konieczność skonstruowania nowych operatorów mutacji i krzyżowania. Podczas ich tworzenia powinniśmy pamiętać o następujących zasadach:

a) Operator mutacji powinien powodować niewielką modyfikację chromosomu, przy zachowaniu jego poprawności syntaktycznej. W miarę możliwości powinien on odzwierciedlać strukturę "sąsiedztwa" na przestrzeni stanów, o ile na danej przestrzeni taka struktura ma sens. W teorii algorytmów genetycznych przyjmuje się ponadto założenie, że za

pomocą ciągu mutacji można dowolny element przestrzeni stanów przekształcić w dowolny inny element.

b) Operator krzyżowania powoduje wymianę materiału genetycznego między dwoma osobnikami, lub też utworzenie trzeciego osobnika o chromosomie złożonym z fragmentów osobników rodzicielskich (procesy krzyżowania z udziałem więcej niż dwóch osobników nie były rozważane). Wymiana taka powinna zachowywać poprawność syntaktyczną chromosomów, a ponadto być operacją pozwalającą na przenoszenie i łączenie schematów (w sensie hipotezy o cegiełkach). Przeważnie zakłada się, że skrzyżowanie osobnika z samym sobą nie powinno powodować żadnych zmian.

4. **Ustalenie funkcji przystosowania.**

Jeżeli naszym zadaniem jest znalezienie ekstremum funkcji celu jednej lub wielu zmiennych, za naszą funkcję przystosowania możemy przyjąć po prostu funkcję celu. Należy przy tym pamiętać, że:

a) Funkcja przystosowania będzie maksymalizowana. Jeżeli naszym zadaniem jest znalezienie minimum funkcji celu, funkcja przystosowania powinna być jej odwrotnością, lub np. funkcją celu pomnożoną przez -1.
b) Funkcja przystosowania powinna być nieujemna. Jeżeli jest inaczej, możemy dodać do niej wystarczająco dużą stałą, albo wartości ujemne zamienić na 0.

Jeżeli w zadaniu optymalizacyjnym nie ma określonej funkcji celu, powinniśmy ją stworzyć sami. W przypadku optymalizacji wielokryterialnej wystarczy zwykle przyjąć kombinację liniową optymalizowanych parametrów składowych.

Jeżeli zadaniem naszym jest znalezienie pewnego punktu X przestrzeni stanów o określonych właściwościach (np. ciągu bitów równego danemu), powinniśmy stworzyć taką funkcję celu, która maksymalizuje się dokładnie na poszukiwanym rozwiązaniu. Oczywiście najprostszą taką funkcją będzie:

$$f(x) = \begin{cases} 1, & gdy\, x = X \\ 0, & wpp. \end{cases}$$

jednak takich funkcji powinniśmy unikać. Mimo, że formalnie poprawna, taka funkcja celu nie niesie żadnej informacji poza faktem, czy x jest rozwiązaniem. Algorytm genetyczny nie będzie miał żadnych wskazówek, czy ewolucja idzie w dobrym kierunku. Trudno w tej sytuacji oczekiwać, że okaże się on skuteczniejszy od np. błądzenia losowego. Prawidłowo skonstruowana funkcja celu powinna mieć tym większą wartość, im lepiej dany osobnik przybliża poszukiwane optimum.

Z inną sytuacją mamy do czynienia gdy nie dysponujemy funkcją celu, ale istnieje metoda pozwalająca rozstrzygnąć, który osobnik jest lepszy, a który gorszy. Tak będzie np. w przypadku porównywania strategii gry w określoną grę. W takiej sytuacji jedną z metod jest przeprowadzenie "turnieju" - porównywanie osobnika z kilkoma innymi i przyznawanie mu punktów za każdy przypadek, w którym okazał się lepszy. Można też użyć opisanego dalej systemu nadawania rang.

Przykład 4.1. Zoptymalizować funkcję:

$$f(x, y) = (x + y)^3 + \sin(x + y^2) + \cos(\sin(x) - xy)$$

na przedziale $x \in [-3 , 3]$, $y \in [0 , 5]$. Interesuje nas dokładność do trzeciego miejsca po przecinku.

Zakładana dokładność powoduje konieczność podziału zakresu zmiennej x na co najmniej 6000 przedziałów, a zmiennej y na co najmniej 5000 przedziałów. Jeżeli zdecydujemy się na kodowanie binarne, oznacza to konieczność zapisu obu wartości na 13 bitach (8192 przedziałów). Sposób interpretacji ciągów binarnych reprezentujących wartości zmiennych dany będzie wzorami:

$$x = -3 + \frac{6 \cdot b_x}{8192}$$

$$y = \frac{5 \cdot b_y}{8192}$$

gdzie b_x i b_y oznacza wartości ciągów binarnych kodujących zmienne x i y. Chromosom składał się będzie z 26 bitów: 13 pierwszych interpretowanych będzie jako b_x, 13 pozostałych jako b_y. Np. chromosom:

01100101100011001010001101

będzie interpretowany jako:

b_x = 0110010110001 = 3249
b_y = 1001010001101 = 4749

x = -0,620
y = 2,898

Jako funkcję przystosowania można przyjąć po prostu funkcję f (wartości ujemne zastępujemy zerami).

Zadanie 4.1. Danych jest n punktów z kwadratu jednostkowego [0,1]×[0,1]. Zaprojektować algorytm genetyczny znajdujący okrąg leżący najbliżej tych punktów. Użyć sumy kwadratów odległości punktów od okręgu jako wartości do zminimalizowania. Zakładana dokładność: 3 miejsca po przecinku.

Zadanie 4.2. Wprowadzić do zadania 1 dodatkowe ograniczenie: rozpatrywane okręgi muszą całkowicie zawierać się w kwadracie jednostkowym.

5. Modyfikacje i ulepszenia niezależne od problemu

Istotą działania algorytmów genetycznych jest wyważone połączenie zjawiska lokalnego poszukiwania dobrych rozwiązań z szerokim, częściowo losowym przeszukiwaniem całej przestrzeni stanów w celu znalezienia obiecujących "nisz ekologicznych". W praktyce stosowania algorytmów genetycznych natkniemy się na dwie podstawowe trudności: zbyt wolną lub zbyt szybką (przedwczesną) zbieżność algorytmu. Ze zbyt wolną zbieżnością mamy do czynienia wtedy, gdy średnia lub maksymalna wartość funkcji celu nie poprawia się dostatecznie szybko - na rozwiązanie trzeba czekać zbyt długo. Oczywiście takiej sytuacji powinniśmy unikać. Okazuje się jednak, że zbyt szybka zbieżność też jest zjawiskiem szkodliwym.

Zjawisko przedwczesnej zbieżności polega na tym, że ewolucja szybko prowadzi do rozwiązania, które jednak nie musi być optymalne. Wyobraźmy sobie funkcję celu o dwóch prawie równych maksimach lokalnych, rozdzielonych głęboką "przepaścią", tzn. obszarem o niskiej wartości funkcji kosztu. Jeżeli algorytm znajdzie osobnika realizującego niższy z tych "szczytów", cała populacja może zostać opanowana przez jego potomków. Szansa na znalezienie wyższego "szczytu", np. przez wielokrotną mutację, jest wówczas minimalna.

Częściowo problemy te rozwiązuje prawidłowe (zwykle, niestety, doświadczalne) dobranie prawdopodobieństw operacji takich, jak mutacja lub krzyżowanie, ewentualnie wprowadzenie nowych technik: np. można z niewielkim prawdopodobieństwem dodawać do populacji całkowicie losowego osobnika. Jednak zwykle to nie wystarczy. Dalsza część rozdziału poświęcona będzie krótkiemu opisowi niektórych modyfikacji algorytmu selekcji, jakie liczni autorzy proponowali na przestrzeni ostatnich trzydziestu lat.

5.1. Skalowanie funkcji celu

Skalowanie jest podstawową i bardzo często stosowaną techniką przyspieszającą zbieżność algorytmu. Polega ona na tym, że wartości funkcji celu poszczególnych osobników poddawane są modyfikacji, zanim staną się wartościami funkcji przystosowania i trafią do algorytmu selekcji (np. "koła ruletki"). O konieczności modyfikacji przekonuje następujący przykład: wyobraźmy sobie funkcję celu:

$$f(x) = 1000 + \text{liczba jedynek w ciągu } x$$

gdzie x jest 5-elementowym ciągiem binarnym. Wartość funkcji przystosowania najsłabszego osobnika wynosi 1000, a najsilniejszego 1005. Jeżeli wielkość populacji wynosi np. 10 i jest ona w połowie wypełniona przez osobniki minimalne, a w drugiej połowie przez osobniki maksymalne, po selekcji algorytmem "koła ruletki" w populacji znajdzie się średnio 5,012 osobnika maksymalnego. Oczywiście takie tempo zbieżności nas nie zadowala. Wystarczy jednak, by przed selekcją wykonać prostą operację:

$$f'(x) = f(x) - 1000$$

aby różnica prawdopodobieństw przeżycia między osobnikami silnymi i słabymi stała się znacząca. Tak zmodyfikowanej funkcji przystosowania używamy tylko do selekcji - gdy

znajdziemy rozwiązanie, przedstawiamy je np. użytkownikowi programu w postaci wartości funkcji celu.

Zasadę, na której opiera się skalowanie, można wyrazić następująco:
Aby mechanizm selekcji skutecznie eliminował słabsze osobniki i zastępował je silniejszymi, iloraz wartości funkcji przystosowania osobników silnych i słabych powinien być jak najwyższy.
Wynika z tego ciekawy wniosek: pomnożenie wartości funkcji przystosowania przez stałą nie zmienia właściwości algorytmu.
Oczywiście pojęcie "osobnik silny" lub "słaby" jest uzależnione od własności funkcji celu, od tego, jak dobry wynik chcemy osiągnąć, a także od czasu - osobniki silne na początku działania algorytmu mogą stać się stosunkowo słabe po kilku pokoleniach.

Jedną z najczęściej używanych technik skalowania jest **skalowanie liniowe**:

$$f'(x) = af(x) + b$$

przy czym parametry a i b dobiera się tak, aby średnia wartość f_{ave} w danej populacji pozostała nie zmieniona, natomiast maksymalna wartość funkcji przystosowania f'_{max} była pewną ustaloną wielokrotnością c maksymalnej wartości funkcji celu f_{max}:

$$f'_{max} = cf_{max}$$

$$f'_{ave} = f_{ave}$$

Rozwiązując ten układ równań otrzymujemy wzory na parametry a i b:

$$a = \frac{(c-1) \cdot f_{ave}}{f_{max} - f_{ave}}$$

$$b = cf_{ave} - af_{max}$$

Parametr c powinien wynosić od 1,2 we wczesnej fazie ewolucji (gdy zależy nam na większym zróżnicowaniu populacji) do 2,0 w poźniejszej fazie (gdy zależy nam na szybkiej zbieżności do rozwiązania). Ewentualne wartości ujemne f'(x) zastępowane są zerem. Ze skalowania rezygnujemy, jeśli cała populacja opanowana jest przez jednego osobnika, w związku z czym wartość średnia i maksymalna są sobie równe.
Obliczanie współczynników skalowania wymaga w każdym kroku ewolucji znajdowania maksymalnej i średniej wartości funkcji celu, ale jest to niewielki koszt obliczeniowy w porównaniu z zyskiem, jaki przynosi lepsza zbieżność algorytmu.
Inną metodą skalowania używaną w praktyce jest **skalowanie potęgowe**:

$$f'(x) = f^k(x)$$

gdzie k>1 jest liczbą wynoszącą u różnych autorów od 1,005 do 2. Metoda ta nie jest tak uniwersalna, jak poprzednia, gdyż liczba k zależy od własności funkcji celu - dla różnych problemów różne wykładniki dają lepsze rezultaty.

Kolejną metodą opartą na modyfikacji wartości funkcji celu jest technika **nadawania rang**. Polega ona na tym, że wartości funkcji celu osobników sortujemy rosnąco, a następnie traktujemy numery tak posortowanych osobników jako wartość funkcji przystosowania wykorzystywaną do selekcji. Tak więc, dla N-elementowej populacji, osobnik najsłabszy będzie miał wartość przystosowania równą 1, następny z kolei 2, a osobnik najsilniejszy N. Tak ustalone wartości mogą być następnie poddane innej formie skalowania, np. skalowaniu potęgowemu, czy prostemu przekształceniu liniowemu. Zaletą tej techniki są oczywista niewrażliwość na rzeczywisty kształt funkcji celu, wadą zaś względna pracochłonność procesu skalowania. Badania twórców tej metody wskazują, że jej praktyczna skuteczność jest zbliżona do innych metod skalowania.

Zadanie 5.1. Na początku rozdziału podano przykład 10-elementowej populacji opanowanej w połowie przez osobniki o wartości funkcji celu 1000, a w drugiej połowie o wartości 1005. Policzyć średnią liczbę lepszych osobników w populacji potomnej (selekcja algorytmem "koła ruletki") w przypadku użycia:

a) skalowania liniowego, c = 2;
b) skalowania potęgowego, k = 2;
c) nadawania rang od 1 do 10.

5.2. Modyfikacje algorytmu selekcji

W dotychczasowych rozważaniach przyjmowaliśmy klasyczny model selekcji oparty na algorytmie "koła ruletki". Zaletami tego modelu jest względna łatwość implementacji, a także łatwość analizy teoretycznej: prawdopodobieństwo wyboru danego osobnika do następnej populacji jest równe jego znormalizowanej wartości funkcji kosztu. Natomiast wadą jest jego duża losowość: posiadanie wysokiej wartości funkcji celu nie gwarantuje osobnikowi przeżycia. Poniżej przedstawiono przykłady alternatywnych metod selekcji.

Wybór losowy według reszt

W metodzie tej obliczana jest średnia liczba potomków danego osobnika w następnej populacji - jest to po prostu iloczyn znormalizowanej funkcji przystosowania i liczby osobników N w populacji. Następnie do populacji potomnej dołączanych jest tyle kopii każdego osobnika, ile wynosi część całkowita tej liczby. Reszta może zostać dopeniona na różne sposoby: można użyć do tego zwykłego algorytmu ruletki, lub algorytmu ruletki pracującego na pozostałych częściach ułamkowych. Tak skonstruowany algorytm daje nam gwarancję, że wszystkie osobniki o wartości funkcji przystosowania powyżej średniej przeżyją przynajmniej w jednym egzemplarzu.

Strategia elitarystyczna

Ta prosta modyfikacja polega na tym, że ustalona liczba (często tylko jeden) najlepszych osobników z populacji jest automatycznie przepisywanych do następnego pokolenia, a reszta dopełniana jest zwykłym algorytmem "koła ruletki". Badania nad skutecznością tej metody wykazały, że nie zawsze prowadzi ona do poprawy wyników, a najlepiej radzi sobie z najprostszymi typami zadań. Jednocześnie jednak wprowadzenie tej modyfikacji okazało się pożyteczne ze względu na otrzymywane dzięki niej wyniki teoretyczne.

Model wartości oczekiwanej

Jest to metoda oparta na zwykłym algorytmie “koła ruletki” z jedną modyfikacją: po wylosowaniu osobnika do następnej populacji, jego znormalizowaną wartość funkcji przystosowania zmniejszamy o 1/N (oczywiście nie dopuszczamy do powstawania wartości ujemnych). W ten sposób prawdopodobieństwo powtórnego wylosowania osobnika jest mniejsze, co zabezpiecza nas częściowo przed dominacją bardzo silnych osobników (zapobieganie przedwczesnej zbieżności).

Model stacjonarny (*steady state GA*)

Nazwa ta odnosi się do klasy algorytmów selekcji, w którym większa część populacji przechodzi nie zmieniona z pokolenia na pokolenie. W modelu tym z populacji wybieramy losowo dwa podzbiory osobników: grupę silniejszą (np. za pomocą algorytmu “koła ruletki”) przeznaczoną do reprodukcji i grupę osobników słabych. Następnie część słabsza jest wymazywana z pamięci i zastępowana potomstwem osobników z grupy silniejszej. Reszta populacji pozostaje nie zmieniona.

Zatłaczanie (*crowding*)

W modelu tym populację dzielimy na dwie nierówne części w sposób losowy. Następnie z części mniejszej wybieramy osobniki do reprodukcji za pomocą np. “koła ruletki”. Wylosowanymi w ten sposób osobnikami zastępujemy osobniki z drugiej części populacji, przy czym staramy się zastępować te osobniki, które są jak najbardziej podobne do osobników zastępujących. Miarą podobieństwa może być np. odległość Hamminga, czyli liczba genów, na których różnią się osobniki. Można też użyć miary odległości z przestrzeni stanów: jeżeli np. osobniki reprezentują parametry liczbowe, za odległość można przyjąć kwadrat ich różnicy.
Idea zatłaczania opiera się na fakcie, że w warunkach naturalnych jeżeli nisza ekologiczna zostanie przepełniona, osobnikom w niej przebywającym coraz trudniej jest znaleźć pożywienie i rozmnożyć się, mimo dobrego przystosowania. Algorytm z zatłaczaniem symuluje tę sytuację: staramy się nie dopuścić do zbyt licznej reprezentacji tylko jednej “niszy ekologicznej”, tzn. grupy dobrze przystosowanych osobników podobnych.

Zasada współudziału

Podobny efekt do algorytmu z zatłaczaniem możemy uzyskać wprowadzając pewne modyfikacje do wartości funkcji celu osobnika. Załóżmy, że mamy miarę odległości d(x,y) między osobnikami - np. jedną z opisanych w poprzednim rozdziale. Ustalamy pewną malejącą **funkcję współudziału** s(d) tak, by:

$$s(0) = 1$$

$$s(d_{\max}) = 0$$

gdzie d_{max} jest największą możliwą odległością między osobnikami. Przykład takiej funkcji:

$$s(d) = 1 - \left(\frac{d}{d_{\max}} \right)^k$$

dla k = 1, k = 1/2, k = 4 itd.

Następnie modyfikujemy wartość funkcji celu każdego osobnika:

$$f'(x) = \frac{f(x)}{\sum_{y \in P} s(d(x,y))}$$

gdzie P - aktualna populacja. W ten sposób wartość funkcji przystosowania jest zmniejszona tym bardziej, im więcej jest w populacji osobników podobnych do danego. Powoduje to większe różnicowanie się populacji i zapełnianie raczej kilku różnych "nisz ekologicznych", niż tylko jednej z nich.

Zadanie 5.2. Które z opisanych powyżej technik zwiększają, a które zmniejszają szybkość zbieżności w porównaniu ze standardowym "kołem ruletki"?

5.3. Bariery reprodukcyjne - modyfikacja algorytmu krzyżowania

W dotychczasowych rozważaniach przyjmowaliśmy model, w którym osobniki do krzyżowania dobierane były w sposób losowy. Okazuje się, że odejście od tego modelu w kierunku bardziej starannego doboru partnerów do krzyżowania może w efekcie przynieść wyniki podobne do modeli z zatłaczaniem czy współudziałem - populacja podzieli się na poszczególne "nisze ekologiczne". W tym celu wprowadzimy bariery reprodukcyjne, czyli pewne ograniczenia, jakimi podlegają pary podlegające krzyżowaniu, zabraniające krzyżowania ze sobą zbyt różnych osobników. Zapobiega to powstawaniu "degeneratów" (bardzo często skrzyżowanie dwóch całkowicie różnych, dobrze przystosowanych osobników daje w efekcie osobnika o bardzo słabym przystosowaniu), oraz powoduje podział populacji na "gatunki", czyli grupy podobnych osobników, nie krzyżujące się między sobą.

Najprostszym sposobem realizacji barier reprodukcyjnych jest liczenie odległości między osobnikami (można użyć jednej z miar odległości opisanych w poprzednich rozdziałach) i dopuszczanie krzyżowania tylko tych osobników, które nie są zbyt odległe. Można też używać **zewnętrznych wzorców kojarzeniowych**, tzn. arbitralnie ustalić podział na gatunki na podstawie podziału na pewne (być może częściowo nachodzące na siebie) klasy. Klasy takie mogą być opisane za pomocą schematów, np.:

* 1 0 * * *

jest opisem gatunku osobników charakteryzujących się wartością 1 na drugiej pozycji i 0 na trzeciej.

Idąc dalej tą drogą, możemy pozbyć się konieczności ustalania przez użytkownika sztywnych zasad podziału na gatunki, zatrudniając do tego celu algorytm genetyczny. Teraz **wzorce kojarzeniowe** stają się integralną częścią chromosomu:

* 1 0 * : 1 0 0 1

Jest to przykład chromosomu osobnika 4-bitowego (jest to tzw. część funkcjonalna chromosomu), z dołączonym na początku 4-elementowym wzorcem kojarzeniowym. Taki

osobnik może się krzyżować wyłącznie z osobnikiem, którego część funkcjonalna pasuje do wzorca. Można też zażądać zgodności obustronnej. Wszystkie operacje genetyczne prowadzone na osobniku dotyczą również wzorca kojarzeniowego. Dzięki temu zasady doboru partnerów podlegają również ewolucyjnemu udoskonalaniu.

Wadami powyższej metody są: konieczność modyfikacji algorytmów mutacji i krzyżowania, praca na rozszerzonym, 3-elementowym alfabecie, oraz dwukrotne powiększenie pamięci zajmowanej przez populację. Tę ostatnią wadę częściowo ogranicza inny model: oprócz krótkiego wzorca kojarzeniowego dołączamy do chromosomu **identyfikator kojarzeniowy** o tej samej długości, natomiast część funkcjonalna może być dużo dłuższa:

1 * * 0 : 0 0 1 1 : 1 0 0 1 1 0 1 1 0 0 0

wzorzec : identyfik. : część funkcjonalna

Następnie przy wyborze partnera porównujemy jedynie wzorzec jednego z osobników z identyfikatorem drugiego.

Zadanie 5.3. Rozszerzyć operatory mutacji i krzyżowania uwzględniając wzorce kojarzeniowe.

5.4. Diploidalny kod genetyczny

W biologicznych procesach ewolucyjnych dużą rolę odgrywa zjawisko **diploidalności**, tzn. posiadania przez osobniki dwóch zestawów genów. Każdą cechę koduje para genów, z których jeden jest **dominujący**, tzn. określający faktyczne występowanie u osobnika danej cechy. Drugi gen, którego wartość jest przysłaniana przez gen dominujący, nazywany jest genem **recesywnym**.

W implementacji komputerowej tego zjawiska, kod genetyczny osobnika składa się z podwójnego zestawu genów, przy czym poszczególne wartości (allele) należą do trójelementowego zbioru { 0, 1, 1_0 }. Symbol “1_0” oznacza “jedynkę recesywną”. W chwili liczenia funkcji celu osobnika, kod diploidalny tłumaczony jest na zwykły alfabet zerojedynkowy zgodnie z zasadą, że 1 dominuje nad 0, a 0 nad 1_0:

pierwszy zestaw	drugi zestaw	wynik
0	0	0
0	1	1
1	0	1
1	1	1
0	1_0	0
1_0	0	0
1	1_0	1
1_0	1	1
1_0	1_0	1

Proces selekcji nie różni się niczym od przypadku klasycznego. Zmianie ulega natomiast operator mutacji i procedura generowania osobników losowych (np. w celu utworzenia populacji początkowej). Aby w wynikowym ciągu binarnym (fenotypie) utrzymać jednakowe prawdopodobieństwa wystąpienia 0 i 1, rozkład symboli w chromosomie powinien być

następujący: 0 - 50%, 1 - 25%, 1_0 - 25%. Operator mutacji powinien zastępować wylosowany gen innym, wylosowanym według podanego wyżej prawdopodobieństwa. Taka operacja może nie zmienić genu - radą na to jest zwiększenie prawdopodobieństwa mutacji.

Operacja krzyżowania jest wzorowana na procesach naturalnych i przebiega w dwóch etapach. Najpierw obydwa osobniki rodzicielskie poddawane są wewnętrznemu krzyżowaniu (standardową metodą) posiadanych zestawów chromosomów, tworząc w ten sposób po dwie **haploidalne** (zawierające pojedynczy zestaw genów) gamety:

Osobnik:		Po krzyżowaniu:		Gamety:	
1 0		0 1		0	1
1 0		0 1		0	1
1 0	→	0 1	→	0	1
1 0		1 0		1	0
1 0		1 0		1	0

Następnie dwa diploidalne osobniki potomne powstają z połączenia parami czterech haploidalnych gamet pochodzących od rodziców:

Gamety rodziców:		Potomek:
0	1_0	0 1_0
0	1_0	0 1_0
0	1	0 1
1	1	1 1
1	1	1 1

Podobnie dla drugiego zestawu gamet.

W praktyce algorytmów genetycznych zjawisko diploidalności wykorzystywane jest w problemach z funkcją celu zmieniającą się w czasie. Doświadczenia przeprowadzone na przykładzie funkcji zmieniającej się drastycznie co kilkadziesiąt pokoleń wykazały, że o ile zwykły algorytm genetyczny zbiegał tylko do jednego z rozwiązań, o tyle algorytm z kodem diploidalnym potrafił szybko dostosowywać się do obydwu sytuacji. Można to wytłumaczyć tym, że w modelu diploidalnym ewolucja może przebiegać na dwóch poziomach: z jednej strony w warstwie dominującej wyszukiwany jest zestaw cech optymalnie odpowiadający aktualnym warunkom (funkcji celu), z drugiej strony znalezione rozwiązania po zmianie warunków mogą zostać zachowane w ukrytej warstwie recesywnej, gdzie są zabezpieczone przed zgubnym działaniem presji selekcyjnej. W razie kolejnej zmiany warunków algorytm nie musi szukać nowych rozwiązań - wystarczy, że ujawnią się ukryte dotychczas cechy. Podobną też rolę zjawisko diploidalności odgrywa w przyrodzie.

6. Strategie ewolucyjne

Grupa metod optymalizacji określanych jako strategie ewolucyjne powstała w latach sześćdziesiątych w Niemczech, niezależnie od rozwoju algorytmów genetycznych. Metody te oparte są na reprezentacji osobników jako wektorów liczb rzeczywistych, przy czym w chromosomie każdemu parametrowi zadania (zmiennej) odpowiadają dwie liczby: wartość zmiennej x oraz parametr jej zmienności σ:

$$((x_1, \dots x_k), (\sigma_1, \dots \sigma_k)) = (\mathbf{x}, \boldsymbol{\sigma})$$

Ewolucja postępuje według następującego schematu:

1. Populacja początkowa n-elementowa składa się z osobników losowych.
2. Za pomocą operacji mutacji i krzyżowania wytwarzanych jest k osobników potomnych.
3. Tworzona jest k+n-elementowa populacja przejściowa złożona z osobników starej populacji i osobników potomnych.
4. Populacja przejściowa jest redukowana do n elementów przez usunięcie k osobników o najgorszej wartości funkcji przystosowania f(**x**).
5. Kroki 2-4 powtarzane są aż do uzyskania satysfakcjonujących wyników.

Alternatywnie można przyjąć, że w punkcie 3. populacja przejściowa tworzona jest tylko z osobników potomnych - wówczas oczywiście k powinno być większe, niż n.

Operatory odpowiadające za powstanie osobników potomnych podobne są do znanych z algorytmów genetycznych operacji krzyżowania i mutacji:

- krzyżowanie polega na złożeniu nowego osobnika z wartości zmiennych przepisanych od losowego z rodziców wraz z parametrami σ:

  $$\begin{matrix} ((x_{1,1}, \dots x_{1,k}), (\sigma_{1,1}, \dots \sigma_{1,k})) \\ ((x_{2,1}, \dots x_{2,k}), (\sigma_{2,1}, \dots \sigma_{2,k})) \end{matrix} \rightarrow ((x_{a1,1}, \dots x_{ak,k}), (\sigma_{a1,1}, \dots \sigma_{ak,k}))$$

  gdzie $a1 \dots ak \in \{1, 2\}$.
  Alternatywnie rozważa się operator krzyżowania liczący średnie arytmetyczne z wartości zmiennych i parametrów σ.
- mutacja jest losową modyfikacją wartości zmiennej, przy czym wielkość zmiany jest uzależniona od odpowiadającego jej parametru σ:

  $$x = x + N(0, \sigma)$$

  gdzie N(0, σ) jest liczbą wylosowaną z rozkładu Gaussa o wartości oczekiwanej 0 i odchyleniu standardowym σ. Parametr σ również podlega mutacji:

  $$\sigma = \sigma \cdot e^{N(0, \Delta)}$$

  gdzie Δ jest stałą.

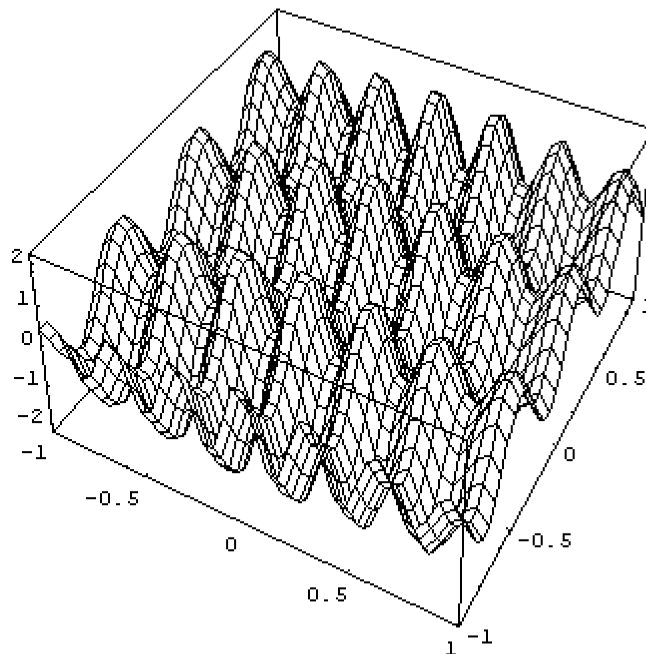
Podstawową różnicą między strategiami ewolucyjnymi a algorytmami genetycznymi jest całkowicie deterministyczny sposób selekcji nowej populacji. Ponadto o funkcji celu w strategiach ewolucyjnych nie musimy zakładać, że jest dodatnia, ani też nie musimy szukać jej maksimum - selekcja polega na porównaniu, które wartości funkcji celu bardziej nam odpowiadają. Odmienne są też podstawy teoretyczne obu metod: w miejsce twierdzenia o schematach można udowodnić odnoszące się do strategii ewolucyjnych **twierdzenie o zbieżności**. Twierdzenie to dowodzone było dla uproszczonego modelu algorytmu: funkcja celu jest ciągła, populacja składa się tylko z jednego osobnika, jedyną operacją jest mutacja, a parametr σ jest stały. Przy takich założeniach (plus kilka dodatkowych warunków), prawdopodobieństwo znalezienia optimum po dowolnie długim czasie jest równe 1.

Strategie ewolucyjne powstały jako metoda poszukiwania rozwiązań problemów o parametrach rzeczywistych i taka też jest ich główna dziedzina zastosowań obecnie.

Przykład 6.1. Zoptymalizować funkcję:

$$f(x,y) = \frac{\cos(20 \cdot x)}{x^2 + 1} + \sin(10 \cdot y) + \frac{y}{3}$$

w przedziale $x \in [-1,1]$, $y \in [-1,1]$.



Rys. 6.1. Wykres funkcji f(x,y). Maksimum: (0 , 0,78).

Funkcję tę co prawda da się analizować metodami standardowymi, jednak mnogość maksimów lokalnych znacznie utrudnia znalezienie maksimum globalnego.

Poniżej przedstawiono kompletny program rozwiązujący to zadanie metodą strategii ewolucyjnych. Program jest uniwersalny, tzn. proste przeróbki pozwalają na używanie go do optymalizacji innych funkcji o większej liczbie zmiennych.

Parametry zadania zapisane są w globalnych stałych i tablicach:

```
// Liczba zmiennych:
#define  K  2

                  // Zakres zmienności parametrów:
double zakres[K][2] = { { -1.0 , 1.0 },
                        { -1.0 , 1.0 } };

double f(double x[])    // Funkcja celu
{
  return 1/(x[0]*x[0]+1)*cos(x[0]*20) + sin(x[1]*10) + x[1]/3.0;
};
```

Do implementacji algorytmu przydatne nam będą pomocnicze funkcje: generująca liczbę losową z rozkładu Gaussa, oraz sortująca. Do generowania zmiennej losowej z rozkładu Gaussa o wartości oczekiwanej 0 i odchyleniu standardowym `sigma` użyto szybkiego algorytmu Box-Müllera:

```
double N(double sigma)  // Rozkład normalny - algorytm Box-Mullera
{
  double u1,u2,wyn;
  u1=rand()/(double)(RAND_MAX+1);
  u2=rand()/(double)(RAND_MAX+1);
  wyn=sqrt(-2*log(u1))*cos(2*3.14159*u2);
  return wyn*sigma;     // zwraca liczbę losową z rozkładu N(0,sigma)
}
```

Ze względu na niewielką liczbę osobników w populacji, użyto prostego algorytmu sortowania przez wstawianie. Składnia wywołania funkcji sortującej podobna jest do składni znanej z algorytmów genetycznych funkcji `RouletteWheel`: funkcja dostaje adresy dwóch tablic, przy czym pierwsza zawiera wartości funkcji celu osobników, w drugiej zaś funkcja umieszcza wynik swego działania: numery osobników posortowane względem wartości z pierwszej tablicy. W zależności od tego, czy zadanie polega na minimalizacji, czy maksymalizacji funkcji celu, należy użyć nierówności w odpowiednią stronę.

```
void insertion_sort(double *wart,int *tab,int n)
{
  for(int i=0;i<n;i++) tab[i]=i;
  for(i=1;i<n;i++)
    if(wart[tab[i]] > wart[tab[i-1]])      //  > - maksymalizacja
    {                                      //  < - minimalizacja
      int j=i;
      do
      {
        int p=tab[j];
        tab[j]=tab[j-1];
        tab[j-1]=p;
        j--;
      } while((j>0)&&(wart[tab[j]] > wart[tab[j-1]]));
                              // > - maksymalizacja
                              // < - minimalizacja
    };
};
```

Oto implementacja klasy `ES_osobnik` - pojedynczego osobnika w populacji. Osobnik zawiera w sobie wartości zmiennych, wartości ich odchyleń standardowych, oraz swoją wartość funkcji celu. To ostatnie pozwala nam zaoszczędzić trochę czasu: przy kopiowaniu osobnika nie musimy liczyć powtórnie jego wartości. Wartość parametru Δ zapisana jest jako globalny symbol `DELTA`.

```
#define DELTA 1.2

class ES_osobnik
{
public:
  ES_osobnik();                     // konstruktor inicjalizacji losowej
  ES_osobnik(ES_osobnik*);          // konstruktor kopiujący
  ES_osobnik(ES_osobnik*,ES_osobnik*);    // konstruktor krzyżujący
  ~ES_osobnik();

  void mutacja();
  double Fitness() { return fitness; };
  friend ostream &operator<<(ostream&,ES_osobnik*);  // prezentacja wyniku

private:
  double *x;            // tablica wartości zmiennych
  double *sigma;        // tablica odchyleń standardowych
  double fitness; // wartość funkcji celu
};

ES_osobnik::ES_osobnik()
{
  x=new double[K];
  sigma=new double[K];
  for(int i=0;i<K;i++)       // inicjalizacja losowa: rozkład jednostajny
  {                          // na zakresie zmiennej
    x[i] = zakres[i][0] +
            rand()/(double)(RAND_MAX+1) * (zakres[i][1]-zakres[i][0]);
    sigma[i] = (zakres[i][1]-zakres[i][0])/3.0;
  };
  fitness = f(x);      // użycie globalnej funkcji celu
};

ES_osobnik::~ES_osobnik()
{
  delete [] x;
  delete [] sigma;
};

ES_osobnik::ES_osobnik(ES_osobnik *second)  // kopiowanie osobnika
{
  x=new double[K];
  sigma=new double[K];
  for(int i=0;i<K;i++)
  {
    x[i] = second->x[i];
    sigma[i] = second->sigma[i];
  };
  fitness = second->fitness;
};

ostream &operator<<(ostream &out,ES_osobnik *o)
{
  for(int i=0;i<K;i++) out << o->x[i] << " (" << o->sigma[i] << ") ";
  out << "= " << o->fitness;
  return out;           // Wyświetlanie osobnika: wartość zmiennej, sigma,
                        // wartość funkcji celu.
};
```

Do implementacji pozostały jeszcze operatory modyfikacji osobników. Krzyżowanie przebiega według zasad opisanych wyżej. Algorytm mutujący zabezpieczony jest przed zdarzającym się

czasem niekontrolowanym wzrostem parametru `sigma`, jak również przed ucieczką wartości zmiennej poza dozwolony przedział:

```
ES_osobnik::ES_osobnik(ES_osobnik *first,ES_osobnik *second)
{
  x=new double[K];
  sigma=new double[K];
  for(int i=0;i<K;i++)          // krzyżowanie:
  {
    int r=rand()%2;            // numer osobnika, od którego ma pochodzić
                               // kolejna zmienna
    x[i] = ( r? first->x[i] : second->x[i] );
    sigma[i] = ( r? first->sigma[i] : second->sigma[i] );
  };
  fitness = f(x);               // aktualizacja wartości osobnika
};

void ES_osobnik::mutacja()
{
  for(int i=0;i<K;i++)
  {
    sigma[i] *= exp(N(DELTA));      // najpierw zmienia się sigma

    if(sigma[i]>zakres[i][1]-zakres[i][0])      // zabezpieczenie przed
      sigma[i]=zakres[i][1]-zakres[i][0];       // nadmiernym wzrostem

    double stare_x = x[i];
    for(int t=0;t<100;t++)      // pętla: 100 razy próbujemy utrzymać
    {                           // wartość zmiennej w zadanym zakresie
      x[i] = stare_x + N(sigma[i]);
      if( (x[i]<=zakres[i][1]) && (x[i]>=zakres[i][0]) ) break;
    };
    if(t==100) x[i] = zakres[i][0] +
      (zakres[i][1]-zakres[i][0])*rand()/(double)(RAND_MAX+1);
                // Jeżeli zmienna uparcie wychodzi poza zakres,
                // ustalamy losowo jej wartość.
  };
  fitness = f(x);
};
```

Właściwą ewolucją zajmuje się inny obiekt: `ES_populacja`.

```
#define WIELK_POP 10            // Wielkość populacji
#define NADMIAR   8             // Populacja pośrednia: WIELK_POP + NADMIAR

class ES_populacja
{
public:
  ES_populacja();               // losowa inicjalizacja osobników
  ~ES_populacja();

  void step();                  // pojedynczy krok ewolucji
  void raport();
  ES_osobnik *Najlepszy() { return pop[0]; };  // najlepszy osobnik jest
                                          // zawsze na początku tablicy
private:
  ES_osobnik **pop;             // tablica osobników
};



ES_populacja::ES_populacja()
{
  pop = new ES_osobnik * [WIELK_POP];
  for(int i=0;i<WIELK_POP;i++) pop[i] = new ES_osobnik;
```

```
};
ES_populacja::~ES_populacja()
{
  for(int i=0;i<WIELK_POP;i++) delete pop[i];
  delete [] pop;
};
```

Funkcja realizująca pojedynczy krok ewolucji działa według następującego schematu:

1. Tworzona jest populacja pośrednia o rozmiarze WIELK_POP + NADMIAR.
2. Aktualna populacja przenoszona jest na początek populacji pośredniej.
3. Reszta populacji (NADMIAR osobników) dopełniana jest osobnikami wytworzonymi przez operacje mutacji i krzyżowania na losowych osobnikach populacji wejściowej. Rodzaj zastosowanej operacji wybierany jest losowo: mutacja zachodzi z prawdopodobieństwem 2/3, pozostałe 1/3 oznacza krzyżowanie.
4. Wartości funkcji celu osobników wpisywane są do tablicy pomocniczej. Wartości te były policzone już wcześniej: każda operacja zmieniająca osobnika (mutacja, krzyżowanie, losowa inicjalizacja) kończy się liczeniem i zapamiętaniem wartości funkcji celu.
5. Osobniki w populacji pośredniej są sortowane ze względu na rosnące (lub malejące) wartości funkcji celu. Kolejność osobników po posortowaniu zapisana jest w tablicy pomocniczej.
6. Tworzona jest populacja końcowa przez skopiowanie WIELK_POP najlepszych osobników z populacji pośredniej. Ubocznym efektem tego procesu jest fakt, że osobniki w populacji końcowej są posortowane poczynając od najlepszego.
7. Kasowane są te osobniki z populacji pośredniej, które nie zostały wybrane do populacji końcowej.

```
void ES_populacja::step()
{
  double *fitness=new double[WIELK_POP+NADMIAR];  // tablica wartości f. celu
  int *numery=new int[WIELK_POP+NADMIAR];  // numery posortowanych osobników
  ES_osobnik **pop_posrednia=new ES_osobnik * [WIELK_POP+NADMIAR];

  for(int i=0;i<WIELK_POP;i++) pop_posrednia[i]=pop[i];
                // populacja aktualna przenoszona jest
                // na początek populacji pośredniej
  for(i=WIELK_POP;i<WIELK_POP+NADMIAR;i++)
    if(rand()%3!=0)  // reszta uzupełniana jest przez mutację i krzyżowanie
    {
      pop_posrednia[i]=new ES_osobnik(pop[rand()%WIELK_POP]);
      pop_posrednia[i]->mutacja();
    } else pop_posrednia[i]=
       new ES_osobnik(pop[rand()%WIELK_POP],pop[rand()%WIELK_POP]);

  for(i=0;i<WIELK_POP+NADMIAR;i++) fitness[i]=pop_posrednia[i]->Fitness();
                // Wartości funkcji celu zapisywane są w tablicy...
  insertion_sort(fitness,numery,WIELK_POP+NADMIAR);
                // ...i sortowane.
  for(i=0;i<WIELK_POP;i++)      // Najlepsze osobniki przenoszone są do
    pop[i]=pop_posrednia[numery[i]];    // populacji końcowej.

  for(i=WIELK_POP;i<WIELK_POP+NADMIAR;i++) delete pop_posrednia[numery[i]];
                        // Reszta populacji pośredniej jest usuwana.
  delete [] fitness;
  delete [] numery;
  delete [] pop_posrednia;
};

void ES_populacja::raport()
{
  cout << "Osobniki najlepsze, kolejno:" <<endl;
  for(int i=0;i<WIELK_POP;i++) cout << pop[i] << endl;
};
```

Oto przykład użycia tak zdefiniowanych klas:

```
void main()
{
  randomize();
  ES_populacja p;

  for(int t=0;t<30;t++)       // t - aktualny krok ewolucji
  {
    p.step();
    p.raport();
  };
  cout << "Najlepszy znaleziony:    " << p.Najlepszy() << endl;
};
```

Parametr Δ reguluje szybkością zbieżności algorytmu: duże wartości (powyżej 1,5) powodują, że program znajduje lokalne maksimum i szybko zbiega do dokładnego rozwiązania, wartości poniżej 1,0 powodują, że program wolniej znajduje rozwiązanie, ale unika fałszywych optimów lokalnych. Zależność tę potwierdza proste doświadczenie przy użyciu powyższego programu. Liczby w tabeli oznaczają procent przypadków, w których znalezione przez algorytm najlepsze rozwiązanie było prawie równe optimum globalnemu:

Δ	Liczba kroków ewolucji		
	30	100	250
2,0	50%	70%	75%
1,2	45%	77%	83%
0,7	37%	74%	90%

Przy okazji warto zauważyć wyraźną przewagę tej metody optymalizacji w porównaniu ze zwykłym algorytmem genetycznym. Dobre rozwiązania nie tylko znajdowane są szybko, ale przede wszystkim z niedostępną algorytmom genetycznym precyzją: zastosowanie reprezentacji zmiennoprzecinkowej uwalnia nas od konieczności wprowadzania sztucznego podziału przestrzeni na ograniczoną liczbę podprzedziałów i pozwala na uzyskanie dokładności ograniczonej w praktyce wyłącznie dokładnością typu `double`. Tajemnica skuteczności "końcowego dostrojenia" rozwiązania leży w tym, że przy prawidłowo dobranej wartości parametru Δ wraz z wartościami zmiennych ewoluują również wartości odchyleń σ odpowiedzialnych za wielkość ich zmian.

Zadanie 6.1. Zbadać zależności między wielkością populacji podstawowej i pośredniej a skutecznością algorytmu. Za miarę skuteczności można przyjąć parametr użyty w przykładzie do oceny doboru parametru Δ: liczymy, jak często (średnio, podczas 1000 uruchomień programu) program zbiega do maksimum globalnego (w porównaniu ze zbieżnością do innych maksimów lokalnych).

Zadanie 6.2. Wprowadzić dodatkowe ograniczenia na wartości x i y w funkcji celu: punkty (x,y) mają pochodzić z koła jednostkowego. Jak należy zmodyfikować program, aby uwzględniał to ograniczenie?

7. Przykłady zastosowań algorytmów genetycznych

W tym rozdziale zaprezentowane zostaną przykłady użycia algorytmów genetycznych do rozwiązywania konkretnych, nietrywialnych problemów. Analizując użyte metody warto zwrócić uwagę na dwie sprawy:

1. Większość prezentowanych algorytmów oparta jest o inny, niż binarny, sposób kodowania chromosomów. Okazuje się, że pomimo opisywanych w rozdziale 4. zalet kodowania binarnego, bardzo często sięga się po inne metody - takie, w których język chromosomu zbliżony jest do języka zadania. Oczywiście, takie "naturalne" kodowanie wymaga zdefiniowania dobrze działających operatorów mutacji i krzyżowania.
2. Wykorzystywane w praktyce algorytmy często są połączeniem czystych algorytmów genetycznych z innymi, charakterystycznymi dla konkretnego zadania technikami i znanymi przybliżonymi algorytmami deterministycznymi. Doświadczenia wskazują, że takie **algorytmy hybrydowe** są często najskuteczniejszą metodą radzenia sobie z trudnymi problemami.

7.1. Algorytmy genetyczne poszukujące strategii

Problem znalezienia skutecznej strategii gry w określoną grę jest zarówno trudnym, jak ważnym problemem. Problem ten jest ważny, gdyż pojęcie gier obejmuje nie tylko popularne gry logiczne lub losowe - w kategoriach teorii gier rozpatruje się wiele zjawisk ekonomicznych czy społecznych. O tym, że problem jest trudny, świadczy chociażby to, że nikt jak dotąd nie podał optymalnej strategii dla stosunkowo prostej gry, jaką są szachy. Jedną z przyczyn, dla których klasa problemów związanych z poszukiwaniem strategii jest trudna, jest rozmiar przestrzeni stanów. Przez strategię należy rozumieć przepis (funkcję), która dla każdego możliwego stanu gry (oraz ewentualnie jej historii) jednoznacznie określa optymalną decyzję gracza. Np. na planszy do gry w go może pojawić się ok. 3^{361} różnych sytuacji (mamy 361 pól, każde może być puste lub zajęte przez pionek biały lub czarny), a gracz ma do wyboru jeden ze średnio 180 możliwych ruchów. Łączna liczba różnych strategii jest więc równa:

$$180^{3^{361}} \approx 10^{10^{180}}$$

Jest to przestrzeń stanów o wiele za duża na to, by jej elementy jednoznacznie zakodować w postaci chromosomu mieszczącego się w pamięci dowolnego komputera. Dlatego też zajmiemy się grami o mniejszej, chociaż nadal bardzo dużej, liczbie kombinacji.

Przykład 7.1. Rekwizytami do pierwszej gry będzie stół i zbiór leżących na nim dowolnych, ale ustalonych przedmiotów (np. kamyków). Gra przeznaczona jest dla dwóch graczy wykonujących ruchy na przemian. Ruch gracza polega na zdjęciu ze stołu jednego lub dwóch kamyków. Wykonanie ruchu jest obowiązkowe, a grę przegrywa ten z graczy, który weźmie ostatni kamyk. Każdy z graczy zna aktualną liczbę kamyków na stole.

W tym przykładzie stan gry jest jednoznacznie wyznaczony przez liczbę kamyków pozostałych na stole. Ustalmy, że nasze poszukiwania strategii ograniczymy do gier z co najwyżej 50

kamykami. Ponieważ możliwych stanów gry jest 50, a gracz może podjąć jedną z dwóch decyzji, liczba możliwych strategii wynosi “tylko” 2^{50}. Przestrzeń stanów można więc zakodować jako 50-elementowy ciąg binarny:

$$\mathbf{x} = x_1 \, x_2 \ldots x_{50}$$

przy czym $x_n = 1$, gdy strategia nakazuje przy n kamykach na stole zabrać jeden kamyk, 0 gdy dwa.

Chromosom będzie się więc składał z 50 cyfr binarnych, co umożliwia użycie standardowych operatorów mutacji i krzyżowania. Ostatnim problemem pozostał więc sposób liczenia wartości funkcji celu. Niestety, jedyną uniwersalną metodą na porównywanie i ocenę skuteczności strategii jest zagranie według jej zasad przeciwko innej strategii. Operacja liczenia funkcji celu sprowadza się więc do zorganizowania w obrębie populacji turnieju - osobniki reprezentujące strategie grają przeciwko sobie. Za zwycięstwa w turnieju strategie otrzymują ustalone nagrody - liczby punktów, które później (po ewentualnym przeskalowaniu) stają się wartościami funkcji przystosowania używanymi w standardowym procesie selekcji.

Można również zastosować mniej czasochłonną, jednak bardziej przypadkową metodę selekcji: osobniki w populacji łączymy losowo w pary, każda para gra ze sobą, osobnik przegrywający jest usuwany z populacji, a następnie populacja jest uzupełniana przez potomków zwycięzców.

Eksperymenty komputerowe wykazują, że algorytm genetyczny dość szybko znajduje optymalną strategię.

Przykład 7.2. Iterowany dylemat więźnia.

W grze znanej jako **dylemat więźnia** uczestniczy dwóch graczy, którzy nie mogą się ze sobą komunikować. Gra polega na tym, że każdy z graczy równocześnie podejmuje jedną z dwóch decyzji: o współpracy z drugim graczem, lub o zdradzie. Wynikiem gry jest wypłata uzależniona od decyzji graczy. Przykład funkcji wypłaty ilustruje tabela (w rubrykach podano liczbę punktów przyznawanych graczowi 1 / graczowi 2):

	Gracz 2	
Gracz 1	Współpraca	Zdrada
Współpraca	6 / 6	0 / 10
Zdrada	10 / 0	1 / 1

Wartości wypłaty nie muszą być dokładnie takie, jak przedstawione wyżej. Ważne jest tylko to, że obustronna współpraca jest bardziej opłacalna od obustronnej zdrady, ale zdrada jest bardzo korzystna w przypadku współpracy drugiego gracza. Nazwa gry pochodzi z jej kryminalnej interpretacji (graczami jest dwóch więźniów oskarżonych o wspólne przestępstwo, współpraca oznacza milczenie w śledztwie, zdrada - zrzucenie winy na kolegę), ale sytuacje podobne do opisanej zdarzają się zadziwiająco często w polityce, socjologii, ekonomii czy psychologii.

Odmianą tego problemu jest **iterowany dylemat więźnia**: grę w dylemat więźnia powtarzamy wielokrotnie z tym samym graczem, pamiętając o wszystkich podjętych wcześniej przez obie strony decyzjach, jednak nadal bez możliwości komunikacji. Zadanie polega na znalezieniu strategii optymalizującej średnią wypłatę po wielu grach z innymi strategiami. Zanim poszukiwaniem takiej strategii zajęły się algorytmy genetyczne, za jedną z lepszych uznawana była strategia “wet za wet”: podczas pierwszej gry z nieznanym graczem współpracujemy z nim, a podczas wszystkich następnych powtarzamy ruch przeciwnika.

Przyjmijmy, że naszą strategię będziemy opierali na informacjach na temat trzech ostatnich ruchów. Każdą strategię można zakodować podobnie jak w poprzednim przykładzie, jako

binarną decyzję (np. 0 - zdrada, 1 - współpraca) podjętą w zależności od historii gry. Wynik każdego z trzech ostatnich posunięć może być jedną z czterech sytuacji (decyzji graczy): 00, 01, 10 i 11. Wszystkich możliwych historii jest więc $4^3 = 64$. Strategię można więc zapisać jako 64-elementowy ciąg binarny, w którym każdy bit określa decyzję gracza podjętą w zależności od tego, która z 64 możliwych historii poprzedza aktualny ruch. Na przykład, jeżeli gracze wykonali dotychczas ruchy (01)(01)(10), można tej historii przypisać numer $010110_2 = 22$, czyli odpowiedzią gracza powinna być wartość zapisana w 22 bicie.

Tak zdefiniowane pojęcie strategii nie obejmuje pierwszych trzech ruchów w grze z nowym partnerem. Do opisu strategii należy więc podać informację o pierwszym ruchu gracza (1 bit), drugim ruchu (4 bity, bo w pierwszym ruchu mogły zajść 4 różne sytuacje) i trzecim ruchu (16 bitów). Razem więc chromosom opisujący kompletną strategię powinien zawierać 85 bitów.

Ze względu na binarną reprezentację problemu, można użyć standardowych operatorów mutacji i krzyżowania. Podobnie jak w poprzednim przykładzie, wartości funkcji przystosowania określono jako wyniki turnieju, w którym oprócz osobników z populacji brały udział również strategie znalezione innymi metodami (w tym strategia "wet za wet").

Wyniki eksperymentu okazały się zaskakująco dobre. Algorytm genetyczny nie tylko sam doszedł do strategii "wet za wet", ale również znalazł inne, bardziej złożone strategie, zdolne skutecznie konkurować ze wszystkimi znanymi wcześniej.

7.2. Kodowanie zmiennoprzecinkowe

W przypadku problemów związanych ze znalezieniem wartości rzeczywistej (lub wektora takich wartości) stosuje się albo kodowanie binarne, albo bardziej naturalną reprezentację chromosomu jako ciągu liczb zmiennoprzecinkowych (floating point, FP-coding). Oczywiście to drugie podejście wymaga zdefiniowania operatorów mutacji i krzyżowania osobników.

Przykład 7.3. Problem kwantyzacji skalarnej.

Mamy dany rozkład prawdopodobieństwa H(z) na przedziale [a , b], ciągły lub dyskretny. Problem kwantyzacji skalarnej polega na znalezieniu ustalonej liczby n reprezentantów V(i) i n+1 granic przedziałów Z(i) (przy czym $V(i) \in [\, Z(i)\, ,\, Z(i+1)\,]$) tak, aby błąd średniokwadratowy kwantyzacji E był minimalny. Wartość błędu E dana jest wzorem:

$$E = \sum_{i=1}^{n} \int_{Z(i)}^{Z(i+1)} H(z) \cdot \big(z - V(i)\big)^2 \, dz$$

Z problemem kwantyzacji spotykamy się np. podczas redukcji liczby poziomów szarości obrazów cyfrowych. Wówczas rozkładem H jest histogram (rozkład) poziomów szarości obrazu a wybór reprezentantów V(i) oznacza wybór odcieni obecnych na obrazie wynikowym. Proces redukcji polega na tym, że jeśli piksel obrazu miał odcień szarości należący do przedziału [Z(i) , Z(i+1)], jego odcień na obrazie wynikowym staje się równy V(i). W tym dyskretnym przypadku we wzorze na E możemy zastąpić całki odpowiednimi sumami, co nie zmienia faktu, że liczenie błędu jest operacją dość czasochłonną.

Istnieje lokalnie skuteczna metoda rozwiązująca problem kwantyzacji. Jest nią algorytm centroidów, którego zasada działania oparta jest na twierdzeniu Maxa-Lloyda. Algorytm ten, wychodząc od dowolnej kwantyzacji, pozwala na jej iteracyjne polepszanie aż do osiągnięcia lokalnego minimum błędu. Niestety, algorytm jest pardzo podatny na wybór punktu startowego, co znacznie utrudnia znalezienie globalnego rozwiązania.

Aby jednoznacznie wyznaczyć kwantyzację, wystarczy przechowywać wartości V(i). Twierdzenie Maxa-Lloyda podaje wzór na optymalne wartości Z(i):

$$Z(i) = \frac{V(i-1) + V(i)}{2} \qquad \text{dla } i = 2 \,..\, n$$

natomiast Z(1) i Z(n+1) pokrywają się z końcami kwantyzowanego przedziału [a , b]. Tak więc chromosom reprezentujący kwantyzację składa się z kolejnych wartości V(i):

{ V(1), V(2), … V(n) }

Funkcją celu jest błąd średniokwadratowy kwantyzacji E, który powinien być minimalny. Za funkcję przystosowania można przyjąć:

$$f\big(V(1),\ldots V(n)\big) = \frac{1}{1 + E\big(V(1),\ldots V(n)\big)}$$

Podczas losowego tworzenia nowego osobnika, losowanych jest n wartości zgodnie z rozkładem prawdopodobieństwa H(z) (algorytmem podobnym do "koła ruletki"), następnie wartości te są sortowane i przypisywane kolejnym parametrom V(1) … V(n). Dzięki temu już na wstępie dobór reprezentantów oddaje w przybliżeniu rozkład H(z).

Mutacja polega na losowym przesunięciu jednej z wartości V(i). Nowa wartość jest losowana z rozkładu jednostajnego na odcinku [Z(i) , Z(i+1)].

Operator krzyżowania składa osobnika potomnego z k wartości pierwszego z rodziców i n-k wartości drugiego:

$$\begin{matrix} \{ V_1(1), \ldots V_1(n) \} \\ \{ V_2(1), \ldots V_2(n) \} \end{matrix} \rightarrow \{ V_1(1), \ldots V_1(k), V_2(k+1), \ldots V_2(n) \}$$

Liczba k wybierana jest losowo. Zauważmy, że nie zawsze taka operacja daje w wyniku ciąg posortowanych liczb. Jeżeli zdarzy się, że $V_1(k) > V_2(k+1)$, należy odwrócić kolejność rodziców: wówczas na pewno $V_2(k) < V_1(k+1)$.

Tak skonstruowany algorytm genetyczny używany był do redukcji obrazów cyfrowych (zdjęć) o 256 odcieniach szarości do 16 odcieni. Wyniki okazały się średnio znacznie lepsze od zwykłego algorytmu centroidów, ale czas pracy algorytmu genetycznego był dłuższy, a ponadto algorytm centroidów zapewniał dokładniejsze lokalne dostrojenie wyników.

Dopiero połączenie algorytmu genetycznego i algorytmu centroidów w jeden algorytm hybrydowy pozwoliło na znaczną poprawę skuteczności i wydajności kwantyzacji. Algorytm ten działał według następującego schematu: uruchamiany był opisany wyżej algorytm genetyczny, przy czym wartość błędu E liczona była nie na podstawie zapisanego w chromosomie schematu kwantyzacji, ale na podstawie tegoż schematu poddanego działaniu kilku kroków algorytmu centroidów. W ten sposób algorytm genetyczny wybierał ten schemat kwantyzacji, który jest najbardziej "obiecujący". Następnie najlepszy osobnik poddawany był ostatecznej optymalizacji algorytmem centroidów.

Oto inne przykłady operatorów genetycznych używanych przy kodowaniu zmiennoprzecinkowym:

- Mutacja: losowa zmiana jednej lub wszystkich wartości. Nowa wartość może zostać (jak w przykładzie wyżej) wylosowana z dozwolonego zakresu zmienności [a , b] jednostajnie, lub zgodnie z przekształceniem:

$$x := \begin{cases} x + \Delta(t, b - x) & z\ prawdop.\ 1/2 \\ x - \Delta(t, x - a) & z\ prawdop.\ 1/2 \end{cases}$$

$$\Delta(t, y) = y \cdot \left(1 - r^{\left(1 - \frac{t}{T}\right)^b}\right)$$

  gdzie r jest liczbą losową z przedziału [0,1], t jest numerem aktualnej generacji, T jest przewidywaną graniczną wielkością dla t, b jest parametrem równym np. 2. Tak skonstruowany operator wprowadza większe zmiany do chromosomu na początku działania algorytmu, mniejsze pod koniec.

- Krzyżowanie: oprócz użytego w przykładzie krzyżowania wzorowanego na klasycznym bitowym, można używać krzyżowania arytmetycznego: osobniki potomne składają się z parametrów będących średnimi ważonymi (za pomocą wspólnego współczynnika θ) parametrów rodziców:

  $x = \theta\ x_1 + (1-\theta)\ x_2$

  Z doborem operatora krzyżowania należy być ostrożnym: użycie powyższego operatora w problemie kwantyzacji skalarnej powodowało gwałtowne pogorszenie się wyników.

7.3. Kodowanie permutacji

Przykład 7.4. Problem komiwojażera.

Danych jest n miast połączonych (każdy z każdym) drogami o znanej długości. Znaleźć drogę o minimalnej długości przechodzącą jednokrotnie przez wszystkie miasta. Algorytm genetyczny używany był do rozwiązywania odmiany tego zadania znanej jako ślepe zagadnienie komiwojażera: dodatkowe ograniczenie polega na tym, że długość przebytej trasy możemy zmierzyć dopiero po odwiedzeniu wszystkich miast.

Naszymi osobnikami będą trasy komiwojażera w jego podróży po miastach. Jak łatwo zauważyć, każdą taką trasę można wyznaczyć jednoznacznie podając kolejność odwiedzin nażdego z n miast. Chromosom będzie więc się składał z permutacji n elementów.

Np.: { 1 7 4 3 6 2 5 }

Minimalizowaną funkcją celu jest długość trasy. Za funkcję przystosowania można więc wziąć np. odwrotność funkcji celu. Pozostaje jeszcze zdefiniowanie odpowiednich operatorów genetycznych.

Mutacja będzie polegała na wykonaniu pojedynczej, losowej transpozycji na elementach permutacji (tzn. zamiany miejscami dwóch losowych miast):

{ 1 2 3 4 5 } → { 1 5 3 4 2 }

Trudniej zdefiniować odpowiedni operator krzyżowania. Łatwo zauważyć, że metoda krzyżowania znana z reprezentacji binarnej nie jest właściwa: skrzyżowanie dwóch permutacji nie zawsze jest permutacją. Poniżej przedstawione zostaną trzy propozycje operatorów krzyżowania zaprojektowanych specjalnie dla permutacji.

Partially matched crossover (PMX).
Na wstępie losowane są dwa punkty wyznaczające w obu chromosomach rodzicielskich sekcję dopasowania:

{ 1 2 3 | 4 5 6 | 7 8 9 }
{ 1 3 5 | 7 9 2 | 4 6 8 }

W tym przykładzie sekcja dopasowania wyznaczona jest przez linie pionowe. Następnie wartości genów z sekcji dopasowania obu osobników łączone są w pary: (4,7)(5,9)(6,2). Pary te traktowane są jako transpozycje. Osobniki potomne powstają przez przekształcenie osobników rodzicielskich za pomocą tych transpozycji:

{ 1 2 3 4 5 6 7 8 9 } —(4,7)→ { 1 2 3 **7** 5 6 **4** 8 9 }
—(5,9)→ { 1 2 3 7 **9** 6 4 8 **5** }
—(6,2)→ { 1 **6** 3 7 9 **2** 4 8 5 }

Podobnie dla drugiego osobnika.

Order crossover (OX).
Sekcja dopasowania wybierana jest losowo, podobnie jak w metodzie PMX:

{ 1 2 3 | 4 5 6 | 7 8 9 }
{ 1 3 5 | 7 9 2 | 4 6 8 }

Tworząc potomka np. ciągu pierwszego, w miejsce jego sekcji dopasowania wstawiamy sekcję dopasowania drugiego ciągu:

{ # # # | 7 9 2 | # # # }

Następnie, wolne miejsca uzupełniamy nie wykorzystanymi jeszcze numerami z pierwszego osobnika, poczynając od pierwszego miejsca za sekcją dopasowania:

{ # # # | 7 9 2 | 8 # # } → … → { 4 5 6 | 7 9 2 | 8 1 3 }

Tak skonstruowany ciąg staje się osobnikiem potomnym. Drugi osobnik rodzicielski poddawany jest analogicznej operacji, tworząc drugi ciąg potomny.
O wyborze tej metody lub metody PMX powinna decydować specyfika problemu. O ile w metodzie OX krzyżowanie powoduje zachowanie części względnych położeń poszczególnych wartości (np. w powyższym przykładzie sekwencja 4 5 6 występuje zarówno u rodzica, jak i potomka, chociaż w innych miejscach), o tyle PMX zachowywał przede wszystkim bezwzględne pozycje (w poprzednim przykładzie liczby 1, 3 i 8 nie zmieniły położenia). Ponieważ dobrze

zbudowany operator krzyżowania powinien przenosić schematy - cegiełki, z których może zostać zbudowane rozwiązanie. W problemie komiwojażera, czymkolwiek nie byłyby cegiełki, bardziej ważna wydaje się względna kolejność odwiedzania miast niż to, które miasto zostanie odwiedzone na początku. Tak więc, odpowiedniejsza do tego problemu wydaje się operacja OX.

Cycle crossover (CX).

Ta operacja krzyżowania nie wymaga wyznaczania sekcji dopasowania. Jej podstawą jest założenie, że każda wartość w osobniku potomnym musi pochodzić od jednego z rodziców. Za ilustrację niech posłuży ten sam przykład, co poprzednio:

{ 1 2 3 4 5 6 7 8 9 }
{ 1 3 5 7 9 2 4 6 8 }

Krzyżowanie rozpoczynamy od losowego miejsca pierwszego chromosomu:

{ # # 3 # # # # # # }

Wartość 3 pochodzi od pierwszego z rodziców. Zauważmy, że u drugiego rodzica w tym miejscu występujeliczba 5. Oznacza to, że skoro to miejsce lest już zajęte, liczba 5 może w osobniku potomnym pochodzić tylko od pierwszego rodzica. Tak więc ustalamy następną wartość osobnika potomnego:

{ # # 3 # 5 # # # # }

Rozumując podobnie jak poprzednio dochodzimy do wniosku, że wartość 9 również musi pochodzić od pierwszego z rodziców:

{ # # 3 # 5 # # # 9 }

Podobnie dla 8, 6 i 2:

{ # # 3 # 5 # # 8 9 }
{ # # 3 # 5 6 # 8 9 }
{ # 2 3 # 5 6 # 8 9 }

Na miejscu wartości 2 w drugim ciągu występuje użyta już wartość 3. W tym momencie nie mamy już żadnych ograniczeń i możemy pozostałe miejsca ciągu potomnego wypełnić wartościami wziętymi z drugiego osobnika:

{ 1 2 3 7 5 6 4 8 9 }

Operator CX, podobnie jak PMX, zachowuje przede wszystkim bezwzględne pozycje wartości. Przedstawiony poniżej program realizuje krzyżowanie permutacji metodą PMX:

```
void PMX(int i1[],int i2[],int i[],int n)
                        // Parametry wejściowe:
                        // i1[], i2[]  - osobniki rodzicielskie
                        // i[]   - osobnik potomny
                        // n     - wielkość osobników
{
  int n1=rand()%n;      // losowane są dwa punkty - granice
  int n2=rand()%n;      // sekcji dopasowania
  int p;
```

```
  if(n1>n2) { p=n1; n1=n2; n2=p; };

  for(int k=0;k<n;k++) i[k]=i1[k];  // osobnik potomny jest modyfikacją
                                    // pierwszego z rodziców
  for(int m=n1;m<n2;m++)
  {
    int u=0;      // zmienna pomocnicza
    for(k=0;k<n;k++)          // pętla przez kolejne transpozycje
    {
      if(i[k]==i2[m]) { i[k]=i1[m]; u++; }            // zamiana
      else if(i[k]==i1[m]) { i[k]=i2[m]; u++; };
      if(u==2) break;         // koniec pętli, jeśli obie zamiany wykonane
    };
  };
};
```

Przykład 7.5. Wyszukiwanie krótkich reduktów w bazach wiedzy.

Definicja: Przez *bazę wiedzy* będziemy rozumieli zbiór m *obiektów* o_1, … o_m, każdy z nich opisany przez wartości N *atrybutów* a_1, … a_N, mogących przyjmować dowolne wartości: liczbowe, opisowe itp. Każdemu obiektowi przypisana jest dodatkowo wartość specjalnego atrybutu zwanego *decyzją* d.

Tak zdefiniowane pojęcie bazy wiedzy obejmuje np. bazę danych o pacjentach zawierających informacje o wynikach badań (atrybuty) i rozpoznanej chorobie (decyzja), lub bibliotekę cyfrowych obrazów zawierającą zeskanowane litery pisane ręcznie przez różne osoby (atrybutami mogą być np. wartości poszczególnych pikseli), wraz z informacją, o jaką literę chodzi (decyzja). Ważnym i trudnym zagadnieniem jest opracowanie metod pozwalających na podstawie wartości atrybutów przewidzieć wartość decyzji. Takie metody pozwoliłyby np. na automatyczne diagnozowanie chorób na podstawie symptomów, czy skuteczne czytanie pisma odręcznego.

Jako jeden z kroków prowadzących do znalezienia reguł łączących wartości atrybutów i decyzję, wprowadzono pojęcie **reduktu**:

Definicja: *Reduktem* bazy wiedzy nazywamy taki podzbiór atrybutów, który wystarczy do rozróżnienia obiektów ze względu na decyzję. To znaczy, że jeśli dwa obiekty mają różną wartość decyzji, to na pewno istnieje taki atrybut należący do reduktu, na którym te obiekty mają różną wartość. Dodatkowo zakłada się, że żaden właściwy podzbiór reduktu nie ma tej własności, tzn. redukt jest lokalnie minimalny. Jeżeli podzbiór spełnia wszystkie powyższe warunki z wyjątkiem warunku minimalności, nazywany jest wówczas *nadreduktem.*

Znaczenie reduktów jest oczywiste: skoro atrybuty należące do reduktu wystarczą do jednoznacznego wyznaczenia decyzji, możemy w procesie poszukiwania praw łączących atrybuty i decyzję ograniczyć się tylko do reduktu. Mimo, że lokalnie redukt jest minimalnym podzbiorem rozróżniającym obiekty, to globalnie może być kilka różnych reduktów. Oczywiście w praktyce najbardziej przydatne są jak najkrótsze redukty.

Problem, do którego rozwiązania użyto algorytmu genetycznego, polega na znalezieniu w danej bazie wiedzy reduktu o możliwie najmniejszej liczbie atrybutów. W pierwszym proponowanym rozwiązaniu zdecydowano się na najbardziej naturalny sposób kodowania: skoro szukamy podzbioru atrybutów spełniających pewne warunki, niech osobnikiem będzie N-elementowy ciąg binarny:

1 0 0 1 0 - odpowiada podzbiorowi złożonemu z pierwszego i czwartego atrybutu.

Takie kodowanie pozwoliło na wykorzystanie standardowych operatorów krzyżowania i mutacji. Funkcja celu osobnika (podzbioru atrybutów) powinna być uzależniona od liczby atrybutów (jak najmniejszej) i czynnika odpowiadającego za poszukiwanie ciągów odpowiadających reduktom:

$$f(x) = \frac{N - L_x}{N} + \frac{C_x}{C} + 0.5 \cdot R_x$$

gdzie L_x jest liczbą jedynek w chromosomie (tzn. liczbą atrybutów w podzbiorze), C jest liczbą par obiektów o różnych wartościach decyzji, C_x jest liczbą takich par rozróżnianych przez badany podzbiór atrybutów, R_x jest zmienną równą 1, gdy podzbiór jest reduktem lub nadreduktem (tzn. gdy $C=C_x$). Tak skonstruowana funkcja przystosowania ma tym większą wartość, im bardziej dany podzbiór atrybutów przypomina redukt, tzn. im więcej rozróżnia par obiektów. Z drugiej strony, dzięki dodatkowemu czynnikowi w sumie, podzbiory będące reduktami mają wyraźną przewagę nad innymi osobnikami, a jednocześnie różnią się między sobą czynnikiem odpowiedzialnym za liczność podzbioru.

Tak skonstruowanego algorytmu można używać do szukania reduktów nawet dużych baz wiedzy (kilkadziesiąt atrybutów, kilka tysięcy obiektów). Jednak metoda taka ma pewne wady: nie ma pewności, czy otrzymany podzbiór rzeczywiście jest reduktem, a nie nadreduktem.

Druga metoda powstała z myślą o wyeliminowaniu tej wady. Jest to typowy przykład algorytmu hybrydowego, łączącego w sobie zalety szybkich, przybliżonych heurystyk i algorytmów genetycznych. Rozpatrzmy następujący algorytm heurystyczny poszukiwania reduktu w bazie wiedzy:

1. Niech R będzie zbiorem wszystkich atrybutów uporządkowanych w pewnej określonej kolejności. Niech i = 1.
2. Ze zbioru R usuwamy i-ty atrybut.
3. Jeżeli podzbiór atrybutów R nie spełnia warunku rozróżnialności (tzn. przestał być nadreduktem), do R wstawiamy z powrotem i-ty atrybut.
4. Jeżeli $i<N$, $i=i+1$, idź do 2.

Po zakończeniu pracy algorytmu w zbiorze R znajduje się redukt. Jest on na pewno minimalny, gdyż gdyby można było zrezygnować z któregokolwiek atrybutu, zostałby on wyrzucony w jednym z kolejnych kroków. Niestety, ta stosunkowo prosta metoda nie zawsze pozwala na znalezienie reduktu globalnie minimalnego. Jednocześnie jednak, jeżeli w pierwszym kroku ustalimy odpowiednią kolejność wyrzucania atrybutów, na pewno w wyniku możemy otrzymać dowolny (a więc również minimalny) z reduktów.

Algorytm genetyczny posłuży właśnie do znalezienia tego najlepszego ustawienia atrybutów na wejściu do algorytmu heurystycznego. Osobnikami będą N-elementowe permutacje. Np.:

{ 3 2 1 5 4 }

oznacza, że algorytm heurystyczny najpierw spróbuje usunąć atrybut o numerze 3, potem 2 itd. Operacja liczenia wartości funkcji celu polega więc na uruchomieniu algorytmu heurystycznego, znalezieniu reduktu odpowiadającego zakodowanej w chromosomie permutacji i policzeniu jego długości. Funkcją przystosowania jest po prostu odwrotność długości reduktu. Jako operatorów genetycznych użyto opisanego wyżej krzyżowania PMX i mutacji jako losowej transpozycji elementów permutacji. Wielkość populacji: 20-50 osobników.

Tak skonstruowany algorytm genetyczny po 15-30 krokach produkował permutację pozwalającą algorytmowi heurystycznemu na znalezienie najkrótszego reduktu. Oczywiście, na wyjściu programu pojawiała się nie permutacja wynikowa, ale odpowiadający jej redukt. Ta

metoda okazała się bardziej czasochłonna od poprzedniej, ale jednocześnie dawała dużo lepsze wyniki - a w dodatku wszystkie będące reduktami.

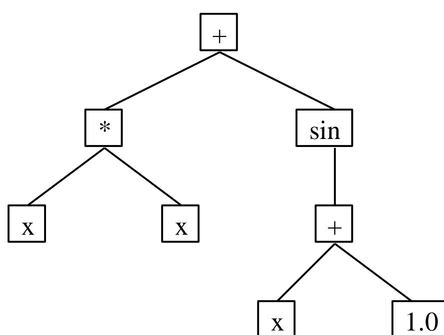
Powyższy przykład różni się w zasadniczy sposób od algorytmów genetycznych opisywanych wcześniej. Dotychczas zawsze osobnik kodował przykładowe rozwiązanie zadania, a algorytm genetyczny służył do tego, by znaleźć osobnika kodującego rozwiązanie optymalne. W ostatnim przykładzie rozwiązanie poszukiwane było przez inny, deterministyczny algorytm. Natomiast algorytm genetyczny służył do takiej modyfikacji przestrzeni stanów (przez zmianę porządku atrybutów), by algorytm heurystyczny mógł skutecznie w niej pracować i znaleźć rozwiązanie globalnie optymalne.

7.4. Programowanie genetyczne

W poprzednim przykładzie rozważaliśmy algorytm genetyczny, który sterował pracą innego programu (algorytmu heurystycznego). Następnym krokiem jest napisanie algorytmu genetycznego piszącego inne programy. W ten sposób otwiera się przed nami kusząca perspektywa samoprogramujących się maszyn.
Zanim jednak nasze algorytmy będą w stanie napisać program w języku C, przyjrzyjmy się prostszemu problemowi.

Przykład 7.6. Danych jest n punktów na płaszczyźnie. Znaleźć funkcję przebiegającą jak najbliżej nich, przy czym w jej zapisie można używać zmiennej x, stałej 1.0, czterech działań, operacji sin, cos, tan, log, e^x, dowolnie zagnieżdżonych. Zapis funkcji powinien być jak najkrótszy.

Naturalnym sposobem reprezentacji złożonych wzorów funkcji jest zapis w postaci drzew. Np:

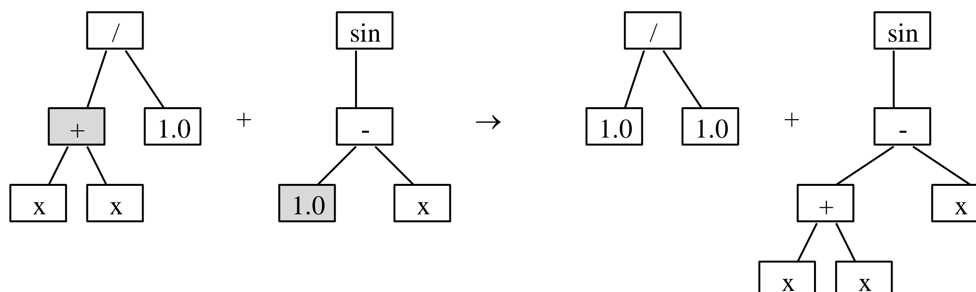


odpowiada funkcji $x^2 + \sin(x+1)$. Taka też jest postać chromosomu - w praktycznej implementacji jest on strukturą dynamiczną mogącą przyjmować prawie dowolny kształt i rozmiar (przeważnie przyjmuje się jakieś ograniczenie na głębokość drzewa, np. 17 poziomów). Elementami drzewa mogą być dwa rodzaje symboli: terminalne, nie generujące dalszych odnóg, umieszczone w liściach drzewa (tutaj: zmienna x i stała 1.0) i nieterminalne, z których wyrastają dalsze poddrzewa (tutaj: jednoargumentowe funkcje sin, cos, tan, log, exp i dwuargumentowe operacje +, -, *, /).

Jako funkcję celu można przyjąć np. średniokwadratowy błąd przybliżenia punktów przez funkcję. Policzenie wartości funkcji celu wymaga więc rozkodowania chromosomu i

podstawienia serii wartości pod zmienną x. Za funkcję przystosowania osobnika przyjmuje się odwrotność tego błędu.

W algorytmach genetycznych z chromosomem w postaci drzewa najważniejszą operacją genetyczną jest krzyżowanie. Polega ono na wylosowaniu dwóch węzłów z drzew rodzicielskich i zamianie miejscami poddrzew o korzeniach w wylosowanych węzłach:



Tak zdefiniowana operacja może spowodować znaczne powiększenie drzewa - musimy więc zawsze kontrolować, czy nie przekraczamy ustalonego limitu głębokości. Krzyżowanie drzew nie ma jednej z podstawowych cech operatorów krzyżowania znanych z innych reprezentacji: skrzyżowanie osobnika z samym sobą może doprowadzić do powstania całkiem nowych osobników.

Inne przeprowadzane na drzewach operacje mają znaczenie drugorzędne. Mogą to być:

- mutacja: zamiana pojedyńczego operatora na inny, ewentualnie zamiana całego wybranego losowo poddrzewa na inne, utworzone z losowych elementów.
- permutacja: wybierany jest dowolny węzeł wewnętrzny drzewa, tzn. nie będący jego liściem. Jeżeli od węzła odchodzi n gałęzi, ich porządek jest zmieniany w losowy sposób. Jeżeli wylosowany węzeł oznaczał operację przemienną, permutacja nie ma widocznych efektów. Jednak np. w przypadku dzielenia, operator ten powoduje zamianę miejscami licznika i mianownika.
- edycja: operacja upraszczająca drzewo przy wykorzystaniu określonych, prostych reguł. Np. w przypadku funkcji opisanych wyżej, edycja może polegać na zamianie poddrzewa x/x na stałą 1.0.

Tak zaprojektowany algorytm genetyczny potrafi znaleźć nawet bardzo złożone formuły. Charakterystyczny jest brak w spisie dopuszczalnych operatorów stałych różnych od 1.0. Okazuje się jednak, że algorytm potrafi radzić sobie bez nich: po prostu sam tworzy potrzebne stałe w formie np: (1.0 + 1.0 + 1.0)*(1.0 + 1.0). Można dopuścić takie operatory edycji, które takie wyrażenie zamienią na stałą równą 6.0.

Oto inne przykłady zastosowania struktur drzewiastych w algorytmach genetycznych:

- Konstrukcja formuł logicznych o określonych właściwościach. Dozwolone operatory: (AND, OR, NOT), symbole terminalne: (x_1, … x_n).
- Odwracanie funkcji. Problem podobny do opisanego w przykładzie, przy czym przybliżane punkty pochodzą z odwracanej funkcji.
- Sterowanie układem fizycznym. Mamy dany układ fizyczny reagujący na bodźce zewnętrzne - np. utrzymywany przez ramię robota stos talerzy w równowadze chwiejnej. Naszym zadaniem jest takie sterowanie parametrami układu fizycznego, by inne jego parametry mieściły się w zadanych granicach - np. by talerze nie spadły na ziemię. W tym celu musimy znaleźć najodpowiedniejszą funkcyjną zależność między stanem układu a sterowanymi parametrami -

stosujemy do tego celu algorytm genetyczny. Funkcję kodujemy za pomocą alfabetu podobnego do użytego w przykładzie. Symbole terminalne: lista zmiennych - parametrów wejściowych. Policzenie funkcji celu polega na wykonaniu symulacji układu fizycznego sterowanego daną funkcją. Jako wartość funkcji celu możemy przyjąć np. uśrednione po czasie odchylenie parametrów od zakładanych wartości.

- Sztuczna mrówka. Mamy daną planszę kwadratową 32×32 pola, ze sklejonymi krawędziami (torus). Na planszy rozmieszczonych jest 89 sztuk "żywności", zajmujących po jednym polu, ułożonych wzdłuż krętych i dziurawych ścieżek. Tytułowa mrówka jest automatem poruszającym się po planszy, wyposażonym w sensor wykrywający żywność na polu bezpośrednio przed nim. Automat ten należy zaprogramować tak, by zebrał w określonej liczbie (np. 400) kroków jak największą ilość żywności. Język programowania składa się z rozkazów (symboli terminalnych) NAPRZÓD, W_LEWO, W_PRAWO, oraz z nieterminalnych symboli JEŻELI_ŻYWNOŚĆ_PRZED_TOBĄ(… , …), GRUPA2(…, …), GRUPA3(… , … , …). Rozkazy sterują ruchem mrówki, instrukcja JEŻELI… wykonuje pierwszy zapisany w nawiasie rozkaz, o ile przed mrówką znajduje się żywność, w przeciwnym razie wykona drugi rozkaz. Instrukcje GRUPA2 i GRUPA3 służą do grupowania rozkazów. Po natrafieniu na tę instrukcję mrówka wykona kolejno wszystkie zapisane w nich rozkazy. Po wygenerowaniu drzewa - programu działania mrówki, przeprowadzana jest symulacja. Mrówka wędruje przez planszę wykonując 400 kroków, po drodze zliczając sztuki żywności, przez które przeszła. Funkcją celu jest liczba zebranych sztuk żywności.

Ze względu na dużą złożoność problemów programowania genetycznego, zwykle stosuje się względnie liczne populacje osobników: typowe wartości to 500 do 5000 sztuk.

Bibliografia

Opracowania w języku polskim:

[1] Goldberg D.E., *Algorytmy genetyczne i ich zastosowania*, Wydawnictwa Naukowo-Techniczne, Warszawa 1995.

[2] Cytowski J., *Algorytmy Genetyczne. Podstawy i zastosowania*. Akademicka Oficyna Wydawnicza PLJ, Warszawa 1996.

[3] Gwiazda T. D., *Algorytmy genetyczne. Wstęp do teorii*. TDG s.c., Warszawa 1995.

Opracowania i prace obcojęzyczne:

[4] De Jong K.A., 1993. *Genetic Algorithms Are NOT Function Optimizers*, w: *Foundations of Genetic Algorithms 2*, red. L.D. Whitley, Morgan Kaufmann Publishers.

[5] Goldberg D.E., 1989. *GA in Search, Optimisation, and Machine Learning*. Addison-Wesley.

[6] Holland J.H. 1975. *Adaptation in natural and artificial systems*. The MIT Press, Cambridge, 1992.

[7] Koza J.R., 1992. *Genetic Programming: On the Programming of Computers by Means of the Natural Selection*. The MIT Press.

[8] Michalewicz Z., 1994. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag.

[9] Nix A., Vose M. D., 1991. *Modelling genetic algorithms with markov chains,* w: *Annals of Mathematics and Artificial Intelligence.*

[10] Suzuki J., 1993. *A Markov Chain Analysis on A Genetic Algorithm.* w: *Proceedings of the Fifth International Conference on Genetic Algorithms.* Morgan Kaufmann Publishers.

[11] Wróblewski J., 1995. *Finding minimal reducts using genetic algorithms (extended version).* ICS Research report 16/95, Politechnika Warszawska.

[12] Wróblewski J., 1995. *Genetic approach to scalar quantisation.* w: *Proceedings of the International Workshop on Intelligent Information Systems IV, Augustów 1995.* Instytut Podstaw Informatyki PAN, Warszawa.