Programowanie obiektowe i inżynieria oprogramowania

Algorytm genetyczny

Tomasz Rogala 406488

1. Wstęp:

Projekt algorytmu genetycznego napisany w języku c++ w programie Microsoft Visual Studio 2022. Składa się z 4 klas: genu ⊂ osobnika ⊂ populacji ⊂ algorytmu.

2. Budowa:

Gen:

- określa parametr/gen, który ma określone wartości i zakresy, można je nazywać, modyfikować zakresy oraz wartości.

- podstawowe zmienne:

```
std::string name, opis;
double min,max,step;
int id;
```

min, max – określają zakres wartości
step – jaki krok jest pomiędzy wartością maksymalną a minimalną
name, opis – nazwa oraz ewentualny opis
id – liczba określająca, które ‘miejsce’ zajmuje ta wartość w zakresie

- podstawowe funkcje:

```
void set_range(double min, double max, double step);
void set_val(double val) { this->id = get_val_id(val); }
void set_rand_val();

double get_step() { return this->step; }
double get_val() const { return min + id * step; }
```

- funkcje zmieniające wartości lub zakres genu oraz funkcje zwracające obecne wartości

Osobnik:

- zawiera określoną liczbę genów, funkcje oceny oraz indywidualną ocenę

- podstawowe zmienne:

```
Tparam genotype[GENS_COUNT] =
{
    Tparam ("x1","", 0,100,1),
    Tparam ("x2","", 0, 100,1)
};

double mark;
```

Określone jakie i ile genów ma ten osobnik oraz jego ocena.

- podstawowe funkcje:

```
void Tcandidate::rate()
{
    double ocena = 0;
    ocena = genotype[0].get_val() * 10 + genotype[1].get_val() * 1;
    this->mark = ocena;
}
```

Tutaj znajduje się funkcja szukana, która określa przystosowanie/ocenę danego osobnika.

Populacja:

- Określa liczbę osobników danej populacji

- podstawowe zmienne:

```
static unsigned int count;
static unsigned int pops_count;
unsigned int id;

double best_val;
```

best_val – zapisuje ocenę najlepszego osobnika populacji jako ocenę populacji.
pops_count – ile osobników ma liczyć dana populacja
count – ile łącznie powstało osobników
id – numer populacji

- podstawowe funkcje:

```
void calculate();
Tcandidate get_best_candidate();
```

calculate() – przelicza ocenę osobników ze względu na ich geny
get_best_candidate() – zwraca kandydata o najlepszej ocenie

Algorytm:

- Rozpoczyna algorytm liczący

```
class TAlg
{
    TPop* wsk_pop_prev;
    TPop* wsk_pop_next;
    double best_ogcand_val;
public:
    TAlg(int pop_count, int max_generation_count = 50, int progress_cap = 10);

};
```

- składa się z 2 obiektów populacji, gdzie druga jest tworzona na podstawie pierwszej

- posiada zmienną zapisującą najlepszego osobnika z całego przebiegu algorytmu

- do konstruktora podaje się:

- ile osobników ma być w danej populacji
- ile maksymalnie generacji ma wygenerować
- warunek zatrzymania polegający na braku poprawy przez liczbę generacji

3. Opis działania algorytmu:

Po utworzeniu obiektu algorytmu `TAlg alg1(20);` tworzy się pierwsza populacja o całkowicie losowych osobnikach oraz oblicza ich przystosowanie.

```
this->best_ogcand_val = 0;
wsk_pop_next = new TPop(pop_count);
wsk_pop_prev = wsk_pop_next;
wsk_pop_next->calculate();
```

Zapisuje najlepszego osobnika oraz wyświetla stan pierwszej generacji.

```
double best_cand_val = wsk_pop_next->get_best_candidate().get_mark();
```

```
if (best_ogcand_val < best_cand_val)
{
    progress_check = 0;
    this->best_ogcand_val = best_cand_val;
}

cout << "Best value of candidate in " << j  << " generation: " << best_cand_val << "\n";
```

Następnie kolejna generacja tworzona jest na podstawie poprzedniej generacji. Do utworzenia nowej generacji zastosowałem oraz dodałem do klasy populacją nową funkcję – koło ruletki.

```
int TPop::get_random_waged_cand_id()
{
    int size = this->pops.size();
    double suma = 0;
    double suma_temp = 0;
    for (int i = 0; i < size; i++)
    {
        suma += pops[i].get_mark();
    }
    double los = (double)rand() * suma;

    for (int i = 0; i < size; i++)
    {
        suma_temp += pops[i].get_mark();
        if (los <= suma_temp)
        {
            return i;
        }
    }
    return 0;
}
```

Funkcja oblicza całkowitą wartość wszystkich ocen osobników, dodaje do siebie oraz skaluje do zakresu 0-1. Generowana zostaje losowa zmienna od 0 do 1. Potem znów dodawana jest po kolei ocena każdego osobnika i gdy przekroczy do wylosowaną wartość zwraca tego osobnika. Dzięki temu im większa ocena i przystosowanie osobnika tym większa szansa na jego wybranie. Tym sposobem

genotyp wybranego osobnika przechodzi do następnej populacji, gdzie ponownie losuje się w jaki sposób to nastąpi.

```
wsk_pop_next->pops[0] = wsk_pop_prev->get_best_candidate();
for (int i=1;i<pop_count;i++)
{
    int los = rand() % 100 + 1;

    int cand1_id = wsk_pop_prev->get_random_waged_cand_id();
    int cand2_id = wsk_pop_prev->get_random_waged_cand_id();
    if (los > 80)
    {
        wsk_pop_next->pops[i] = wsk_pop_next->pops[cand1_id];
    }
    else if (los < 40)
    {
        wsk_pop_next->pops[i].crossing(wsk_pop_prev->pops[cand1_id], wsk_pop_prev->pops[cand2_id]);
    }
    else
    {
        wsk_pop_next->pops[i] = wsk_pop_next->pops[cand1_id];
        wsk_pop_next->pops[i].mutating();
    }

}
```

Dla bezpieczeństwa osobnik z najlepszą oceną zawsze przechodzi dalej. Losowanie wygląda następująco – 20% procent szans na przepisanie tego osobnika dalej bez zmian, 40% szans na mutację jego genów oraz 40% na krzyżowanie z innym również wylosowanym osobnikiem.

```
void Tcandidate::crossing(Tcandidate cand1, Tcandidate cand2)
{
    for (int i = 0; i < GENS_COUNT; i++)  // przejscie po kazdym genie rodzicow
    {
        double los = rand() % 100 + 1; // los którego rodzica przejdzie do potomka

        if (los > 50)
        {
            this->genotype[i] = cand1.genotype[i];
        }
        else
        {
            this->genotype[i] = cand2.genotype[i];
        }
    }

}
```

Funkcja krzyżowania przechodzi po liczbie genów rodziców osobnika oraz dodaje gen o danym numerze losowego rodzica. W ten sposób funkcja jest uniwersalna od liczby genów.

```
void Tcandidate::mutating()
{

    int ile_mutacji = rand() % GENS_COUNT; // losowanie liczby ile razy ma zajść mutacja

    for (int i = 0; i < ile_mutacji; i++)
    {
        int wybor_gen = rand() % GENS_COUNT; // losowanie który gen zostanie zmutowany tym razem
        int size_of_mutation = rand() % this->genotype[wybor_gen].get_step_ratio()/10;
        double los = rand() % 100 + 1; // losowanie wartości czy wprzód czy do tyłu

        if (los > 50)
        {
            genotype[wybor_gen].set_val(genotype[wybor_gen].get_val() + size_of_mutation);
        }
        else
        {
            genotype[wybor_gen].set_val(genotype[wybor_gen].get_val() - size_of_mutation);
        }

    }

}
```

Funkcja mutacji losuje ile zmian genów nastąpi, który gen zmodyfikować oraz w którą stronę wykonać krok o wartości 10% zakresu genu.

Dodatkowo algorytm posiada 2 warunki zatrzymania: gdy nie następuje poprawa wyników przez daną liczbę generacji, oraz po wykonaniu limitu obliczeń.

4. Przykładowe działanie algorytmu:

```
int main()
{
    srand(time(NULL));


    TAlg alg1(20);

    return 0;
}
```

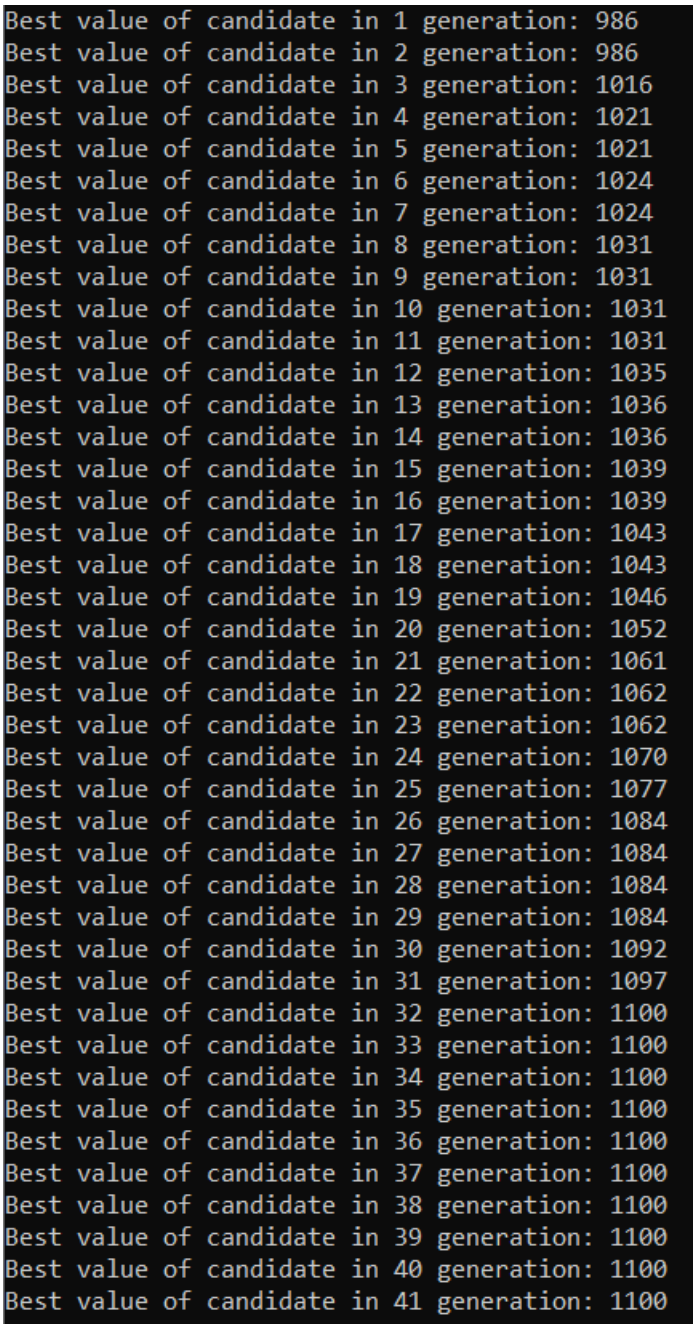
Generowane populacje o 20 osobnikach, 50 maksymalnych generacji oraz zatrzymanie po 10 nieudanych próbach poprawy, gdzie funkcja celu ma postać:

```
ocena = genotype[0].get_val() * 10 + genotype[1].get_val() * 1;
```

A geny osobników:

```
Tparam genotype[GENS_COUNT] =
{
    Tparam ("x1","", 0,100,1),
    Tparam ("x2","", 0, 100,1)
};
```

Z prostego wnioskowania maksymalne wartości genów osiągną maksimum globalne funkcji celu.



Algorytm zadziałał poprawnie i znalazł maksimum.