Projektowanie efektywnych algorytmów

Projekt zadanie 3

Kasper Radom (264023) *Algorytm Genetyczny (ewolucyjny)*

Piątek 11:15

Dr Jarosław Mierzwa

1. Wstęp

Zadanie polegało na implementacji i zbadaniu algorytmu genetycznego. Polega on na symulowaniu procesu ewolucji. Program tworzy wirtualne obiekty zwane chromosomami. W problemie ATSP każdy chromosom jest odpowiednikiem jednego cyklu Hamiltona w grafie więc tym samym jednego z rozwiązań. Wylosowana populacja poddawana jest ocenie aby wyliczyć dla każdego osobnika wartość jego cyklu.

Jakość rozwiązania, a więc i potencjał chromosomu wyznaczony jest przez wagę cyklu zakodowanego w osobniku. Jakość rozwiązania zależna jest od wagi krawędzi, a więc te są kluczowe przy przekazywaniu informacji chromosomom potomnym. Chcąc zoptymalizować jakość przekazywanych informacji, świat nauki pracuje nad znajdowaniem coraz to dokładniejszych algorytmów krzyżowania. W poniższym projekcie zaimplementowany został algorytm PMX (Partially Mapped Crossover). Działa on następująco:

- Dla danej pary chromosomów (0123456, 2310546) losujemy dwa punkty krzyżowania i rozdzielamy chromosomy według wylosowanych indeksów, na przykład (012|345|6, 231|054|6)
- Następnie kopiujemy wylosowane fragmenty do potomków (***|054|*, ***|345|*)
- Następnie pilnując, aby wierzchołki grafy w potomkach nie powtarzały się przepisujemy kolejno z rodzica 1 do potomka 1 i z rodzica 2 do potomka 2. Jeśli zauważamy, że kolejne wierzchołki będą się powtarzać, bierzemy wierzchołek odpowiadający mu w drugim rodzicu. Wierzchołki odpowiadające sobie to: $3 \leftrightarrow 0$, $4 \leftrightarrow 5$. Po wykonaniu tej operacji otrzymujemy parę: (312|056|6, 201|345|6). Są to potomkowie. Kandydaci do nowej populacji.

Pula osobników przeznaczonych do bycia rodzicami wybierana jest metodą turniejową. Polega ona na losowaniu z populacji określonej liczby osobników i wybraniu z nich najlepszego. Proces jest powtarzany ze zwracaniem tak długo, aż lista rodziców osiągnie pożądaną wielkość.

Aby zapobiec utknięciu w minimum lokalnym stosuje się różnego typu mutacje z pewnym prawdopodobieństwem. W zadaniu zaimplementowane zostały dwa rodzaje mutacji:

- Mutacja typu Insert - polega na wylosowaniu wierzchołka i indeksu na który ma być on wstawiony. Przykładowo w liście (0,1,2,3,4,5) dla wylosowanego wierzchołka o indeksie 3 i miejsca docelowego, 1 lista zostanie przetransformowana na (0,3,1,2,4,5)

- Mutacja typu Exchange – posługuje się ona prostą metodą swap. Zamienia dwa wierzchołki o wylosowanych indeksach miejscami. Dla wyżej wymienionej listy, po zastosowaniu tej mutacji przybierze ona formę (0,4,2,3,1,5) dla indeksów 1 oraz 4

Aby zapobiegnąć przerostowi populacji należy omówić kolejni ważny proces. Jest nim sukcesja. Może ona również przybierać różne formy. W zadaniu zaimplementowany został model sukcesji metodą elitarną. Polega ona na wybraniu tych osobników których funkcja przystosowania jest najkorzystniejsza pod względem rozwiązania problemu.

Algorytm wykonuje się puki nie wystąpi warunek zakończenia. W tym wypadku warunkiem przerwania jest upłynięty czas.

2. Opis klas w projekcie

2.1. GeneticAlg

Jest to klasa zajmująca się obsługą całego algorytmu. Zawiera wszystkie zmienne oraz funkcje z nim związane. Przy jej omawianiu należy wspomnieć o kliku najistotniejszych metodach:

- *start* – służy do sterowania całym algorytmem. Inicjalizuje klasę odpowiedzialną za mierzenie czasu, oraz dzieli algorytm na etapy:
- losowanie populacji, metoda *generatePopulation:*

```
vector<Chromosome> GeneticAlg::generatePopulation(int size) {
    vector<Chromosome> population;
    for(int i=0; i<size; i++) {
        Chromosome* chromosome = new Chromosome();
        //wypelniamy chromosom wierzchołkami grafu
        for (int j = 0; j < matrixSize; j++) {
            chromosome->path.push_back(j);
        }
        //dodajemy wierzchołek ostatni aby powstał cykl
        chromosome->path.push_back(0);
        //losowo mieszamy kolejność wierzchołków rozwiązania
        shuffle(chromosome->path.begin() + 1, chromosome->path.end() - 1,
generator);

        //dodajemy rozwiązanie do populacji
        population.push_back(*chromosome);
    }
    return population;
}
```

Wypełnia ona wektor kolejnymi wierzchołkami grafu, a następnie losowo miesza ich kolejność

- metoda *ratePopulation* zajmuje się ustawieniem parametru *cost* w każdym chromosomie. Pozwala to w dalszych etapach skutecznie oceniać przystosowanie danego osobnika

```
void GeneticAlg::ratePopulation() {
    for(Chromosome & chromosome : population){
        chromosome.calcCost(matrix, matrixSize);
    }
}
```

- kolejna metoda implementuje wyżej opisaną sukcesją metodą elitarną

```
vector<Chromosome> GeneticAlg::succession(vector<Chromosome> population)
{//sukcesja elitarna
    vector<Chromosome> newPopulation;
    //sortowanie wektora
    std::sort(population.begin(), population.end(),
              [](const Chromosome& a, const Chromosome& b) {
                  return a.cost < b.cost;
              });

    newPopulation.assign(population.begin(), population.begin() + popula-
tionSize);
    return newPopulation;
}
```

- za przeprowadzenie turnieju odpowiedzialna jest funkcja *doTournament* zwraca ona wektor chromosomów przypisywany następnie do zmiennej *parents* i używany przy krzyżowaniu

```
vector<Chromosome> GeneticAlg::doTournament(vector<Chromosome> population){
    int tournamentSize = int(matrixSize * 0.08);
    if(tournamentSize<2)tournamentSize=2;
    int winnersSize = population.size();
    int chromosomeIndex;

    vector<Chromosome> parents;
    vector<Chromosome> tournmentGroup;
    Chromosome chromosome;

    parents.push_back(bestInIteration);

    //wypełniamy vektor parents nowymi chromosomami
    for(int i=0; i<winnersSize;i++){
        tournmentGroup.clear();
        //bierzemy określoną liczbę chromowomów. losujemy je ze zwracaniem.
wybieramy najlepszych z wylosowanej puli
        for(int i=0; i<tournamentSize; i++) {
            chromosomeIndex = uniform_int_distribution<int>(0, int(popula-
tion.size()) - 1)(generator);
            chromosome = population[chromosomeIndex];
            tournmentGroup.push_back(chromosome);
        }
        chromosome = findBestChromosome(tournmentGroup);
        parents.push_back(chromosome);

    }

    return parents;

}
```

- proces krzyżowania przebiega po kolei dla każdego osobnika w puli potencjalnych rodziców. Każdy ma równe szanse na stworzenie potomka.

```
for(int i=0;i<parents.size()-1;i+=2) {
    double randomValue = uniform_real_distribution<double>(0.0, 1.0)(gener-
ator);
    if(randomValue < crossingPoss){
        pair<Chromosome, Chromosome> children = crossing(parents[i], par-
ents[i+1]);
        population.push_back(children.first);
        population.push_back(children.second);
    }
}
```

Jeśli rodzicowi przypadła szansa na krzyżowanie wykonywana jest na nim metoda *pmxCrossover*:

```
pair<Chromosome, Chromosome> GeneticAlg::pmxCrossover(Chromosome &parent1,
Chromosome &parent2) {
    Chromosome child1;
    Chromosome child2;
    int index1;
    int index2;

    do {
        index1 = uniform_int_distribution<int>(1, parent1.path.size() -
2)(generator);
        index2 = uniform_int_distribution<int>(1, parent1.path.size() -
2)(generator);
    }while(index1 >= index2);

    //przypisywanie fragmentu chromosomu ktory nie zostanie zmieniony
    vector<int> fromFirst;
    for(int i = index1; i<=index2;i++){
        fromFirst.push_back(parent1.path[i]);
    }

    vector<int> fromSecond;
    for(int i = index1; i<=index2;i++){
        fromSecond.push_back(parent2.path[i]);
    }

    int value;
    for(int i=0; i<parent1.path.size()-1; i++){
        //dodawanie sekcji dopasowań do potomków
        if(index2>=i and i>=index1){
            child1.path.push_back(fromSecond[i-index1]);
            child2.path.push_back(fromFirst[i-index1]);
            continue;
        }

        //pierwszy potomek
        value = parent1.path[i];
        //dopuki wartosc jest na sekcji dopasowań szukamy nowej wartości
        while(find(fromSecond.begin(), fromSecond.end(), value) != fromSec-
ond.end()){
            //szukamy indeksu na liscie dopasowań
            auto iterator = find(fromSecond.begin(), fromSec-
ond.end(),value);
            int matchListIndex = distance(fromSecond.begin() , iterator);
            value = fromFirst[matchListIndex];
```

```
        }
        child1.path.push_back(value);

        //drugi potomek
        value = parent2.path[i];
        //dopuki wartosc jest na sekcji dopasowań szukamy nowej wartości
        while(find(fromFirst.begin(), fromFirst.end(), value) != from-
First.end()){
            //szukamy indeksu na liscie dopasowań
            auto iterator = find(fromFirst.begin(), fromFirst.end(),value);
            int matchListIndex = distance(fromFirst.begin() , iterator);
            value = fromSecond[matchListIndex];
        }
        child2.path.push_back(value);
    }

    child1.path.push_back(0);
    child2.path.push_back(0);

    return make_pair(child1, child2);
}
```

- Na końcu pętli *do while* wykonywane jest mutowanie analogicznie do krzyżowania-po kolei dla każdego osobnika sprawdzane jest czy mutacja będzie miała miejsce w jego przypadku

```
for(Chromosome &chromosome:population) {
    double randomValue = uniform_real_distribution<double>(0.0, 1.0)(gener-
ator);
    if(randomValue < mutationPoss){
        mutation(chromosome);
    }
}
```

Mutacja zaimpementowana jest w dwóch rodzajach, metoda *insertMutation:*

```
void GeneticAlg::insertMutation(Chromosome* chromosome) {
    int index1;
    int index2;
    //losowanie punkców do mutacji
    index1 = uniform_int_distribution<int>(1, chromosome->path.size() -
2)(generator);
    do {
        index2 = uniform_int_distribution<int>(1, chromosome->path.size() -
2)(generator);
    }while(index1 == index2);

    //zapamietanie usuwanego wierzchołka
    int vertex = chromosome->path[index1];
    //usuwanie wierzchołka
    chromosome->path.erase(chromosome->path.begin()+index1);
    //dodawanie wierzchołka na wybrany indeks
    chromosome->path.insert(chromosome->path.begin()+index2, vertex);
}
```

Metoda *exchangeMutation*:

```
void GeneticAlg::exchangeMutation(Chromosome* chromosome) {
    int index1;
    int index2;

    index1 = uniform_int_distribution<int>(1, chromosome->path.size() -
2)(generator);
    do {
        index2 = uniform_int_distribution<int>(1, chromosome->path.size() -
2)(generator);
    }while(index1 == index2);

    swap(chromosome->path[index1],chromosome->path[index2]);
}
```

2.2. Chromosome

Klasa reprezentująca chromosomy. Zawiera zmienne *path* przechowujące cykl który przypadł danemu chromosomowi oraz *cost*, która zawiera koszt wyżej wymienionej ścieżki. Klasa posiada również metodą *calcCost* uzupełniającą zmienną *cost* na podstawie tabeli sąsiedztwa podanej jako argument.

```
void Chromosome::calcCost(int **matrix, int matrixSize) {
    cost = 0;
    for(int i = 0; i< matrixSize; i++){
        cost += matrix[path[i]][path[i+1]];
    }
}
```

2.3. Time

Klasa Time zaimplementowana została identycznie jak w poprzednim projekcie. Pozwala ona na zapisanie czasu rozpoczęcia algorytmu i dzięki temu na uzyskanie czasu w dowolnej chwili jego funkcjonowania.

2.4. Test

Klasa tworzona na początku działania programu. Przechowuje podane przez użytkownika parametry, zarządza pamięcią oraz pozwala w łatwy sposób zarządzać eksperymentami na instancjach problemu. Poza metodami odpowiedzialnymi za komunikacje z użytkownikiem, warto dodać kilka komentarzy do reszty funkcji:

- **readMatrix** – funkcja pozwala na łatwe wczytanie grafu z tabeli sąsiedztwa zawartej w pliku
- **startAlgorithm –** rozpoczyna działanie algorytmu o podanych parametrach

3. **Wyniki pomiarów**

Algorytmy zostały uruchomione kolejno na 2min, 4min oraz 6min dla kolejnych rozmiarów problemu

plik	ftv47.atsp					
liczebność populacji	5		10		15	
rodzaj mutacji	exchange	insert	exchange	insert	exchange	insert
koszt	2691	2032	2578	2106	2402	2353
czas [s]	7,507	8,793	79,139	63,397	5,371	30,880

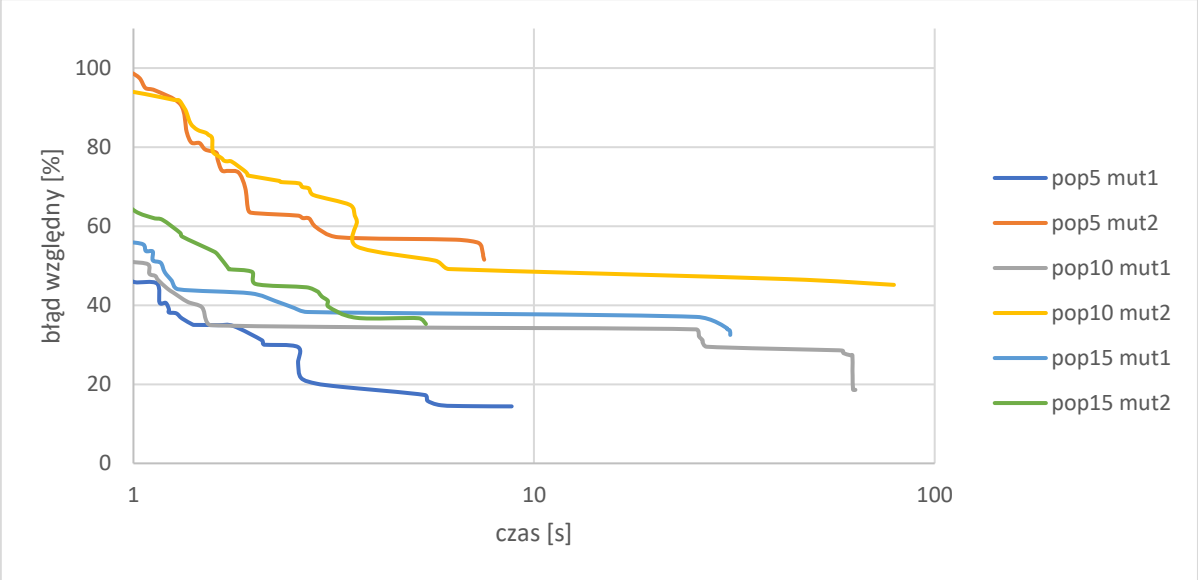
plik	ftv170.atsp					
liczebność populacji	5		10		15	
rodzaj mutacji	exchange	insert	exchange	insert	exchange	insert
koszt	7336	5533	7928	6356	7455	5800
czas [s]	239,093	236,534	236,031	234,511	221,452	223,760

plik	rgb403.atsp					
liczebność populacji	5		10		15	
rodzaj mutacji	exchange	insert	exchange	insert	exchange	insert
koszt	3334	3244	3424	3251	3369	3370
czas [s]	358,399	358,020	359,612	352,078	359,447	353,383

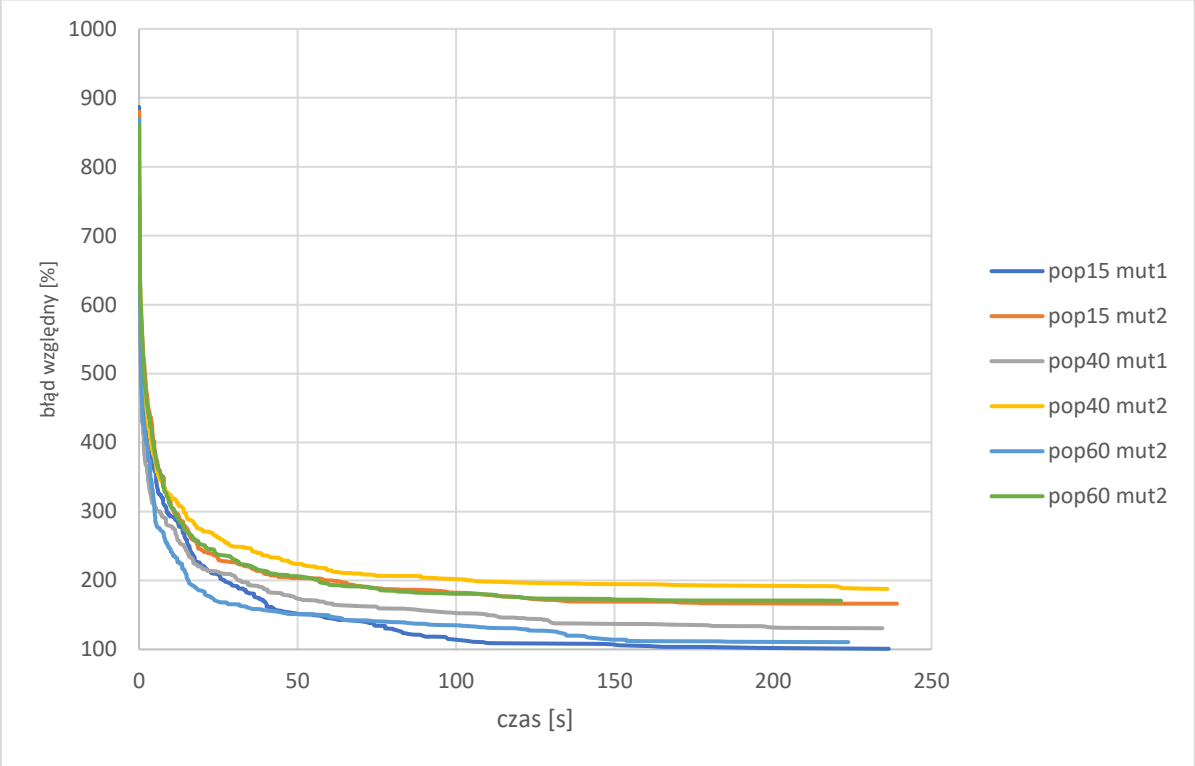
4. **Prezentacja graficzna wyników symulacji**

W celu zaoszczędzenia miejsca na wykresach, wprowadzone zostały sktóty; popX – liczebność populacji – X, mut 1- mutacja sposobem exchange, mut 2- mutacja insert.

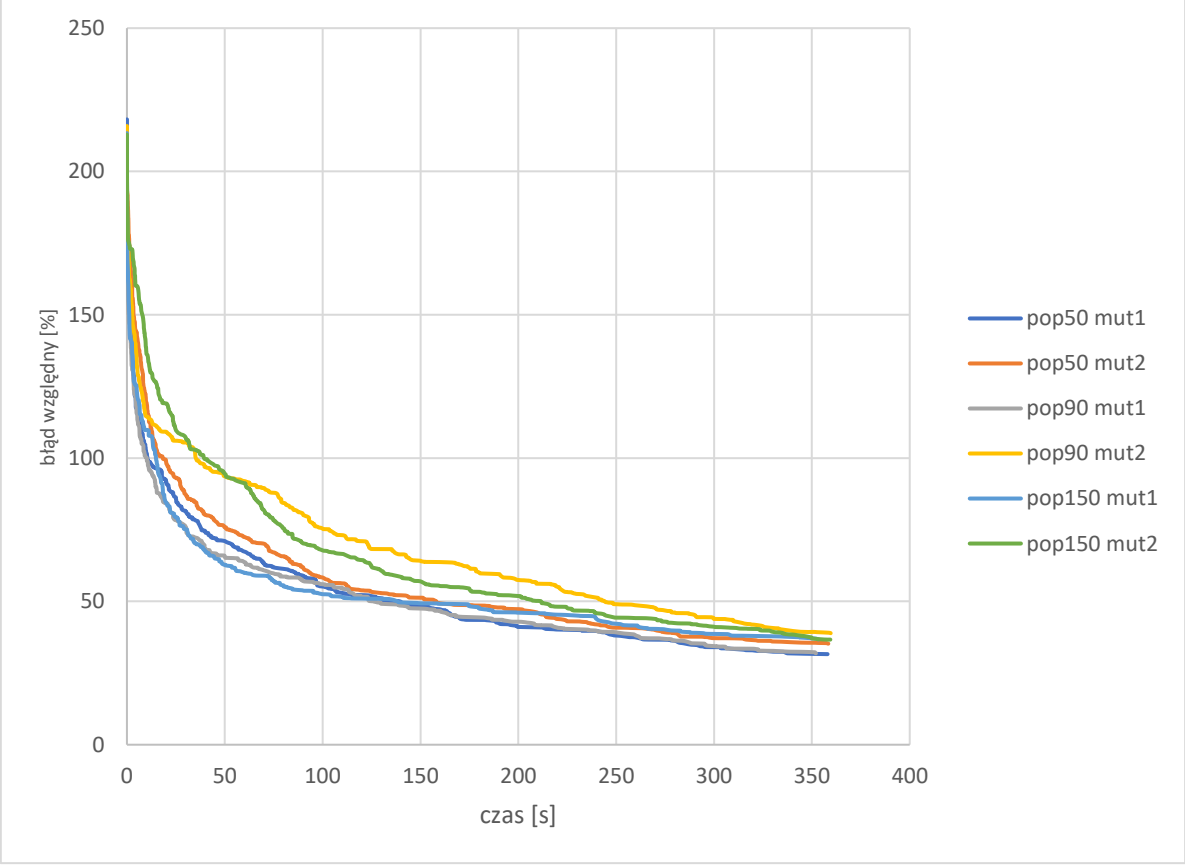
ftv47.atsp



ftv170.atsp



rgb403.atsp



4. **Podsumowanie**
   Projekt utrwalił wiedzę z zakresu algorytmu genetycznego. Pozwolił dogłębnie poznać jego zasadę działania oraz możliwości. Dla większych instancji stosowania tego typu algorytmu wydaje się korzystniejsze. Jeśli natomiast błąd był po stronie implementacji, wynikało to zapewne z nieefektywnie dobranych wielkości populacji do toamiaru problemu.