Politechnika
Wrocławska

Sprawozdanie 4
Algorytm Genetyczny - Problem komiwojażera

Optymalizacja procesów dyskretnych

Aleksander Żołnowski - 272536

9 czerwca 2025

1 Wstęp

Zadanie do wykonania polegało na zaimplementowaniu algorytmu genetycznego, w celu rozwiązania klasycznego problemu komiwojażera (TSP – Travelling Salesman Problem). Problem ten polega na wyznaczeniu najkrótszej możliwej trasy, która odwiedza każde z zadanych miast dokładnie jeden raz i wraca do punktu początkowego. Dodatkowo należało zbadać wpływ parametrów na działanie algorytmu.

Wyniki uzyskane przez algorytm są uzyskane na zbiorze danych `"kroA100.tsp"`.

2 Struktura programu i przygotowanie danych

Pierwszym etapem było przygotowanie struktury programu w celu umozliwienia wczytania danych. W tym celu zaimplementowane zostały struktura **City** gdzie przechowywane były współrzędne miasta oraz klasa **Tsp** gdzie przechowywalismy wektor z miastami oraz metody odpowiedzialne za między innymi wczytywanie danych, generowanie tras, implementację algorytmu genetycznego czy zapisywanie wyników. Wektor z miastami stanowi wewnętrzną reprezentację aktualnej trasy w obiekcie klasy Tsp

```
class Tsp
{
private:
    int size;
    std::vector<City> cities;
    std::mt19937 rng{ std::random_device{}() };

public:
    void readFileContent(const std::string& filePath, int skipLines);
    void nearestNeighborMethod();
    void twoOpt();
    void randomSolution();
    std::string getSolutionOrder();
    float getSolutionDistance();
    void writeSolution();
    //void simulatedAnnealing(float tempStart = 10000.0f, float alpha = 0.976f, int iterations = 400, int innerLoop = 100);   // default
    void simulatedAnnealing(float tempStart = 10000.0f, float alpha = 0.976f, int iterations = 400, int innerLoop = 200);   // kroA100
    //void simulatedAnnealing(float tempStart = 12500.0f, float alpha = 0.978f, int iterations = 500, int innerLoop = 300);   // kroA150
    //void simulatedAnnealing(float tempStart = 18000.0f, float alpha = 0.985f, int iterations = 600, int innerLoop = 400);   // kroA200

    void geneticAlgorithm(int populationSize, int generations, float mutationRate);

private:
    float countDistance(const City& cityA, const City& cityB) const;
    float evaluate(const std::vector<City>& individual);
    void mutate(std::vector<City>& individual, float mutationRate);
    std::vector<City> crossover(const std::vector<City>& parent1, const std::vector<City>& parent2);
};
```

(a) Definicja klasy TSP

```
struct City
{
    int id;
    float x;
    float y;
};
```

(b) Definicja struktury City

Po przygotowaniu struktur danych zaimplementowano metodę odpowiedzialną za wczytywanie miast z pliku do wektora znajdującego się w klasie `Tsp`. Metoda ta pomija nagłówkowe linie pliku, a następnie odczytuje dane w formacie: identyfikator miasta oraz jego współrzędne x i y.

```
1  void Tsp::readFileContent(const std::string& filePath, int skipLines)
2  {
3      ifstream file(filePath);
4
5      if(!file.is_open())
6      {
7          cerr << "Can not open file " + filePath << endl;
8          return;
9      }
10
11     int lineCounter = 0;
12     string line;
13
14     while(lineCounter++ < skipLines && getline(file, line))
15     {
16         if (line == "DIMENSION")
17         {
18             file >> size;
19             cities.reserve(size + 1);
20         }
21
22         continue;
23     }
24
25     while(getline(file, line))
26     {
27         if (line == "EOF")
28         {
```

```
            break;
        }

        istringstream iss(line);
        City city;
        if (!(iss >> city.id >> city.x >> city.y))
        {
            cerr << "Error parsing city from line: " << line << endl;
            continue;
        }

        cities.push_back(city);
    }

    file.close();
}
```

3 Wyznaczanie trasy

W celu poprawnego działania algorytmów rozwiązujących problem komiwojażera, niezbędne było zaimplementowanie podstawowych metod. Kluczowe z nich to obliczanie odległości pomiędzy miastami, wyznaczanie całkowitej długości trasy oraz reprezentacja kolejności odwiedzanych miast.

3.1 Obliczanie odległości - countDistance

Metoda `countDistance` służy do obliczenia euklidesowej odległości pomiędzy dwoma miastami. Dzięki niej możliwe jest obliczenie długości każdej krawędzi trasy na podstawie współrzędnych miast.

```
float Tsp::countDistance(const City& cityA, const City& cityB) const
{
    float x_diff_square = pow(cityB.x -  cityA.x, 2);
    float y_diff_square = pow(cityB.y - cityA.y, 2);

    float distance = sqrt(x_diff_square + y_diff_square);

    return distance;
}
```

3.2 Długość całkowita trasy - getSolutionDistance

Metoda ta sumuje długości wszystkich kolejnych odcinków trasy, łączących kolejne miasta w aktualnej permutacji, a na końcu dodaje również odległość powrotu do miasta początkowego, domykając cykl. Funkcja ta jest kluczowa dla oceny jakości wygenerowanych tras przez poszczególne algorytmy.

```
float Tsp::getSolutionDistance()
{
    float distance = 0;

    for (int i = 0; i < cities.size() - 1; i++)
    {
        distance += countDistance(cities[i], cities[i+1]);
    }

    if(!cities.empty() && cities.front().id != cities.back().id)
    {
        distance += countDistance(cities.back(), cities.front());
    }

    return distance;
}
```

3.3 Kolejność odwiedznia miast - getSolutionOrder

Dla celów prezentacji oraz analizy wyników, zaimplementowano metodę `getSolutionOrder`, która generuje tekstową reprezentację kolejności odwiedzanych miast w formie ciągu znaków.

```
string Tsp::getSolutionOrder()
{
    string order;

    for(const City& city : cities)
    {
        order += to_string(city.id) + "␣";
    }

    if(!cities.empty() && cities.front().id != cities.back().id)
    {
        order += to_string(cities.front().id);
    }

    return order;
}
```

3.4 Wizualizacja trasy

Dodatkowo, program umożliwia zapisanie aktualnej kolejności odwiedzanych miast do pliku tekstowego `solution_order.txt`, którego zawarotść wykorzystywana jest w osobnym programie, który odpowiada za wizualizacje utworzonej trasy.

```
if __name__ == "__main__":
    if(len(sys.argv) < 2):
        print("Invalid␣amount␣of␣arguments")
        exit(1)

    cities_file_name = sys.argv[1]
    cities_skip_lines = sys.argv[2]

    order = load_order()
    cities = load_cities(cities_file_name, cities_skip_lines)

    coords = [cities[i] for i in order]
    x, y = zip(*coords)

    plt.scatter(*zip(*cities.values()), c='blue', s=100, label='Cities')
    plt.plot(x, y, 'r-', linewidth=2, label='Path')

    for city_id, (x_i, y_i) in cities.items():
        plt.text(x_i + 0.2, y_i + 0.2, str(city_id), fontsize=9)

    plt.title("Travelling␣Salesman␣Route")
    plt.legend()
    plt.grid(True)
    plt.axis("equal")
    plt.show()
```

Powyższy fragment prezentuje główną część skryptu napisanego w Pythonie odpowiedzialnego za wizualizacje trasy w formie graficznej.

4 Algorytm genetyczny

Algorytm genetyczny jest metaheurystyczną metodą optymalizacji inspirowaną procesem ewolucji biologicznej. W naszym przypadku został zastosowany do rozwiązania problemu komiwojażera. Reprezentacja rozwiązania to permutacja miast, a jako funkcję przystosowania przyjęto łączną długość trasy.

Proces ewolucyjny składa się z następujących kroków:

1. **Inicjalizacja populacji** - Losowe wygenerowanie początkowej populacji możliwych tras.
2. **Ocena** - Obliczenie długości każdej trasy.
3. **Selekcja** - Wybór najlepszych osobników do krzyżowania (20%).
4. **Krzyżowanie** - Zastosowanie krzyżowania porządkowego (order crossover, OX) do stworzenia potomstwa.

5. **Mutacja** - Zamiana miejscami dwóch losowych miast w trasie z pewnym prawdopodobieństwem.

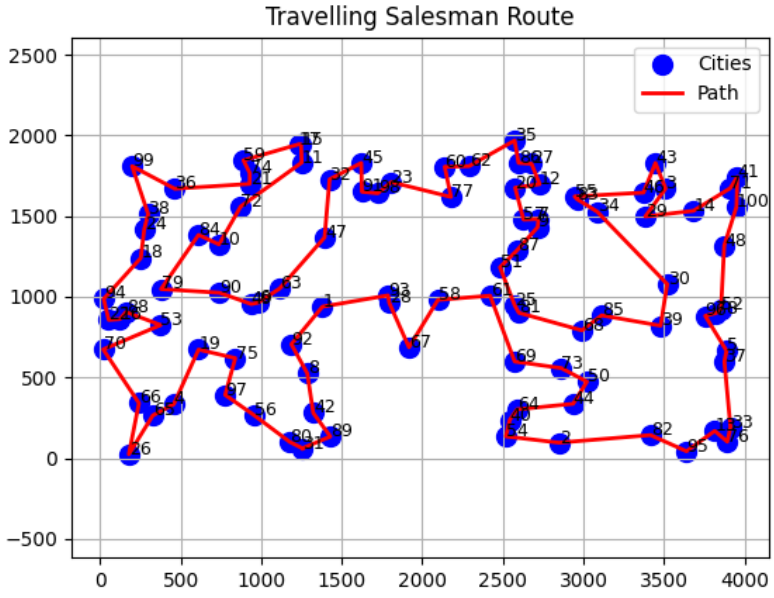
6. **Zastępowanie** - Aktualizacja populacji poprzez zachowanie najlepszych osobników oraz potomstwa.

Algorytm powtarza się przez ustaloną liczbę pokoleń, a najlepszy osobnik z ostatniej populacji jest uznawany za wynik optymalizacji.

```
void Tsp::geneticAlgorithm(int populationSize, int generations, float mutationRate)
{
    std::vector<std::vector<City>> population;

    for (int i = 0; i < populationSize; i++)
    {
        std::vector<City> individual = cities;
        std::shuffle(individual.begin() + 1, individual.end(), rng);
        population.push_back(individual);
    }

    float bestDistance = std::numeric_limits<float>::max();
    std::vector<City> bestSolution;

    for (int g = 0; g < generations; g++)
    {
        std::sort(population.begin(), population.end(), [&](const auto& a, const auto& b)
        {
            return evaluate(a) < evaluate(b);
        });

        if (evaluate(population.front()) < bestDistance)
        {
            bestDistance = evaluate(population.front());
            bestSolution = population.front();
        }

        int eliteCount = std::max(1, (int)(populationSize * 0.2));

        std::vector<std::vector<City>> newPopulation(population.begin(), population.begin() +
            eliteCount);

        while (newPopulation.size() < populationSize)
        {
            int i1 = rand() % eliteCount;
            int i2 = rand() % eliteCount;
            auto child = crossover(population[i1], population[i2]);
            mutate(child, mutationRate);
            newPopulation.push_back(child);
        }

        population = newPopulation;
    }

    cities = bestSolution;
}
```

Działanie powyższego algorytmu zbadano dla przykładowych wartości parametrów:

- Wielkość populacji - 200 osobników
- Pokolenia - 1000
- Prawdopodobieństwo mutacji - 17%



Rysunek 2: Wizualizacja otrzymanej trasy

Genetyczny	
time [ns]	distance
20485551400	22845.6

Rysunek 3: Tabela wyników

4.1 Ocena

Funkcja `evaluate` oblicza całkowitą długość trasy dla danego osobnika (permutacji miast). Przechodzi po kolejnych parach miast, sumując odległości między nimi, a na końcu dodaje odległość między ostatnim a pierwszym miastem, zamykając cykl. W ten sposób uzyskujemy wartość funkcji przystosowania, która jest minimalizowana przez algorytm genetyczny.

```
float Tsp::evaluate(const vector<City>& individual)
{
    float dist = 0.0f;
    for (int i = 0; i < individual.size() - 1; ++i)
        dist += countDistance(individual[i], individual[i + 1]);
    dist += countDistance(individual.back(), individual.front());
    return dist;
}
```

4.2 Mutacja

Mutacja wprowadza losowe zmiany do osobnika w celu zwiększenia różnorodności populacji i uniknięcia zbieżności do lokalnych minimów. W implementacji zastosowano mutację typu *reverse mutation* — z pewnym prawdopodobieństwem wybierany jest losowy fragment trasy (pomiędzy dwoma indeksami) i jego kolejność jest odwracana. Operacja ta nie narusza poprawności permutacji (wszystkie miasta są obecne dokładnie raz).

```
void Tsp::mutate(vector<City>& individual, float mutationRate)
{
    if ((float)rand() / RAND_MAX < mutationRate)
    {
        int i = 1 + rand() % (individual.size() - 2);
        int j = i + 1 + rand() % (individual.size() - i - 1);
        std::reverse(individual.begin() + i, individual.begin() + j);
    }
}
```

4.3 Krzyżowanie

Krzyżowanie służy do tworzenia nowego osobnika na podstawie dwóch rodziców. W tym celu wykorzystano technikę *Order Crossover (OX)*. Najpierw wybierany jest losowy fragment jednego z rodziców, który zostaje bezpośrednio skopiowany do potomka. Następnie, zachowując kolejność, uzupełnia się pozostałe miasta z drugiego rodzica, pomijając te, które już występują w potomku. Pozwala to na zachowanie cech strukturalnych obu rodziców, jednocześnie gwarantując poprawną permutację.

```
vector<City> Tsp::crossover(const vector<City>& parent1, const vector<City>& parent2)
{
    int size = parent1.size();
    int start = rand() % size;
    int end = start + rand() % (size - start);

    vector<City> child(size, City{-1, -1, -1});
    unordered_set<int> used;

    for (int i = start; i <= end; ++i)
    {
        child[i] = parent1[i];
        used.insert(parent1[i].id);
    }

    int idx = (end + 1) % size;
    for (int i = 0; i < size; ++i)
    {
        const City& c = parent2[(end + 1 + i) % size];
        if (used.count(c.id) == 0)
        {
            child[idx] = c;
            idx = (idx + 1) % size;
        }
    }

    return child;
}
```

5 Wpływ parametrów

Aby dobrać optymalne parametry działania algorytmu genetycznego, przeprowadzono serię testów z różnymi wartościami:

- liczby osobników w populacji (rozmiar populacji),
- liczba pokoleń
- współczynnik mutacji

Dla każdego zestawu parametrów zapisano wynik końcowy (długość znalezionej trasy) oraz czas wykonania. Celem było znalezienie wartości dających dobrą jakość rozwiązań przy akceptowalnym czasie działania.

5.1 Rozmiar populacji

Na początku zbadaliśmy wpływ rozmiaru populacji, zachowując dla wszystkich takie same wartości liczby pokoleń(1000) i współczynnika mutacji(0.17)

roz. populacji	czas[ns]	trasa
10	404662900	56618.4
50	3511533600	27423.9
100	7871781200	24504.3
200	20485551400	22845.6
400	42494403400	24267.2

Rysunek 4: Tabela wyników wpływu rozmiaru populacji

Z otrzymanych wyników można zaobserwować znaczącą poprawę działania algorytmu wraz ze wzrostem liczby osobników w populacji. Jednak po przekroczeniu około 200 jednostek dalszy wzrost nie przekłada się już na poprawę jakości rozwiązań, natomiast czas wykonania rośnie znacząco. Optymalna wartość znajduje się w okolicach 200 osobników.

5.2 Liczba pokoleń

Kolejnym krokiem było zbadanie wpływu liczby pokoleń na wynik działania algorytmu. Podczas badań zachowaliśmy stałe wartości dla rozmiaru populacji (200) oraz współczynnika mutacji (0.17).

l. pokoleń	czas[ns]	trasa
10	412036500	119407.0
100	2393027200	50413.5
400	7975124400	27240.3
1000	20485551400	22845.6
2000	36122206500	23957.2

Rysunek 5: Tabela wyników wpływu liczby pokoleń

Na podstawie otrzymanych wyników widać, że liczba pokoleń ma istotny wpływ na jakość rozwiązań. Dla małej liczby iteracji algorytm nie ma możliwości wystarczającego dopasowania, przez co wyniki są słabe. Wraz ze wzrostem liczby pokoleń obserwujemy wyraźną poprawę, jednak po przekroczeniu około 1000 iteracji wzrost jakości rozwiązań wyhamowuje, a czas działania rośnie. Co więcej, pojawia się ryzyko przeuczenia i zbieżności do lokalnych minimów.

5.3 Współczynnik mutacji

Ostatnim etapem było zbadanie jak duże znaczenie ma wielkość współczynnika mutacji na działanie algorytmu. Dla każego z pomiarów wartości rozmiaru popluacji (200) i liczby pokoleń (1000) pozostały stałe.

wsp. mutacji	czas[ns]	trasa
0.05	17164316600	26066.4
0.17	20485551400	22845.6
0.3	19927547200	22719.6
0.6	22496403700	23152.5
0.9	24342229700	24423.9

Rysunek 6: Tabla wyników wpływu wsp. mutacji

Z uzyskanych danych wynika, że wpływ współczynnika mutacji na jakość rozwiązań jest mniejszy niż w przypadku pozostałych parametrów. Zbyt niski współczynnik może prowadzić do zastoju w procesie ewolucyjnym i zbieżności do lokalnych minimów, natomiast zbyt wysoki może powodować zbytnią losowość i destabilizację populacji.

6 Podsumowanie

W ramach projektu zaimplementowano algorytm genetyczny służący do rozwiązywania problemu komiwojażera. Przeprowadzono eksperymenty z parametrami algorytmu, co pozwoliło zaobserwować ich wpływ na jakość i stabilność wyników. Algorytm pozwolił uzyskać dobre jakościowo rozwiązania w rozsądnym czasie obliczeniowym, potwierdzając skuteczność metod ewolucyjnych w rozwiązywaniu problemów optymalizacji kombinatorycznej.