Politechnika
Wrocławska

Sprawozdanie 1
Problem szeregowania zadań na 1 maszynie

Optymalizacja procesów dyskretnych

Aleksander Żołnowski - 272536

3 kwietnia 2025

1 Wstęp

Zadanie do wykonania dotyczyło rozwiązania problemu szeregowania zadań na jednej maszynie. Należało znaleźć minimalny czas zakończenia wszystkich zadań na maszynie C_{max} przy uwzględnieniu parametrów każdego z zadań w modelu RPQ, gdzie

- R - czas po którym zadanie jest gotowe
- P - czas produkcji
- Q - czas stygnięcia

W ramach zajęć zaimplementowano i zbadano różne metody rozwiązania tego problemu. Badania były prowadzone dla 4 zbiorów danych `data1.txt`, `data2.txt`, `data3.txt` i `data4.txt`

2 Przygotowanie danych

Pierwszym etapem było wczytanie do pamięci programu danych z plików tekstowych, gdzie zawarte są informacje o etapach. Plik zawiera liczbe zadań oraz wartości parametrów każdego z zadań. Dane są wczytywane do property klasy `Scheduler` jaką jest wektor struktur `Task`, przechowujący informacje o id, R, P i Q każdego z zadań.

```
class Scheduler
{
private:
    std::vector<Task> tasks;

public:
    void readFileContent(std::string dataFile);
    void quickSort(int low, int high);
    int calculateC_max();
    const std::vector<Task>& getTasks() const;
    void sortRQ(int maxIterations, int tabuSize);
    void schrage();
    void display_order();

private:
    int partition(int low, int high);
};
```

(a) Definicja klasy Scheduler

```
struct Task
{
    int id;
    int R;
    int P;
    int Q;
};
```

(b) Definicja struktury Task

Dane z plików były wczytywane przy użyciu metody `readFileContent`, gdzie należało przekazać nazwe pliku, który ma być wczytany.

```
1  void Scheduler::readFileContent(string dataFile)
2  {
3      ifstream file(dataFile);
4
5      if (!file.is_open())
6      {
7          cerr << "Error: Can not open the '" + dataFile + "' file" << endl;
8          return;
9      }
10
11     int numberOfTasks = 0;
12     file >> numberOfTasks;
13
14     tasks.reserve(numberOfTasks);
15
16     for (int i = 0; i < numberOfTasks; i++)
17     {
18         Task task;
19         task.id = i + 1;
20         file >> task.R >> task.P >> task.Q;
21         tasks.push_back(task);
22     }
23
24     file.close();
25 }
```

3 Obliczenie C_{max}

Pierwszym sposobem było obliczenie C_{max} dla każdego zbioru danych w kolejności w jakiej zostały wczytane z plików. W tym celu wykorzystano podstawowa funkcje do obliczania C_{max}.

```
int Scheduler::calculateC_max()
{
    int time = 0;
    int C_max = 0;

    for (const Task& task : tasks)
    {
        if (time < task.R)
        {
            time = task.R;
        }
        time += task.P;
        C_max = max(C_max, time + task.Q);
    }

    return C_max;
}
```

w ten sposób otrzymano następujące wyniki

	Data:1	Data:2	Data:3	Data:4	Suma
123	25994	33465	57403	51444	168306

Rysunek 2: Wyniki dla nieposortowanych danych

4 Sortowanie po R

Kolejnym krokie było zaimplemenotwanie algorytmu sortującego zadania po czasie gotowości R. W tym celu zastosowano algorytm `QuickSort`

```
int Scheduler::partition(int low, int high)
{
    int pivot = tasks[high].R;

    int i = low - 1;

    for (int j = low; j <= high - 1; j++)
    {
        if(tasks[j].R <= pivot)
        {
            i++;
            swap(tasks[i], tasks[j]);
        }
    }

    swap(tasks[i+1], tasks[high]);

    return i + 1;
}

void Scheduler::quickSort(int low, int high)
{
    if(low < high)
    {
        int pi = partition(low, high);

        quickSort(low, pi - 1);
        quickSort(pi + 1, high);
    }
}
```

Dzięki zastosowaniu sortowania otrzymaliśmy poniższe wyniki, które się znacznie polepszyły

	Data:1	Data:2	Data:3	Data:4	Suma
SortR	14239	33465	40042	39616	127362

Rysunek 3: Wyniki po sortowaniu wedlug R

5 Sortowanie RQ

Po wykonaniu sortowania po czasie gotowości zadań, algorytm został rozbudowany o dodatkowa optymalizacje polegająca na przeszukiwaniu i zamianie zadań w celu znalezienia lepszego ułożenia, które da lepszy czas. W tym celu wykorzystano metode przeszukiwania lokalnego implementując Tabu Search, czyli huerystyczny algorytm polegający na wielokrotnym testowaniu różnych permutacji zadań w celu wybrania najlepszej kolejności.

```
void Scheduler::sortRQ(int maxIterations, int tabuSize)
{
    quickSort(0, tasks.size() - 1);

    vector<Task> bestSolution = tasks;
    int bestC_max = calculateC_max();
    unordered_set<string> tabuList;

    for (int iter = 0; iter < maxIterations; iter++)
    {
        vector<Task> bestNeighbor = tasks;
        int bestNeighborC_max = bestC_max;
        int swapIdx1 = -1, swapIdx2 = -1;

        for (size_t i = 0; i < tasks.size() - 1; i++)
        {
            for (size_t j = i + 1; j < tasks.size(); j++)
            {
                swap(tasks[i], tasks[j]);
                int newC_max = calculateC_max();
                string move = to_string(i) + "," + to_string(j);
                if (newC_max < bestNeighborC_max && tabuList.find(move) == tabuList.end())
                {
                    bestNeighborC_max = newC_max;
                    bestNeighbor = tasks;
                    swapIdx1 = i;
                    swapIdx2 = j;
                }
                swap(tasks[i], tasks[j]);
            }
        }

        if (swapIdx1 != -1)
        {
            tabuList.insert(to_string(swapIdx1) + "," + to_string(swapIdx2));
            if (tabuList.size() > tabuSize) tabuList.erase(tabuList.begin());
            tasks = bestNeighbor;
            bestC_max = bestNeighborC_max;
        }
    }
}
```

W ten sposób otrzymaliśmy jeszcze lepsze wyniki niż dla samego sortowania po R:

	Data:1	Data:2	Data:3	Data:4	Suma
SortRQ	13966	20918	33683	33878	102445

Rysunek 4: Wyniki dla sortowania RQ

6 Algorytm Schrage

Ostatnim sposobem, który zaimplementowaliśmy i zbadaliśmy był algorytm Schrage'a. Służy on do szeregowania zadań polegając na dynamicznym dodawaniu dostępnych zadań do zbioru gotowych zadań i wybieraniu z nich tego, które ma największą wartość . Algorytm ten pozwala skutecznie minimalizować czas zakończenia wszystkich zadań, szczególnie w przypadku ograniczeń wynikających z różnych wartości .

```
void Scheduler::schrage()
{
    quickSort(0, tasks.size() - 1);

    vector<Task> N = tasks;
    vector<Task> G;
    vector<Task> order;

    int t = 0;
    int C_max = 0;

    while(!N.empty() || !G.empty())
    {
        while(!N.empty() && N.front().R <= t)
        {
            G.push_back(N.front());
            N.erase(N.begin());
        }

        if(!G.empty())
        {
            vector<Task>::iterator maxQ_it = max_element(G.begin(), G.end(), compareQ);

            Task e = *maxQ_it;
            G.erase(maxQ_it);

            order.push_back(e);
            t += e.P;
            C_max = max(C_max, t + e.Q);
        }
        else
        {
            t = N.front().R;
        }
    }

    tasks = order;
}
```

Sposób ten pozwolił osiągnać nam najlepszy czas ze wszystkich zaimplementowanych metod

	Data:1	Data:2	Data:3	Data:4	Suma
Schrage	13981	21529	31683	34444	101637

Rysunek 5: Wyniki algorytmu Schrage

7 Podsumowanie

Przebadano i zaimplementowano różne rozwiązania problemu szerogowania zadań na jednej maszynie, mającego na celu minimalizację czasu zakończenia wszystkich zadań C_{max}. Badania zostały przeprowadzone na 4 zbiorach danych, na których testowane były różne algorytmy i podejścia. Najlepsze wyniki uzyskano wykorzystując algorytm Schrage.

	123	SortR	SortRQ	Schrage
Data:1	25994	14239	13966	13981
Data:2	33465	33465	20918	21529
Data:3	57403	40042	33683	31683
Data:4	51444	39616	33878	34444
Suma:	168306	127362	102445	101637

Rysunek 6: Podsumowanie wyników

Politechnika
Wrocławska

Sprawozdanie 2
Problem WiTi

Optymalizacja procesów dyskretnych

Aleksander Żołnowski - 272536

16 kwietnia 2025

1 Wstęp

W ramach zadania należało rozwiązać problem WiTi, polegający na optymalnym szeregowaniu zadań na jednej maszynie w celu minimalizacji całkowitej kary za opóźnienia. Dla każdego zadania podano trzy parametry: `P`, `W` oraz `D`, gdzie:

- P - czas produkcji
- W - waga zadania, odzwierciedlająca koszt jego opóźnienia
- D - termin dostarczenia, po którym naliczana jest kara

Celem było znalezienie takiego uporządkowania zadań, które minimalizuje łączną karę wynikającą z przekroczenia terminów. W ramach zajęć problem został rozwiązany poprzez implementację algorytmu PD.

2 Przygotowanie danych

Pierwszym etapem było zaimplementowanie funkcji odpowiedzialnej za wczytywanie danych z plików tekstowych. Pliki te zawierają liczbę zadań oraz wartości parametrów każdego z nich: czas wykonania (P), wagę (W) i termin dostarczenia (D).
Dane te są wczytywane do właściwości klasy **Scheduler**, którą stanowi wektor struktur **Task**. Każda struktura `Task` przechowuje informacje o pojedynczym zadaniu, czyli id, czasie wykonania (P), wadze (W) oraz terminie (D).

```
class Scheduler
{
private:
    std::vector<Task> tasks;

public:
    void readFileContent(std::string filePath);
    int delayCost();
    void displayOrder();
    const std::vector<Task>& getTasks() const;
    void witiDP();
};
```

(a) Definicja klasy Scheduler

```
struct Task
{
    int id;
    int P;
    int W;
    int D;
};
```

(b) Definicja struktury Task

Dane z plików były wczytywane przy użyciu metody `readFileContent`, gdzie należało przekazać ścieżkę do pliku, który ma być wczytany.

```
void Scheduler::readFileContent(string filePath)
{
    ifstream file(filePath);

    if(!file.is_open())
    {
        cerr << "Can not open file " + filePath << endl;
        return;
    }

    int size = 0;
    file >> size;

    tasks.reserve(size);

    for (int i = 0; i < size; i++)
    {
        Task task;
        task.id = i + 1;
        file >>  task.P >> task.W >> task.D;

        tasks.push_back(task);
    }

    file.close();
}
```

3 Obliczenie kary za opóźnienia

Po wczytaniu danych do pamięci programu, przystąpiono do obliczenia kar za opóźnienia dla każdego zestawu danych, w kolejności, w jakiej zadania zostały wczytane z plików. Do tego celu zaimplementowano funkcję `delayCost()`, której zadaniem jest obliczenie całkowitego kosztu opóźnień na podstawie aktualnej kolejności zadań.

```
int Scheduler::delayCost()
{
    int currentTime = 0;
    int coast = 0;

    for(const Task& task : tasks)
    {
        currentTime += task.P;

        if(currentTime > task.D)
        {
            coast += (currentTime - task.D) * task.W;
        }
    }

    return coast;
}
```

w ten sposób otrzymano następujące wyniki

Data	10	11	12	13	14	15	16	17	18	19	20	SUM
time	3994	6323	7778	7322	6851	6388	5925	5468	6189	5656	13252	75146

Rysunek 2: Wyniki dla nieposortowanych danych

4 Algorytm DP

Kolejnym etapem było zaimplemetowanie algorytmu DP, którego celem jest znalezienia optymalnego ułożenia zadań minimalizującego całkowity koszt opóźnienia. Algorytm DP ,czyli programowania dynamicznego wykorzystuje reprezentację podzbiorów zadań za pomocą masek bitowych, aby znaleźć permutację zadań minimalizującą całkowitą karę za opóźnienia. Dla każdej możliwej kombinacji zadań obliczany jest minimalny koszt wykonania, a następnie na podstawie wartości pomocniczych odtwarzana jest optymalna kolejność. Podejście to zapewnia dokładne rozwiązanie dla małych instancji problemu.

```
void Scheduler::witiDP()
{
    int n = tasks.size();
    int size = 1 << n;
    vector<int> dp(size, INT_MAX);
    vector<int> prev(size, -1);
    vector<int> lastTask(size, -1);

    dp[0] = 0;

    for (int mask = 0; mask < size; ++mask)
    {
        int time = 0;
        for (int i = 0; i < n; ++i)
        {
            if (mask & (1 << i))
            {
                time += tasks[i].P;
            }
        }

        for (int j = 0; j < n; ++j)
        {
            if (!(mask & (1 << j)))
            {
                int nextMask = mask | (1 << j);
                int completionTime = time + tasks[j].P;
                int tardiness = max(0, completionTime - tasks[j].D);
                int cost = dp[mask] + tardiness * tasks[j].W;

```

```
                if (cost < dp[nextMask])
                {
                    dp[nextMask] = cost;
                    prev[nextMask] = mask;
                    lastTask[nextMask] = j;
                }
            }
        }
    }

    vector<Task> optimalOrder;
    int mask = size - 1;
    while (mask)
    {
        int taskIndex = lastTask[mask];
        optimalOrder.push_back(tasks[taskIndex]);
        mask = prev[mask];
    }

    reverse(optimalOrder.begin(), optimalOrder.end());

    tasks = optimalOrder;
}
```

Dzięki zastosowaniu algorytmu DP otrzymaliśmy następujące wyniki dla naszych zbiorów danych:

Data	10	11	12	13	14	15	16	17	18	19	20	SUM
time	766	799	742	688	497	440	423	417	405	393	897	6467

Rysunek 3: Wyniki po sortowaniu wedlug R

5 Podsumowanie

W ramach zadania zaimplementowano algorytm programowania dynamicznego (DP) w celu rozwiązania problemu WiTi. Otrzymane wyniki jednoznacznie pokazują, że zastosowanie algorytmu DP znacząco poprawia jakość rozwiązania w porównaniu do bazowego ułożenia zadań. W każdym przypadku całkowity koszt opóźnień został zredukowany, często nawet kilkukrotnie. Metoda DP okazała się niezwykle skuteczna dla mniejszych instancji problemu, oferując dokładne i zoptymalizowane wyniki.

Data	10	11	12	13	14	15	16	17	18	19	20	SUM
123	3994	6323	7778	7322	6851	6388	5925	5468	6189	5656	13252	75146
PD	766	799	742	688	497	440	423	417	405	393	897	6467

Rysunek 4: Podsumowanie wyników

Politechnika
Wrocławska

Sprawozdanie 3
TSP - Problem komiwojażera

Optymalizacja procesów dyskretnych

Aleksander Żołnowski - 272536

22 maja 2025

1 Wstęp

Zadanie do wykonania polegało na zaimplementowaniu i porównaniu różnych metod rozwiązywania 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.

W tym celu został stworzony program, który implementuje różne sposoby rozwiązania tego problemu takie jak wyznaczanie trasy metodą najbliższego sąsiada wraz zoptymalizowaniem jej za pomocą algorytmu 2-opt, generowanie rozwiązania losowego czy heurystyczną metode symulowanego wyżarzania. Dodatkowo została zaimplementowana wizualizacja graficzna trasy odwiedzanych miast.

Wyniki uzyskane przez algorytmy 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, opytmalizacje czy zapisywanie wyników. Wektor ten stanowi wewnętrzną reprezentację aktualnej trasy w obiekcie klasy Tsp. Wczytane dane mogą być następnie przetwarzane i modyfikowane przez różne algorytmy rozwiązujące problem komiwojażera.

```
class Tsp
{
private:
    int size;
    std::vector<City> cities;

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 = 200);

private:
    float countDistance(const City& cityA, const City& cityB) const;
};
```

(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 - getSolutionOrder

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 - getSolutionDistance

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.

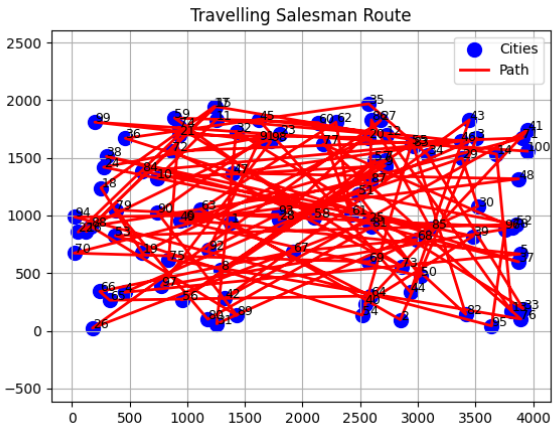
```
if __name__ == "__main__":
    if(len(sys.argv) < 2):
        print("Invalid amount of arguments")
        exit(1)

    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 Domyślna kolejność

Pierwszym przetestowanym sposobem była analiza trasy w kolejności wynikającej bezpośrednio z danych w pliku wejściowym. Po wczytaniu współrzędnych miast, nie została wykonana żadna dodatkowa optymalizacja ani losowe przetasowanie — miasta były odwiedzane w kolejności ich identyfikatorów (np. 1, 2, 3, ..., n).

Taka kolejność odpowiada niejako „naiwnemu” rozwiązaniu, które nie uwzględnia żadnej heurystyki ani analizy przestrzennej położenia miast. W większości przypadków prowadzi to do bardzo nieefektywnej i długiej trasy, która może zawierać liczne przecinające się odcinki oraz zbędne powroty.



Rysunek 2: Wizualiacja trasy - 123

123	
time [ns]	distance
100	188750

Rysunek 3: Tabela wynikow - 123

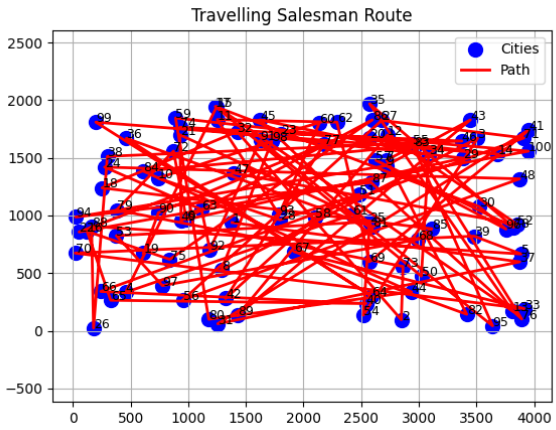
Na rysunku powyżej przedstawiono wizualizację tej domyślnej trasy, a poniżej niej w tabeli zaprezentowano czas działania (pomijalny, ponieważ nie zachodzi żadne przetwarzanie), a także uzyskaną długość trasy.

5 Generowanie rozwiązania losowego

Jednym z najprostszych podejść do rozwiązania problemu komiwojażera jest wygenerowanie losowej trasy odwiedzania miast. W implementacji odbywa się to poprzez przetasowanie wektora miast z pominięciem pierwszego elementu (aby zapewnić, że trasa rozpoczyna się i kończy w tym samym mieście(numer 1). Dzięki temu otrzymujemy pełną, zamkniętą trasę, która przechodzi przez wszystkie miasta dokładnie raz, jednak bez żadnej optymalizacji.

```
1 void Tsp::randomSolution()
2 {
3     auto rng = std::default_random_engine {};
4     std::shuffle(cities.begin() + 1, cities.end(), rng);
5 }
```

Na rysunku poniżej przedstawiono przykładową trasę wygenerowaną losowo. Widoczna jest jej chaotyczna struktura oraz liczne skrzyżowania odcinków. Poniżej niej zaprezentowano uzyskane wyniki — w tym całkowity czas działania oraz długość trasy.



Rysunek 4: Wizualiacja trasy - losowej

Losowe	
time [ns]	distance
5100	178160

Rysunek 5: Tabela wynikow - losowe

Metoda ta nie daje zwykle zadowalających wyników, szczególnie dla dużej ilośći miast gdzie prawdopodobieństwo wylosowania dobrej trasy jest bardzo niskie. Wynika to z tego, że metoda nie bierze pod uwagę żadnych zależności przestrzennych pomiędzy punktami. Losowa kolejność może skutkować trasą znacznie dłuższą niż inne bardziej zaawansowane metody, a w tym przypadku wylosowana trasa jest dłuższa niż domyślna wczytana z pliku.

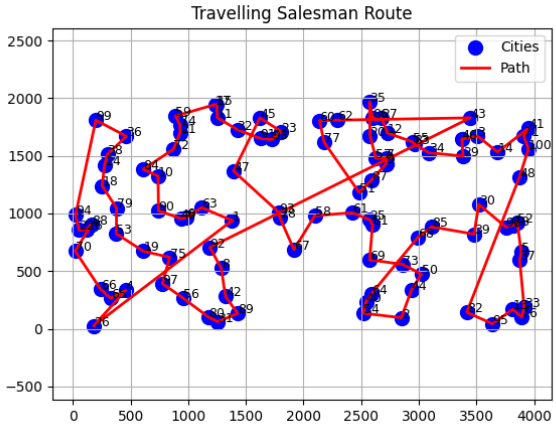
6 Algorytm najbliższego sąsiada

Jedną z klasycznych heurystyk stosowanych do rozwiązywania problemu komiwojażera jest algorytm najbliższego sąsiada. W prezentowanej implementacji metoda ta polega na rozpoczęciu trasy od pierwszego miasta, a następnie wybieraniu w każdej iteracji miasta znajdującego się najbliżej bieżącego. Wybrane miasto dodawane jest do trasy, a następnie usuwane z listy pozostałych do odwiedzenia. Proces powtarzany jest aż do odwiedzenia wszystkich miast, po czym trasa zamykana jest powrotem do punktu początkowego

```
1  void Tsp::nearestNeighborMethod()
2  {
3      City currentCity = cities.front();
4
5      vector<City> remainingCites = cities;
6      remainingCites.erase(remainingCites.begin());
7
8      vector<City> route;
9      route.push_back(currentCity);
10
11     while(!remainingCites.empty())
12     {
13         auto nearestCity = min_element(remainingCites.begin(), remainingCites.end(), [&](const
               City& a, const City& b)
14             {
15                 return countDistance(currentCity, a) < countDistance(currentCity, b);
16             });
17
18
19         currentCity = *nearestCity;
20         route.push_back(currentCity);
21
22         remainingCites.erase(nearestCity);
23     }
24
25     route.push_back(cities.front());
```

```
    cities = route;
}
```

Wyniki są znacznie lepsze niż te generowane w sposób losowy co możemy zaobserwować na poniższym rysunku z wizualizacją oraz zaobserwować bardzo duża róznice w długości trasy w tabeli wyników.



Rysunek 6: Wizualiacja trasy - sąsiad

Sąsiad	
time [ns]	distance
670400	26856.4

Rysunek 7: Tabela wynikow - sąsiad

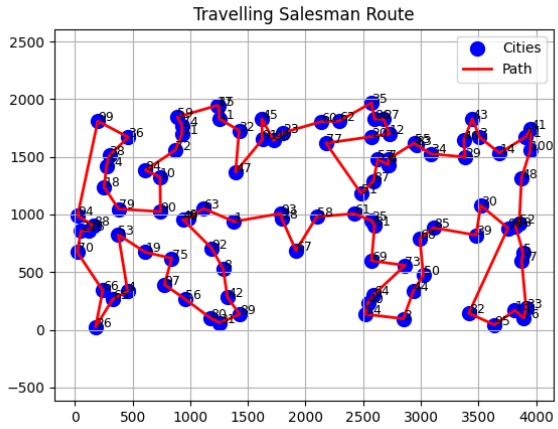
Choć algorytm działa szybko i jest prosty w implementacji, nie gwarantuje znalezienia optymalnej trasy. Wiele zależy od punktu startowego — w niektórych przypadkach może prowadzić do nieefektywnych rozwiązań. Nie rozwiązuje on problemu z krzyżowaniem tras

Aby poprawić jakość rozwiązania, na trasę wygenerowaną metodą najbliższego sąsiada zastosowano algorytm optymalizacji lokalnej 2-opt. Algorytm ten polega na przeszukiwaniu sąsiedztwa obecnej trasy i zamienianiu miejscami wybranych fragmentów, jeśli prowadzi to do skrócenia całkowitej długości trasy. Operacja taka jest powtarzana aż do momentu, w którym dalsze zamiany nie przynoszą poprawy.

```
void Tsp::twoOpt()
{
    bool improvement = true;
    int sizeCities = cities.size();

    while (improvement)
    {
        improvement = false;
        for (int i = 1; i < sizeCities - 2; ++i)
        {
            for (int k = i + 1; k < sizeCities - 1; ++k)
            {
                float delta =
                    countDistance(cities[i - 1], cities[k]) +
                    countDistance(cities[i], cities[k + 1]) -
                    countDistance(cities[i - 1], cities[i]) -
                    countDistance(cities[k], cities[k + 1]);

                if (delta < -1e-6)
                {
                    reverse(cities.begin() + i, cities.begin() + k + 1);
                    improvement = true;
                }
            }
```

```
        }
    }
}
```

Na rysunku poniżej przedstawiono poprawioną trasę po zastosowaniu algorytmu 2-opt, a w tabeli poniżej widoczne są wyniki uzyskane po optymalizacji.



Rysunek 8: Wizualiacja trasy - sąsiad+2opt

Sąsiad + 2-opt	
time [ns]	distance
4043900	22955.9

Rysunek 9: Tabela wynikow - sąsiad+2opt

Można zaobserwować zauważalnie krótsza długość trasy oraz wydłużony czas wykonania, wynikający z dodatkowych obliczeń. Dzięki dodatkowej optymalizacji został rozwiązany problem krzyżowania się tras oraz trasa staje się zdecydowanie krótsza.

7 Symulowane wyżarzanie

Kolejnym podejściem do rozwiązania problemu komiwojażera było zastosowanie metaheurystyki inspirowanej procesami fizycznymi – algorytmu symulowanego wyżarzania (Simulated Annealing). Algorytm ten imituje proces powolnego chłodzenia materiału, podczas którego cząsteczki mają początkowo dużą swobodę ruchu, pozwalającą na eksplorację przestrzeni rozwiązań, a następnie stopniowo „zamrażają się" w stanie minimalnej energii (lokalnym minimum).

7.1 Domyślne parametry

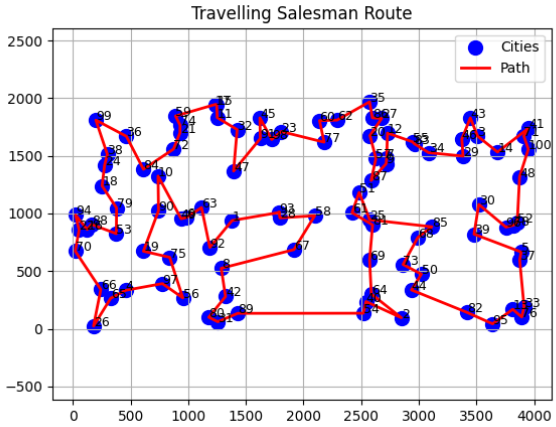
W implementacji wykorzystano następujące domyślne parametry:

- temperatura początkowa: `tempStart = 10000.0f`
- współczynnik chłodzenia: `alpha = 0.976f`
- liczba iteracji zewnętrznych:`iterations = 400`
- liczba iteracji wewnętrznych na każdą temperaturę: `innerLoop = 100`

```
void Tsp::simulatedAnnealing(float tempStart, float alpha, int iterations, int innerLoop)
{
    randomSolution();

    vector<City> currentPath = cities;
    std::rotate(currentPath.begin(), currentPath.begin() + 1, currentPath.end());
```

```
    float currentDist = getSolutionDistance();

    vector<City> bestPath = currentPath;
    float bestDist = currentDist;

    float temp = tempStart;

    for (int i = 0; i < iterations; ++i)
    {
        for (int j = 0; j < innerLoop; ++j)
        {
            int a = rand() % (currentPath.size() - 1);
            int b = rand() % (currentPath.size() - 1);
            if (a > b) std::swap(a, b);

            std::reverse(currentPath.begin() + a, currentPath.begin() + b + 1);

            float newDist = 0.0f;
            for (size_t k = 0; k < currentPath.size() - 1; ++k)
                newDist += countDistance(currentPath[k], currentPath[k + 1]);
            newDist += countDistance(currentPath.back(), currentPath.front());

            float delta = newDist - currentDist;

            if (delta < 0 || (exp(-delta / temp) > ((float)rand() / RAND_MAX)))
            {
                currentDist = newDist;
                if (currentDist < bestDist)
                {
                    bestDist = currentDist;
                    bestPath = currentPath;
                }
            }
            else
            {
                std::reverse(currentPath.begin() + a, currentPath.begin() + b + 1);
            }
        }

        temp *= alpha;
    }

    cities = bestPath;
}
```

W każdej iteracji algorytm wybiera losową modyfikację obecnego rozwiązania (np. zamianę dwóch miast) i oblicza różnicę w długości trasy. Gorsze rozwiązania są akceptowane z pewnym prawdopodobieństwem, które maleje wraz ze spadkiem temperatury – mechanizm ten pozwala algorytmowi „wyjść" z lokalnych minimów i zwiększa szansę na znalezienie lepszego rozwiązania globalnego.



Rysunek 10: Wizualiacja trasy - symulowane wyżarzanie

SA	
time [ns]	distance
271510800	23865.1

Rysunek 11: Tabela wynikow - symulowane wyżarzanie

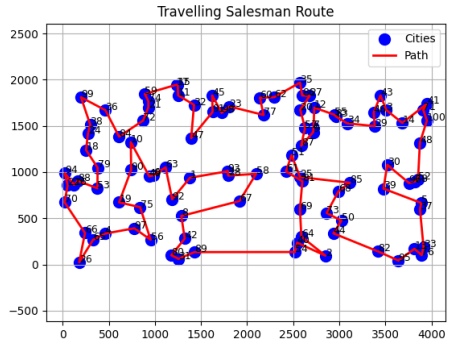
Na rysunku przedstawiono przykładową trasę uzyskaną metodą symulowanego wyżarzania, a w tabeli zaprezentowano czas wykonania i długość trasy. Wyniki pokazują, że metoda ta daje bardzo dobre rezultaty jakościowe, często zbliżone do najlepszych uzyskanych przez inne podejścia, przy umiarkowanym czasie działania.

7.2 Dostosowanie parametrow i ich wpływ na wyniki

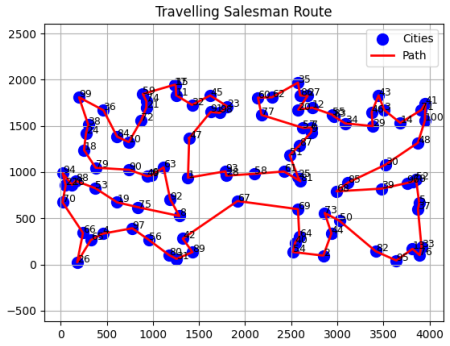
W celu oceny skuteczności algorytmu symulowanego wyżarzania, przeprowadzono badania z dwoma zestawami parametrów:

- domyślnymi — stosowanymi we wcześniejszych testach: `tempStart = 10000.0`, `alpha = 0.976`, `iterations = 400`, `innerLoop = 100`,
- dostosowanymi — dostosowanymi dla danego zbioru w oparciu o jakość rozwiązań i czas działania.

7.2.1 Zbiór `kroA100`



(a) Domyślne



(b) Dostosowane

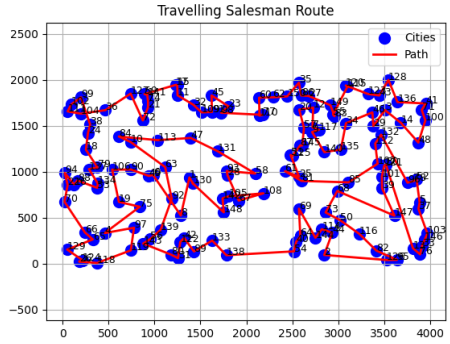
Rysunek 12: Wizualizacja tras dla kroA100

kroA100		
parametry	domyślne	dostosowane
Temp_wew	10000	10000
alpha	0.976	0.976
iter_zew	400	400
iter_wew	100	200
czas [ns]	271510800	536869400
dystans	23865.1	22485.7

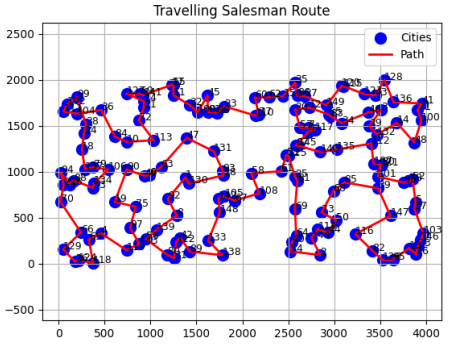
Rysunek 13: Porównanie wyników dla kroA100

Dostosowanie parametrów pozowiło skrócić trase o około 5%, co poskutkowało wydłużeniem czasu 2-krotnie.

7.2.2 Zbiór `kroA150`



(a) Domyślne



(b) Dostosowane

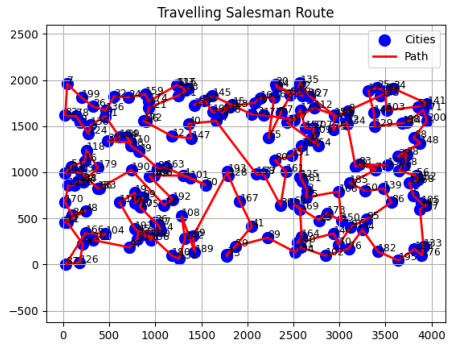
Rysunek 14: Wizualizacja tras dla kroA150

Dla zbioru danych kroA150 różnica jest znacząca, udało się skrócić długość trasy o około 10% jednakże czas wzrósł ponad 3 krotnie.

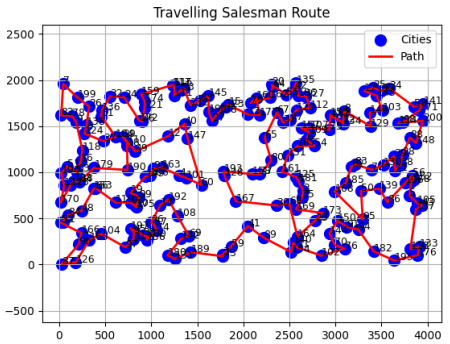
kroA150		
parametry	domyślne	dostosowane
Temp_wew	10000	12500
alpha	0.976	0.978
iter_zew	400	500
iter_wew	100	300
czas [ns]	401108700	1507709800
dystans	32801.3	27752.9

Rysunek 15: Porównanie wyników dla kroA150

7.2.3 Zbiór `kroA200`



(a) Domyślne



(b) Dostosowane

Rysunek 16: Wizualizacja tras dla kroA200

kroA200		
parametry	domyślne	dostosowane
Temp_wew	10000	18000
alpha	0.976	0.985
iter_zew	400	600
iter_wew	100	400
czas [ns]	537542600	3171047000
dystans	39632.9	31852.3

Rysunek 17: Porównanie wyników dla kroA200

W przypadku tego zbioru danych widzimy największą rożnice po odpowiednim dobraniu współczynników. Zarówno wizualznie udało pozbyć się skrzyżowań oraz trasa uległa polepszeniu o niecałe 20%. Jednakże czas wzrósł drastycznie, ponad 5 krotnie.

7.2.4 Podsumowanie wpływu parametrów

Dostosowanie parametrów algorytmu symulowanego wyżarzania ma istotny wpływ na jakość otrzymywanych rozwiązań. We wszystkich testowanych przypadkach udało się zauważalnie skrócić długość tras — od około 5% dla zbioru kroA100 do niemal 20% dla kroA200. Poprawa jakości wiąże się jednak ze znacznym wzrostem czasu wykonania, który w najcięższym przypadku (kroA200) wzrósł ponad pięciokrotnie. Wskazuje to na konieczność kompromisu między jakością rozwiązania a wydajnością czasową, a także na znaczenie indywidualnego dostrajania parametrów do rozmiaru i charakterystyki konkretnego problemu.

8 Podsumowanie

Poniższa tabela podsumowuje i zestawia ze sobą wyniki zaimplementowanych algorytmów dla zbioru danych `kroA100.tsp`. Wyniki testów wykazały wyraźne różnice w jakości uzyskiwanych rozwiązań oraz czasie wykonania.

	Podsumowanie - KROA100					
	123	Losowe	Sąsiad	Sąsiad + 2-opt	SA	SA - optymalne
time [ns]	100	5100	670400	4043900	271510800	536869400
distance	188750	178160	26856.4	22955.9	23865.1	22485.7

Rysunek 18: Podsumowanie wyników

Najprostsze metody, takie jak algorytm losowy czy naiwne wczytanie domyślnej kolejności, charakteryzują się bardzo szybkim czasem działania, jednak jakość ich wyników jest znacznie bardzo niska w porównaniu do bardziej zaawansowanych podejść. Algorytm najbliższego sąsiada pozwala osiągnąć zadowalający efekt jednak widać jego niedoskonalości, dlatego algorytm 2-opt stanowi jego skuteczne ulepszenie pozwalając znacząco skrócić długość ścieżki i pozbycie się skrzyżowań, przy wciąż rozsądnym czasie działania. Najlepsze rezultaty pod względem jakości rozwiązania uzyskano za pomocą algorytmu symulowanego wyżarzania dla dostosowanych parametró, który pozwolił na znalezienie ścieżek najbliższych optymalnym, kosztem dłuższego czasu działania.

Ciekawą obserwacją jest, że algorytm sąsiada + 2-opt pozwala osiągnąc lepsze wyniki niż klasczyna metoda symulowanego wyżarzania z domyślnymi parametrami. Dopiero ich dostosowanie pozwala osiągnąć najlepszy wynik.

Podsumowując, zastosowanie bardziej złożonych metod heurystycznych, takich jak 2-opt i symulowane wyżarzanie, pozwala uzyskać znacznie lepsze wyniki dla problemu TSP, co potwierdza ich przydatność w rozwiązywaniu zadań optymalizacji kombinatorycznej. Dobór odpowiedniego algorytmu zależy jednak od wymagań konkretnego zastosowania – w przypadkach wymagających szybkich przybliżeń warto rozważyć prostsze metody, natomiast gdy kluczowa jest jakość rozwiązania, lepszym wyborem będą algorytmy wykorzystujące lokalne przeszukiwanie lub strategie probabilistyczne.

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.

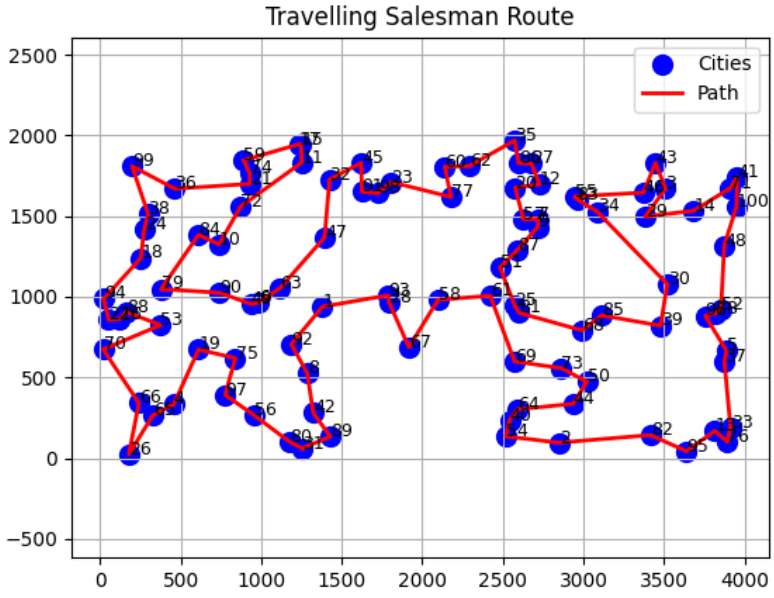
6. **Zastępowanie** - Aktualizacja populacji poprzez zachowanie najlepszych osobników oraz potomstwa.

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.