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.

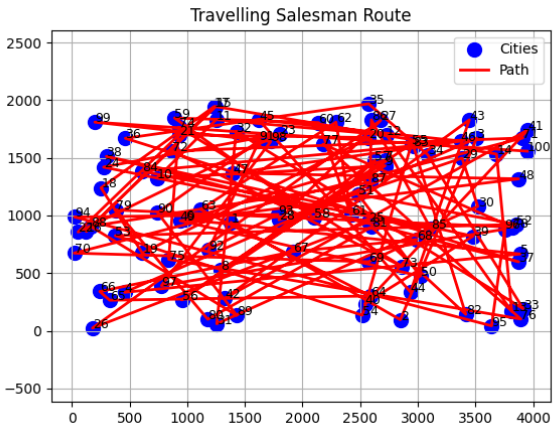
```
if __name__ == "__main__":
    if(len(sys.argv) < 2):
        print("Invalid amount of arguments")
        exit(1)

    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

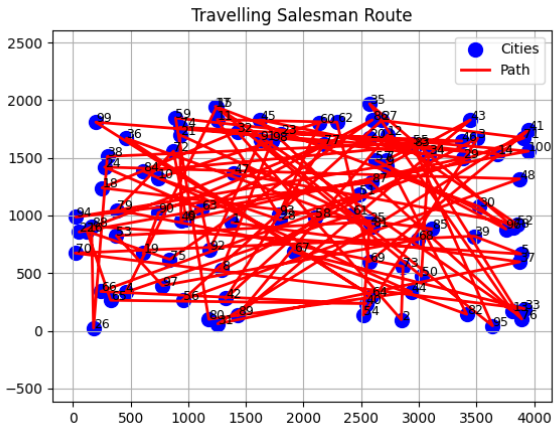
Na rysunku powyżej przedstawiono wizualizację tej domyślnej trasy, a poniżej niej w tabeli zaprezentowano czas działania (pomijalny, ponieważ nie zachodzi żadne przetwarzanie), a także uzyskaną długość trasy.

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.

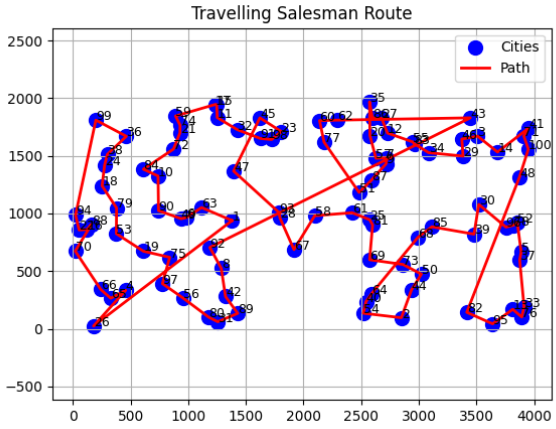
6 Algorytm najbliższego sąsiada

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

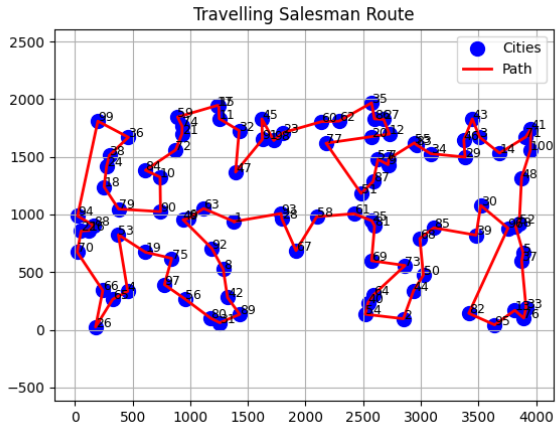
Choć algorytm działa szybko i jest prosty w implementacji, nie gwarantuje znalezienia optymalnej trasy. Wiele zależy od punktu startowego — w niektórych przypadkach może prowadzić do nieefektywnych rozwiązań. Nie rozwiązuje on problemu z krzyżowaniem tras

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

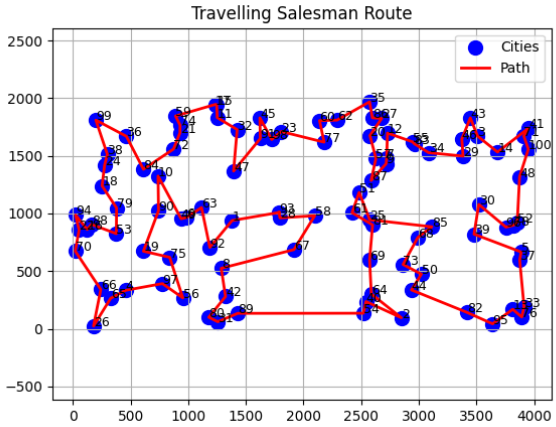
W implementacji wykorzystano następujące domyślne parametry:

- 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

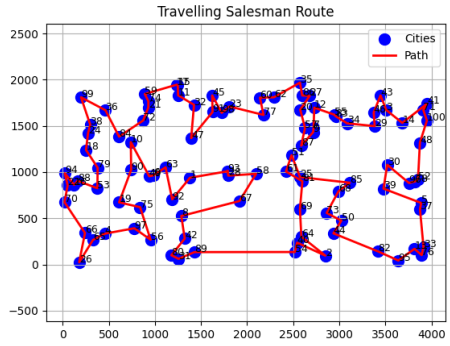
Na rysunku przedstawiono przykładową trasę uzyskaną metodą symulowanego wyżarzania, a w tabeli zaprezentowano czas wykonania i długość trasy. Wyniki pokazują, że metoda ta daje bardzo dobre rezultaty jakościowe, często zbliżone do najlepszych uzyskanych przez inne podejścia, przy umiarkowanym czasie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

7.2 Dostosowanie parametrow i ich wpływ na wyniki

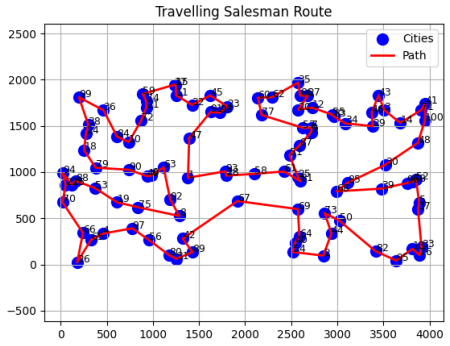
W celu oceny skuteczności algorytmu symulowanego wyżarzania, przeprowadzono badania z dwoma zestawami parametrów:

- 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

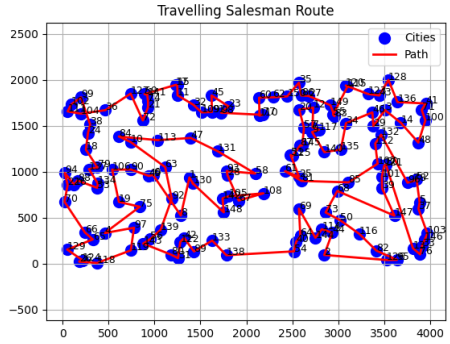
Rysunek 12: Wizualizacja tras dla kroA100

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

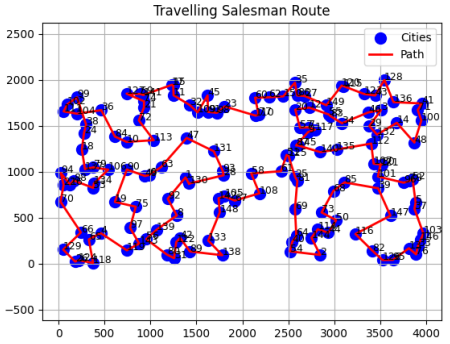
Rysunek 13: Porównanie wyników dla kroA100

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

7.2.2 Zbiór `kroA150`



(a) Domyślne



(b) Dostosowane

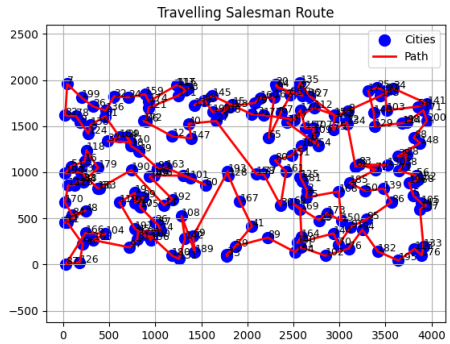
Rysunek 14: Wizualizacja tras dla kroA150

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

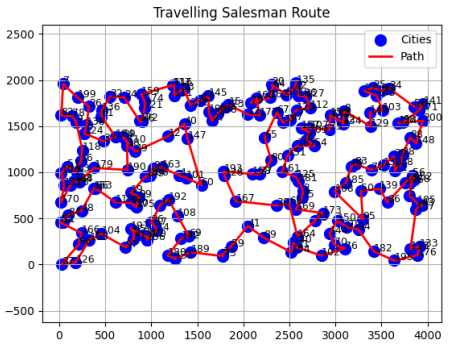
kroA150		
parametry	domyślne	dostosowane
Temp_wew	10000	12500
alpha	0.976	0.978
iter_zew	400	500
iter_wew	100	300
czas [ns]	401108700	1507709800
dystans	32801.3	27752.9

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.