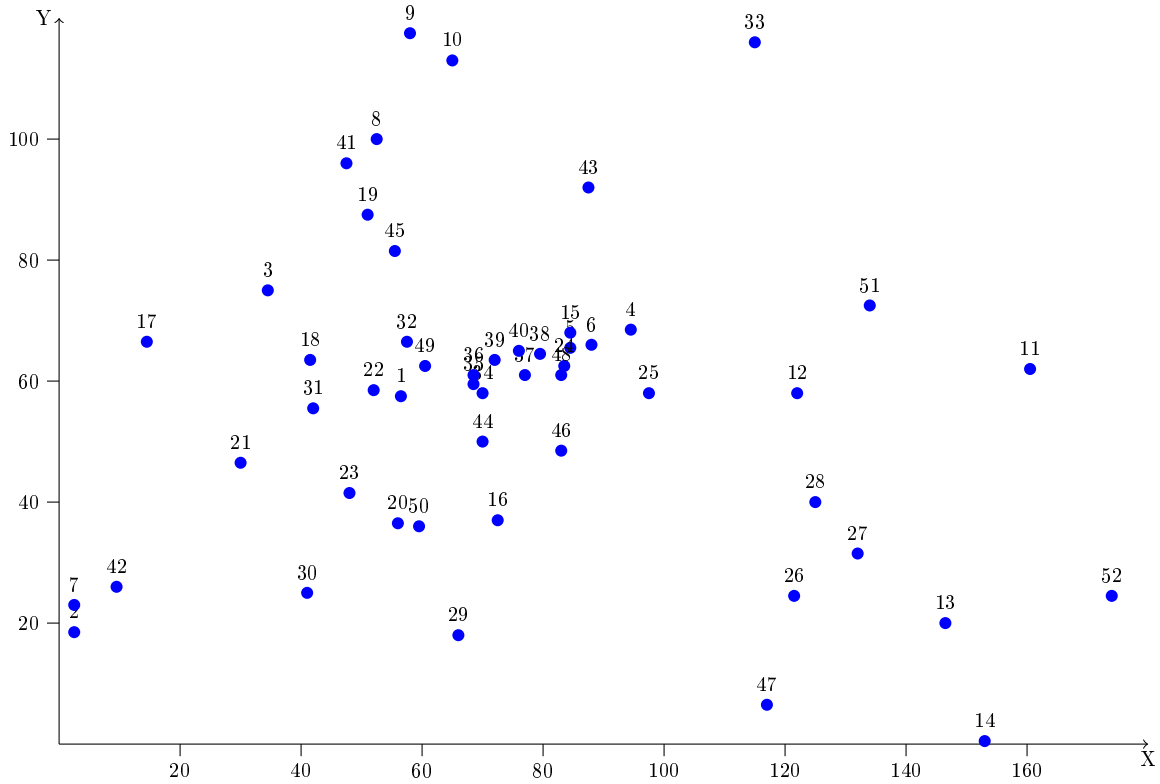
Problem komiwojażera - TSP

Julia Mista 160173
Kacper Skaza 160174

1 Algorytm mrówkowy

1.1 Inicjalizacja

Jako instancję poczatkową wybraliśmy berlin52, dla której wartości współrzednych punktów zostały przedstawione w skali 1:10 dla zwiększenia przejrzystości wykresu.



1.2 Opis algorytmu

Algorytm mrówkowy oparty jest na symulacji zachowania mrówek, które poszukując najkrótszej trasy między punktami pozostawiaja po sobie feromony. Nasz kod podzielony jest na kilka głównych funkcji takich jak: *tspAntColony, antTravel, selectNextCity* i *updatePheromones*. Algorytm zaczyna się od wywołania funkcji *tspAntColony*, która zgodnie ustaloną ilością iteracji oraz ilością mrówek wywołuje funkcję *antTravel*, wykonujacą trase dla jednej mrówki. Wybiera ona losowy punkt startowy, a następnie przechodzi przez resztę grafu, wybierając kolejne destynacje na podstawie prawdopodobieństwa zależnego od intensywności feromonu i odległości między punktami. Po przejściu wszystkich mrówek w danej iteracji wywoływana jest funkcja *updatePheromones*, odpowiadajaca za aktualizacje poziomów feromonów w grafie.

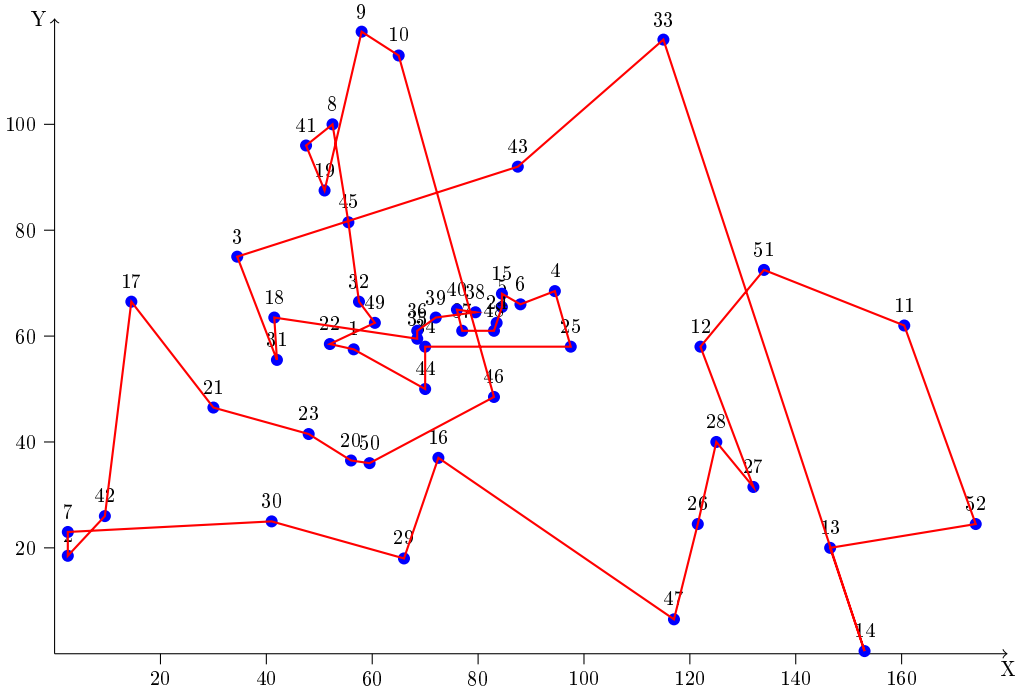
1.3 Pseudokod

```
funkcja updatePheromones(n, pheromone, paths, lengths):
        deltaTau [0..n-1][0..n-1] ← 0.0

        dla ant = 0,1,...,NUM_ANTS-1 wykonuj:
            dla i = 0,1,...,n - 2 wykonuj:
                cityA ← paths[ant][i]
                cityB ← paths[ant][i+1]
                deltaTau[cityA][cityB] ← deltaTau[cityA][cityB] + Q / lengths[ant]
                deltaTau[cityB][cityA] ← deltaTau[cityB][cityA] + Q / lengths[ant]

            lastCity ← paths[ant][n-1]
            firstCity ← paths[ant][0]
            deltaTau[lastCity][firstCity] ← deltaTau[lastCity][firstCity] + Q / lengths[ant]
            deltaTau[firstCity][lastCity] ← deltaTau[firstCity][lastCity] + Q / lengths[ant]

        dla i = 0,1,...,n - 1 wykonuj:
            dla j = 0,1,...,n - 1 wykonuj:
                pheromone[i][j] ← RHO * pheromone[i][j] + deltaTau[i][j]


funkcja selectNextCity(n, currentCity, graph, pheromone, visited):
    probabilities [0..n-1] ← 0.0
    sum ← 0.0

    dla i = 0,1,...,n-1 wykonuj:
        jeżeli visited[i] = Fałsz to:
            probabilities[i] ← (pheromone[currentCity][i]^ALPHA) *
            ((1.0 / graph[currentCity][i])^BETA)
            sum ← sum + probabilities[i]

    randVal ← (losowa_liczba() * sum) / (RAND_MAX * 1.0)
    cumulative ← 0.0

    dla i = 0,1,...,n - 1 wykonuj:
        jeżeli visited[i] = Fałsz to:
            cumulative ← cumulative + probabilities[i]
            jeżeli cumulative ≥ randVal to:
                zwróć i

    zwróć -1
```

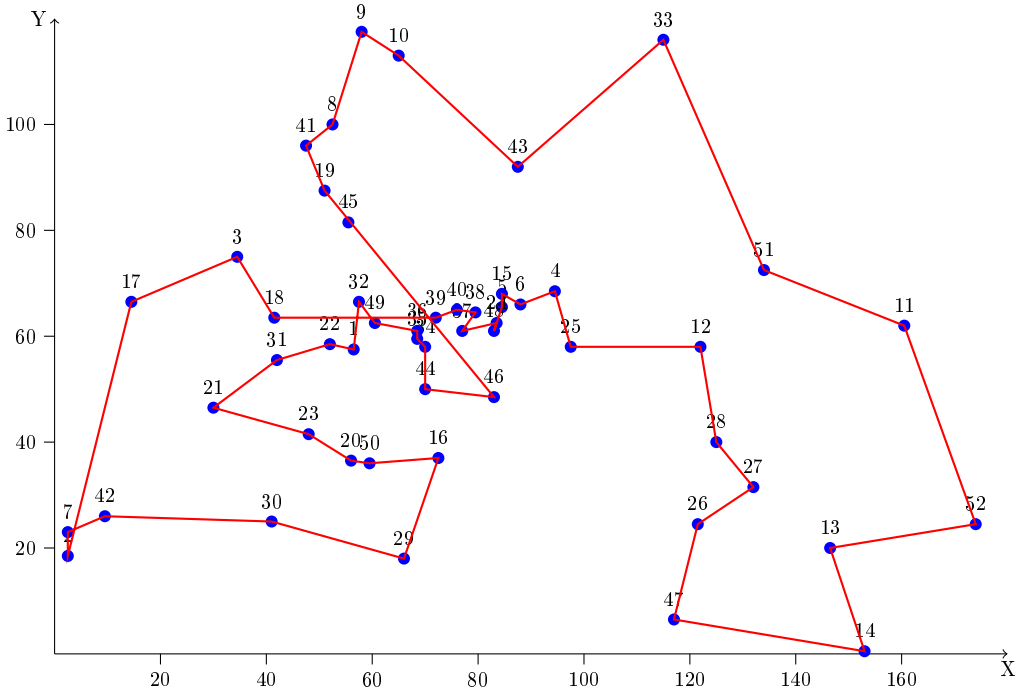
```
funkcja antTravel(n, antNum, paths, lengths, graph, pheromone):
    visited [0..n-1] ← Fałsz
    path []
    length ← 0.0
    startCity ← losowa_liczba() mod n
    currentCity ← startCity

    visited[currentCity] ← Prawda
    dodaj currentCity do path

    dla i = 0,1,...,n-1 wykonuj:
        nextCity ← selectNextCity(n, currentCity, graph, pheromone, visited)
        dodaj nextCity do path
        visited[nextCity] ← Prawda
        length ← length + graph[currentCity][nextCity]
        currentCity ← nextCity

    length ← length + graph[currentCity][startCity]
    dodaj startCity do path

    lengths[antNum] ← length
    paths[antNum] ← path


funkcja tspAntColony(n, bestPath, bestLength, graph):
    pheromone [0..n-1][0..n-1] ← 1.0
    sameResult ← 0

    dla i = 0,1,...,MAX_ITERATIONS wykonuj:
    paths [0..NUM_ANTS-1][]
    lengths [0..NUM_ANTS-1]

    threads []

    dla ant = 0,1,...,NUM_ANTS - 1 wykonuj:
        dodaj nowy wątek do threads z wywołaniem antTravel(n, ant, paths,
        lengths, graph, pheromone)

    dla każdego wątku w threads:
        poczekaj na zakończenie wątku

    dla ant = 0,1,...,NUM_ANTS - 1 wykonuj:
        jeżeli lengths[ant] < bestLength to:
            bestLength ← lengths[ant]
            bestPath ← paths[ant]
            sameResult ← -1

    updatePheromones(n, pheromone, paths, lengths)

    sameResult ← sameResult + 1
    jeżeli sameResult > MAX_SAME_ITERATIONS i i > MIN_ITERATIONS:
        zakończ algorytm
```

1.4 Przykład działania

- po wykonaniu pierwszej iteracji



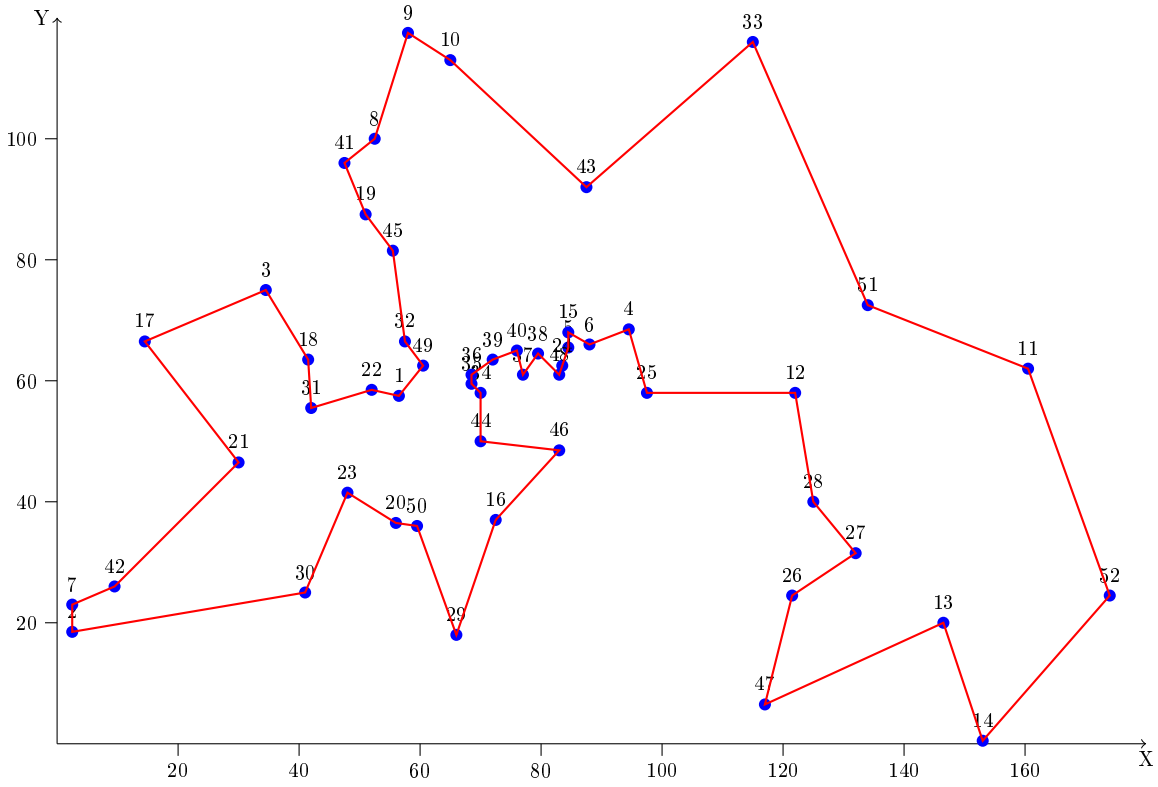
- po wykonaniu piątej iteracji



Na podstawie wykresów zaobserwować można stopniowe poprawianie się rozwiązania wraz z kolejnymi iteracjami algorytmu.

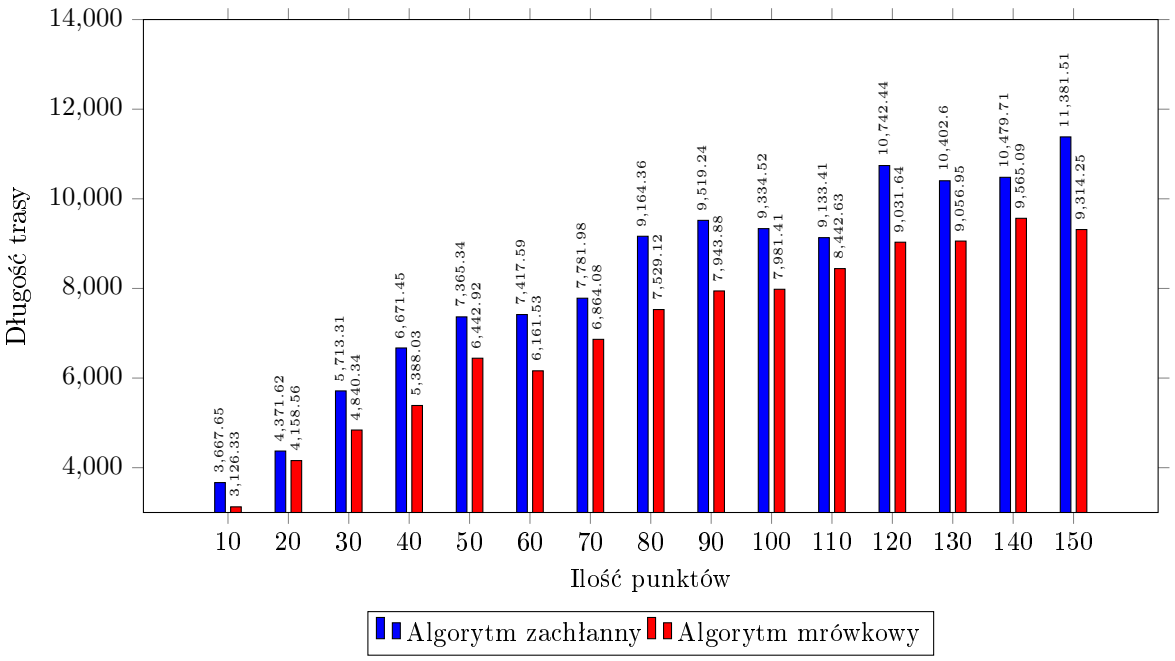
1.5 Finalizacja

Po wykonaniu dziesięciu iteracji dochodzimy do najlepszego wyniku uzyskiwanego przez naszą implementacje dla instancji berlin52 - długość trasy: 7544,37.

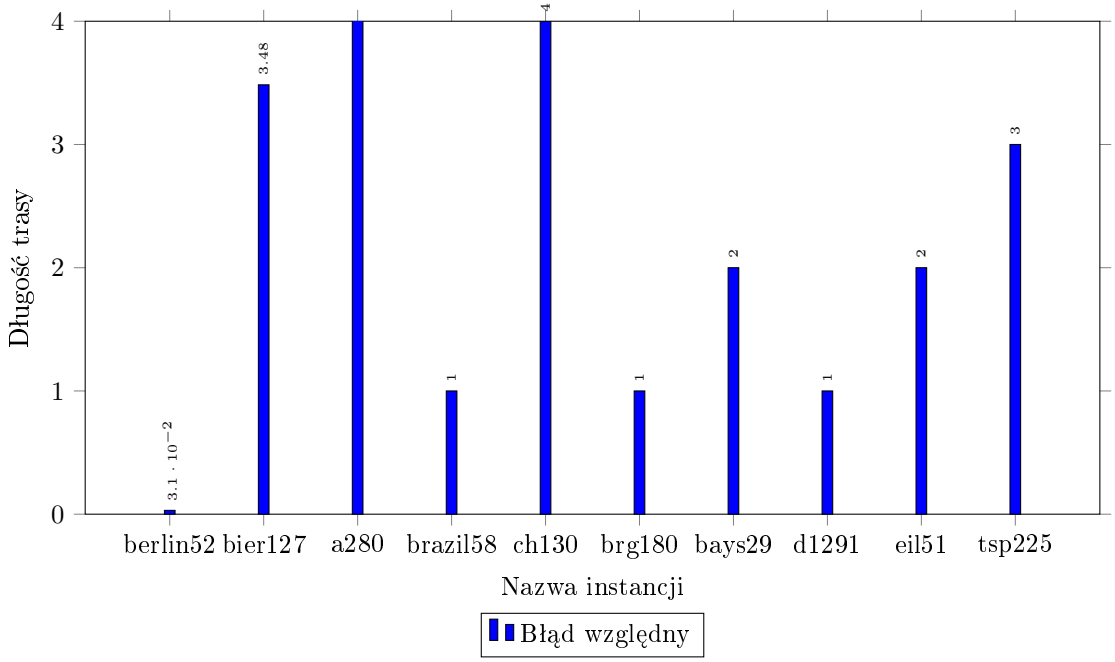


1.6 Wykresy

- porównanie wyników uzyskanych algorytmem mrówkowym i zachłannym dla losowych instancji o różnej wielkości



- wartość błędu względnego naszego algorytmu w stosunku do wartości optymalnej dla wybranych benchmarków



1.7 Wyniki dla instancji rankingowych

Nazwa instancji	Uzyskany wynik
berlin52	7544,37
bier127	122402,23
tsp1000	24246
tsp500	92214,46
tsp250	13071,16