

UNIWERSYTET TECHNOLOGICZNO-PRZYRODNICZY

IM. J. I J.ŚNIADECKICH W BYDGOSZCZY

WYDZIAŁ TELEKOMUNIKACJI, INFORMATYKI I ELEKTROTECHNIKI ZAKŁAD TECHNIKI CYFROWEJ

Sztuczna inteligencja

Algorytm Mrówkowy dla zastosowania problemu komiwojażera

Michał Sulecki

Dominik Wiśniewski

[1. Wstęp teoretyczny 3](#_Toc42804777)

[1) Problem komiwojażera 3](#_Toc42804778)

[2) Opis algorytmu mrówkowego 3](#_Toc42804779)

[3) Lista użytych miast 3](#_Toc42804780)

[2. Założenia programu oraz prezentacja interfejsu graficznego 4](#_Toc42804781)

[1) Opis wykorzystanego oprogramowania 4](#_Toc42804782)

[2) Założenia dla aplikacji 4](#_Toc42804783)

[3) Interfejs graficzny 5](#_Toc42804784)

[3. Implementacja algorytmu w programie 7](#_Toc42804785)

[4. Badanie algorytmu – próby/wyniki/wykresy (?) 7](#_Toc42804786)

[5. Podsumowanie 8](#_Toc42804787)

[6. Bibliografia 8](#_Toc42804788)

# Wstęp teoretyczny

## Problem komiwojażera

Problem wędrującego komiwojażera (ang. TSP – Traveling Salesman Problem) polega na odwiedzeniu każdego z miast w celu sprzedaży towarów. Komiwojażer wyrusza z jednego miasta i przechodzi przez każde kolejne tylko raz i wraca do punktu startowego. Ze wszystkich możliwych dróg, wybiera najkrótszą możliwą drogę, która spełni podany warunek. Najkrótsza trasa określa trasę o najmniejszym „koszcie” – przykładowo może być to trasa najkrótsza długością, czasem, kosztem pieniężnym przebycia trasy.

Korzystając z listy miast, tworzymy graf zupełny, którego wagami na krawędziach są długości pomiędzy miastami. Poprzez algorytm, wyszukiwany jest cykl Hamiltona, który posiada minimalną sumę wag krawędzi.

## Opis algorytmu mrówkowego

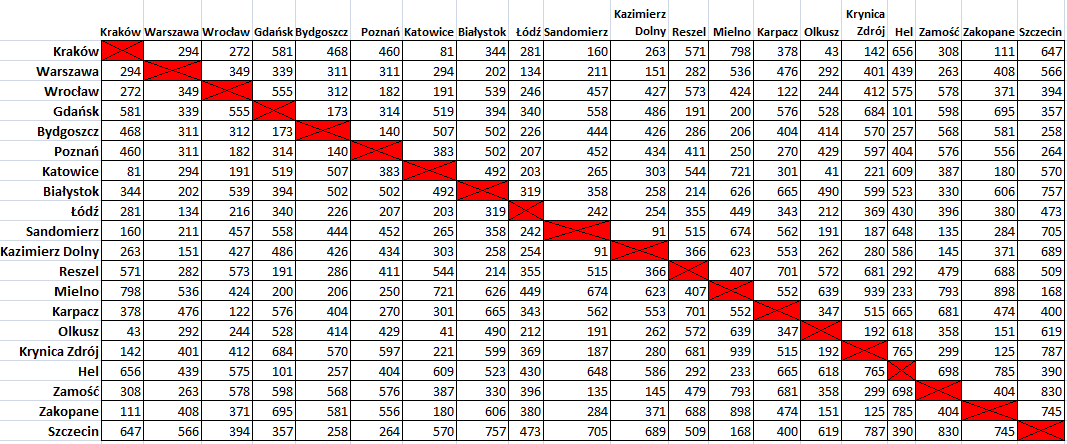
Algorytm mrówkowy jest przedstawicielem grupy algorytmów genetycznych. Inspiracją do jego powstania było zachowanie tytułowych mrówek. Mrówki poszukują jedzenia i jako rój tworzą ścieżki, po których zostawiają feromony. Poruszają się po trasach, które mają najwięcej feromonów – czyli takich, które są najbardziej uczęszczane. Jeżeli mrówka znajdzie pożywienie bliżej mrowiska, feromon utrzyma się dłużej, niż na trasie dalszej, z powodu jego ciągłego parowania w atmosferze. Im więcej feromonu, tym częściej rój wybiera daną ścieżkę cały czas pozostawiając feromon. Mrówki komunikują się poprzez wyczuwanie feromonu; im więcej feromonu tym większa szansa znalezienia pożywienia.

## Lista użytych miast

W celu przetestowania algorytmu w rzeczywistości, wybrane zostało 20 największych miast w Polsce względem populacji. Algorytm może zostać wykorzystany do np.:  
- Wyznaczania najkrótszej trasy turystycznej po wybranych miastach  
- Wyznaczenie najkrótszej trasy dla transportu dużych towarów między głównymi punktami w miastach.

Aby korzystać z listy miast, należy utworzyć macierz n x n , gdzie n oznacza liczbę miast, a komórki zawierają odległości między nimi. Odległości miast zostały wygenerowane poprzez Google Maps – zawsze wybierana droga najszybsza, podana w kilometrach, która bazuje na rzeczywistych trasach.

Za oznaczenie X na czerwonym polu odpowiada wartość odległości miasta do samego siebie, która w programie wynosi 0.



# Założenia programu oraz prezentacja interfejsu graficznego

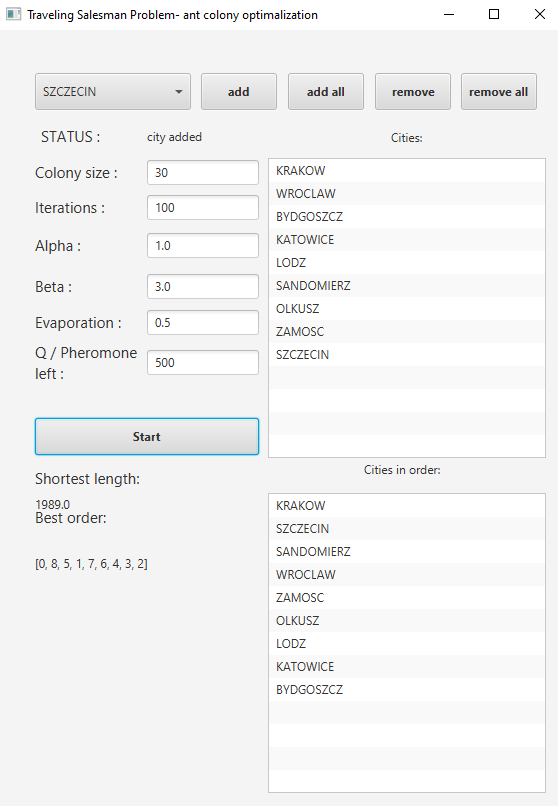
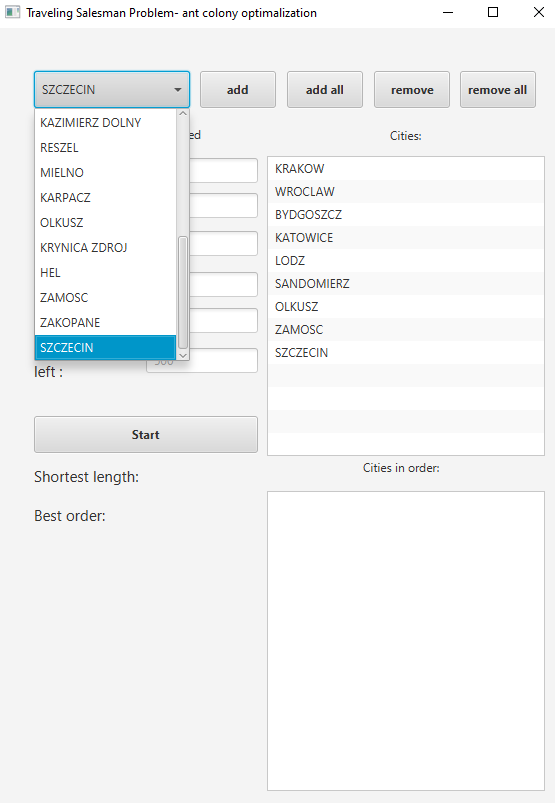
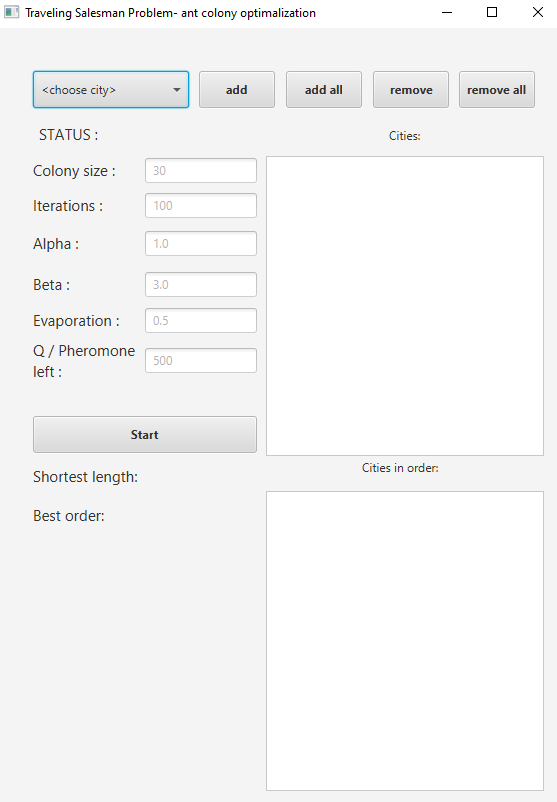
* 1. Opis wykorzystanego oprogramowania

Programową realizację algorytmu mrówkowego wykonano w języku Java z wykorzystaniem elementów interfejsu graficznego JavaFX.

Program aktualnie zawiera bazę 20 największych miast opisanych w punkcie powyżej, z możliwością rozszerzenia funkcjonalności o dodanie interfejsu Google Maps API , które wymaga klucza dostępu i założonego konta transakcyjnego. Z tego powodu na potrzeby badań i testów, wykorzystano stałą macierz odległości, które w przyszłości można rozwinąć.

## Założenia dla aplikacji

Użytkownik wybiera od 1 do 20 niepowtarzalnych miast z listy z których program ma wyliczyć najkrótszą możliwą ścieżkę komiwojażera wraz z podaną kolejnością. Kolejność musi zostać zachowana, jednak start może nastąpić z dowolnego miasta aby uzyskać ten sam wynik – cykl Hamiltona.

* 1. Interfejs graficzny  
     

Opis kontrolerów użytych w interfejsie graficznym:

- rozwijalna lista, zawierająca pulę 20 miast, które możemy dodać do widoku listy ‘cities’, celem wyznaczenia najkrótszej trasy,

- przyciski: *add*, *add* *all*, *remove*, *remove* *all*, służące kolejno do: dodania pojedynczego miasta wybranego z rozwijalnej listy obok do widoku listy ‘cities’, dodania wszystkich miast z puli do widoku listy ‘cities’, usunięcia zaznaczonego miasta z widoku listy ‘cities’, usunięcia wszystkich wybranych miast z widoku listy ‘cities’,

- widok listy ‘cities’- widok listy, zawierający wszystkie miasta wybrane przez użytkownika,

- pole *Colony size*- ilość stworzonych mrówek,

- pole *Iterations*- ilość przejść przez trasy,

- pole *Alpha*- waga/ ważność feromonu,

- pole *Beta*- priorytet odległości,

- pole *Evaporation*- parowanie feromonu,

- pole *Q/ Pheromone left*- feromon zostawiany przez jedną mrówkę na ścieżce,

- przycisk *start*- po wciśnięciu go, program się uruchamia wykorzystując wszystkie wyżej podane parametry przez użytkownika oraz wypisuje poniżej najkrótszą odległość i najlepszą kolejność odwiedzenia miast,

- widok listy ‘cities in order’- widok listy zawierający po kolei miasta, które należy odwiedzić w celu jak najkrótszej trasy.

# Implementacja algorytmu w programie

//Opisać klasę Ant, opisać parametry i metody ACO

Aplikację napisaliśmy w języku Java a dodatkowo wykorzystaliśmy bibliotekę JavaFX do stworzenia interfejsu użytkownika. Językiem przewodnim w aplikacji jest język angielski, jednak nazwy miast są po polsku- ze względu na to, że są to polskie miasta. Cały kod aplikacji jest dostępny w repozytorium GitHub, do którego link znajduje się w bibliografii.

Klasa Ant- tutaj są tworzone nowe mrówki. Klasa ta odpowiedzialna jest za to, aby każda wyprodukowana mrówka miała możliwość odwiedzenia danego miasta, zapamiętania wszystkich odwiedzonych miast oraz posiadania wiedzy na temat długości ścieżki.

Klasa ACO- klasa, w której cały algorytm jest wykonywany. Metody, odgrywające istotne role w algorytmie to:

- tworzenie roju mrówek,

- implementacja macierzy 20x20 z odległościami pomiędzy miastami,

- generowanie macierzy n x n z odległościami pomiędzy wybranymi przez użytkownika miastami gdzie n- liczba wybranych miast,

- przygotowanie mrówek do symulacji (zaczęcie podróży od losowego miasta),

- przemieszczenie mrówek przy każdej iteracji (pamiętając o tym, że każda mrówka stara się podążać za poprzedzającą ją),

- wybór następnego miasta do odwiedzenia przez każdą mrówkę,

- liczenie prawdopodobieństwa, które miasto zostanie wybrane jako następne,

- zaktualizowanie trasy, którą przeszła każda z mrówek,

- zaktualizowanie najlepszej (najkrótszej) trasy,

- wyczyszczenie tras po zakończeniu symulacji.

Klasa FXMLController- klasa odpowiedzialna za stworzenie interfejsu graficznego użytkownika. Znajdują się tam definicje przycisków, cała logika działania interfejsu oraz gettery, aby można było m.in. przekazać listę i liczbę miast do stworzenie algorytmu.

W programie użyliśmy BidiMapy, która w przeciwieństwie do HashMapy pozwala nam nie tylko szukać wartości po kluczach ale również kluczy po wartości.

# Badanie algorytmu – próby/wyniki/wykresy (?)

Badanie na podstawie różnych parametrów, np. 30/50/100 mrówek, parametr zostawiania i parowania feromonu, alfa 1.0,3.0,5.0, beta 1.5,5.0,12.0 itp. Itd. Warto odwołać się do Optimap, które też bazuje na ACO, ale wyniki mogą się nieznacznie różnić ze względu na różne dostępne trasy na mapach ( mnogość małych różnic dróg pomiędzy miastami mającymi kilka tras dostępnych). Dodatkowo można sprawdzać działanie na alife.pl

# Podsumowanie/wnioski

# Bibliografia

* OptiMap – Route planer for Google Maps  
  <http://www.optimap.net/>
* Behind the Scenes of OptiMap  
  <https://gebweb.net/blogpost/2007/07/05/behind-the-scenes-of-optimap>
* Repozytorium GitHub naszego projektu

<https://github.com/MCtyskie/AIProject>