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, 113112

Dominik Wiśniewski, 113131

1. Wstęp teoretyczny 3
   1) Problem komiwojażera 3
   2) Opis algorytmu mrówkowego 3
   3) Lista użytych miast 3
2. Założenia programu oraz prezentacja interfejsu graficznego 4
   1) Opis wykorzystanego oprogramowania 4
   2) Założenia dla aplikacji 4
   3) Interfejs graficzny 5
3. Implementacja algorytmu w programie 9
4. Badanie algorytmu – próby/wyniki/wykresy 19
5. Podsumowanie/ wnioski 21
6. Bibliografia 23

1. Wstęp teoretyczny

1) Problem komiwojażera

Problem wędrującego komiwojażera (ang. TSP – TravelingSalesman 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.

2) 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.

3) 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.

	Kraków	Warszawa	Wrocław	Gdańsk	Bydgoszcz	Poznań	Katowice	Białystok	Łódź	Sandomierz	Kazimierz Dolny	Reszel	Mielno	Karpacz	Olkusz	Krynica Zdrój	Hel	Zamość	Zakopane	Szczecin
Kraków		294	272	581	468	460	81	344	281	160	263	571	798	378	43	142	656	308	111	647
Warszawa	294		349	339	311	311	294	202	134	211	151	282	536	476	292	401	439	263	408	566
Wrocław	272	349		555	312	182	191	539	246	457	427	573	424	122	244	412	575	578	371	394
Gdańsk	581	339	555		173	314	519	394	340	558	486	191	200	576	528	684	101	598	695	357
Bydgoszcz	468	311	312	173		140	507	502	226	444	426	286	206	404	414	570	257	568	581	258
Poznań	460	311	182	314	140		383	502	207	452	434	411	250	270	429	597	404	576	556	264
Katowice	81	294	191	519	507	383		492	203	265	303	544	721	301	41	221	609	387	180	570
Białystok	344	202	539	394	502	502	492		319	358	258	214	626	665	490	599	523	330	606	757
Łódź	281	134	216	340	226	207	203	319		242	254	355	449	343	212	369	430	396	380	473
Sandomierz	160	211	457	558	444	452	265	358	242		91	515	674	562	191	187	648	135	284	705
Kazimierz Dolny	263	151	427	486	426	434	303	258	254	91		366	623	553	262	280	586	145	371	689
Reszel	571	282	573	191	286	411	544	214	355	515	366		407	701	572	681	292	479	688	509
Mielno	798	536	424	200	206	250	721	626	449	674	623	407		552	639	939	233	793	898	168
Karpacz	378	476	122	576	404	270	301	665	343	562	553	701	552		347	515	665	681	474	400
Olkusz	43	292	244	528	414	429	41	490	212	191	262	572	639	347		192	618	358	151	619
Krynica Zdrój	142	401	412	684	570	597	221	599	369	187	280	681	939	515	192		765	299	125	787
Hel	656	439	575	101	257	404	609	523	430	648	586	292	233	665	618	765		698	785	390
Zamość	308	263	578	598	568	576	387	330	396	135	145	479	793	681	358	299	698		404	830
Zakopane	111	408	371	695	581	556	180	606	380	284	371	688	898	474	151	125	785	404		745
Szczecin	647	566	394	357	258	264	570	757	473	705	689	509	168	400	619	787	390	830	745	

Tabela 1- odległości pomiędzy miastami

2. Założenia programu oraz prezentacja interfejsu graficznego

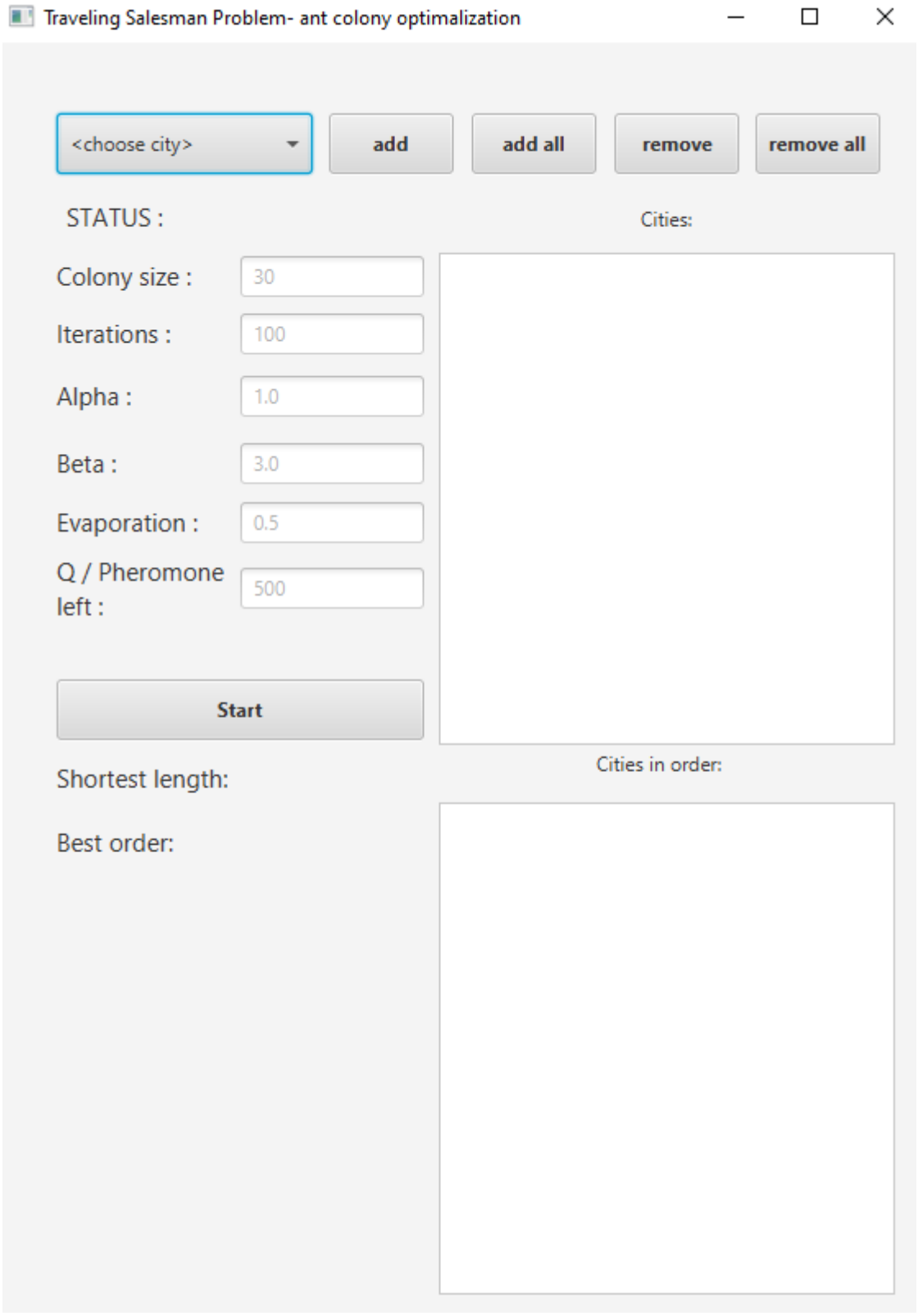
1) Opis wykorzystanego oprogramowania

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.
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

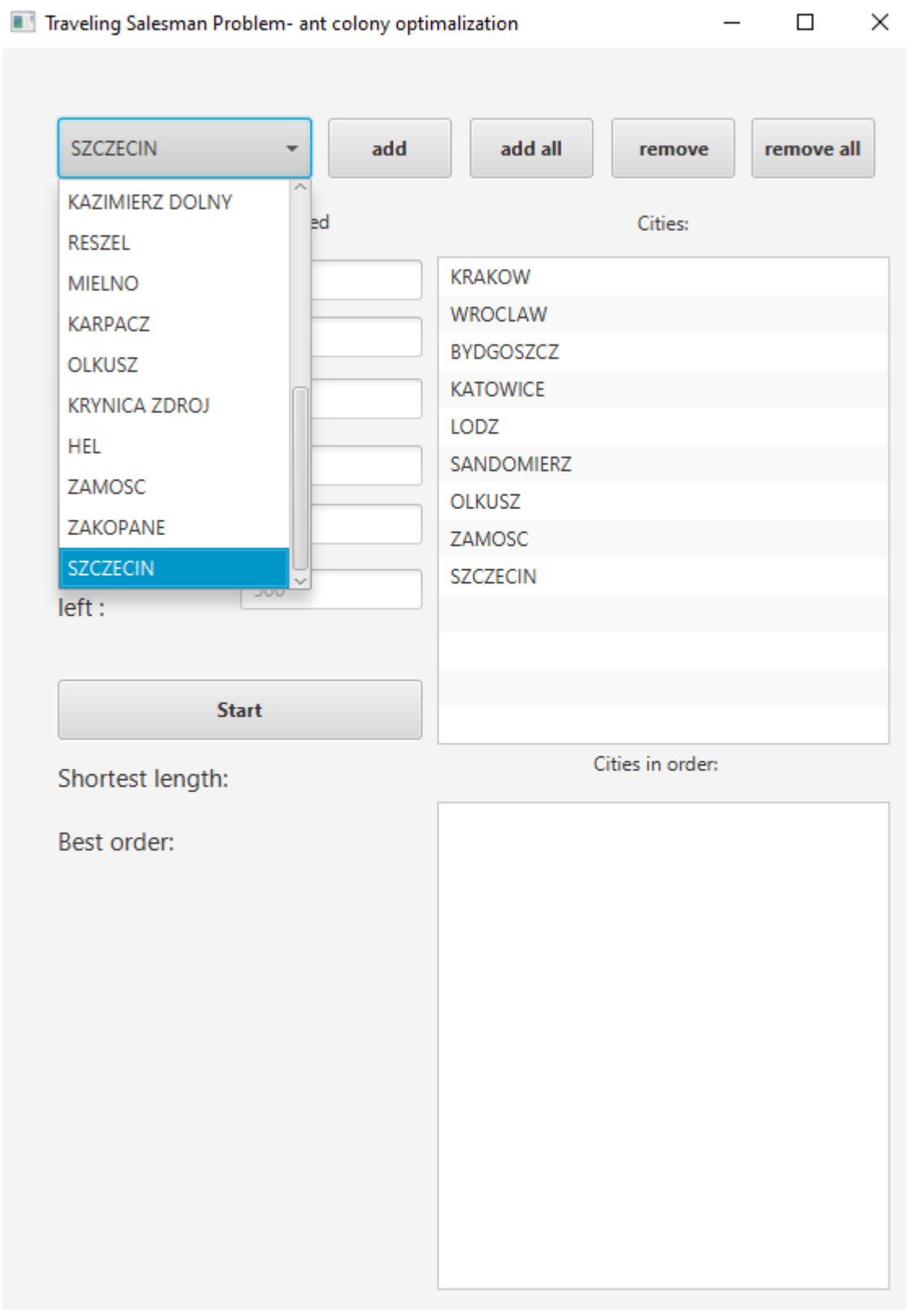
2) Założenia dla aplikacji

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.
Dla opisanych poniżej parametrów występuje odgórne założenie poprawiające rezultaty; Ogólnikowo podchodząc do algorytmu, parametr beta musi być większy od parametru alfa, ponieważ wpływ parametrem beta na większą eksplorację jest ważniejszy w celu odnalezienia optymalnej trasy, niż eksploatacja parametrem alfa. Odpowiednie dobrane te dwa parametry decydują o całości rozwiązania.

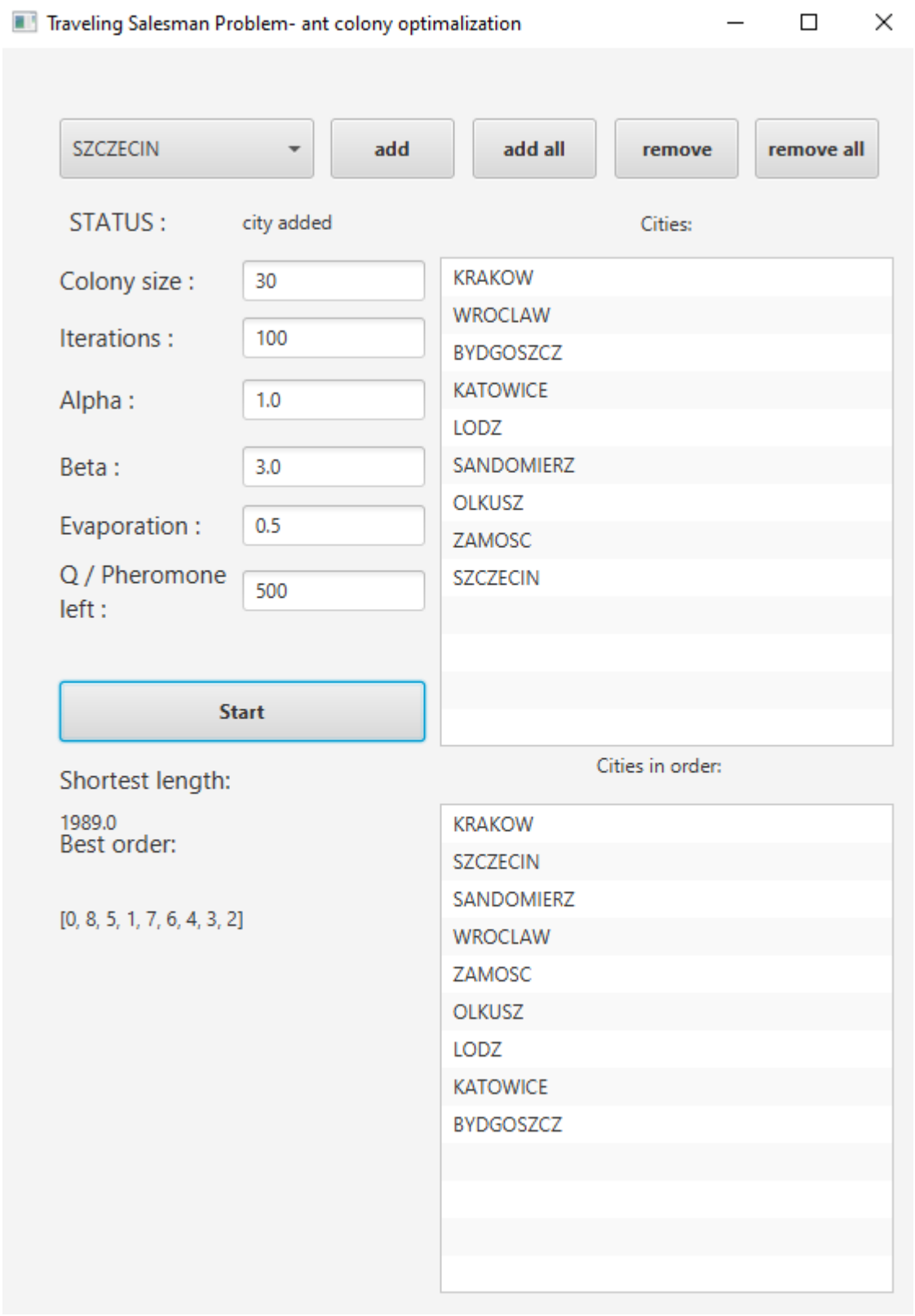
3) Interfejs graficzny



Rysunek 1- początkowy widok interfejsu



Rysunek 2- wybór miast



Rysunek 3- przykładowy wynik działania programu

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 pulido widoku listy ‘cities’, usunięcia zaznaczonego miasta z widoku listy ‘cities’, usunięcia wszystkich wybranych miastz 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/wpływ feromonu na danej ścieżce,

- 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 cykl kolejności odwiedzenia miast dla najbardziej optymalnej trasy.

3. Implementacja algorytmu w programie

Klasa Ant

```
@Data
public class Ant {
    private int routeSize;
    private int route[];
    private boolean visited[];

    public Ant(int routeSize){
        this.routeSize = routeSize;
        this.route = new int[routeSize];
        this.visited = new boolean[routeSize];
    }

    public boolean getVisitedByIdx(int idx){
        return visited[idx];
    }

    public int getRouteByIdx(int idx){
        return route[idx];
    }

    public void visitCity(int currentIdx,int city){
        route[currentIdx+1]=city;
        visited[city]=true;
    }
    public double routeLength(double graph[][]){
        double length = graph[route[routeSize-1]][route[0]];
        for(int i=0;i<routeSize-1;i++){
            length += graph[route[i]][route[i+1]];
        }
        return length;
    }

    public void clear() {
        for(int i=0;i<routeSize;i++){
            visited[i]=false;
        }
    }


}
```

Klasa mrówki posiada pola :

- Długość ścieżki
- Tablicę ścieżek
- Tablicę odwiedzonych miast

Każda mrówka ma możliwość odwiedzenia danego miasta, zapamiętania wszystkich odwiedzonych miast oraz posiadania wiedzy na temat długości ścieżki. Program tworzy podaną ilość obiektów mrówek, które działają po grafie wybranych miast.

Klasa ACO

```
@Setter
public class ACO {
    private double c =1.0;
    private double alpha;//= 1;
    private double beta; //= 5;
    private double evaporation; //= 0.5;
    private double Q; //= 500;
    private double randomFactor= 0.01;

    private int iterations;// = 1000;

    private List<Ant> ants = new ArrayList<>();
    private int numOfCities;
    private int numOfAnts;
    private double graph[][];
    private double routes[][];
    private Random random = new Random();
    private double probabilities[];

    private int currentIdx;
    private int[] bestOrder;
    private double bestLength;

    private static BidiMap<Integer,String> citiesMap;

    public ACO(int numOfSelectedCities, int colonySize){

        graph = generateCitiesMatrix(numOfSelectedCities);
        numOfCities = graph.length;
        numOfAnts=colonySize;
        routes = new double[numOfCities][numOfCities];
        probabilities = new double[numOfCities];
        createAnts(numOfAnts);
    }

    public int[] getBestOrder(){
        return Arrays.copyOf(bestOrder,bestOrder.length);
    }

    public void createAnts(int numOfAnts){
        IntStream.range(0, numOfAnts).forEach(i -> ants.add(new Ant(numOfCities)));
    }
```

```
public double[][] allCitiesMatrix(){
    //double[][] matrix=new double[20][20];
    double[][]matrix={ {0,294,272,581,468,460,81,344,281,160,263,571,798,378,43,142,656,308,111,647},
                       {294,0,349,339,311,311,294,202,134,211,151,282,536,478,292,401,439,263,408,566},
                       {272,349,0,555,312,182,191,539,246,457,427,573,424,122,244,412,575,578,371,394},
                       {581,339,555,0,173,314,519,394,340,558,486,191,200,576,528,684,101,598,695,357},
                       {468,311,312,173,0,140,507,502,226,444,426,286,206,404,414,570,257,568,581,258},
                       {460,311,182,314,140,0,383,502,207,452,434,411,250,270,429,597,404,576,556,264},
                       {81,294,191,519,507,383,0,492,203,265,303,544,721,301,41,221,609,387,180,570},
                       {344,202,539,394,502,502,492,0,319,358,258,214,626,665,490,599,523,330,606,757},
                       {281,134,216,340,226,207,203,319,0,242,254,355,449,343,212,369,430,396,380,473},
                       {160,211,457,558,444,452,265,358,242,0,91,515,674,562,191,187,648,435,284,705},
                       {263,151,427,486,426,434,303,258,254,91,0,366,623,553,262,280,586,145,371,689},
                       {571,282,573,191,286,411,544,214,355,515,366,0,407,701,572,681,292,479,688,509},
                       {798,536,424,200,206,250,721,626,449,674,625,407,0,552,639,939,233,793,898,168},
                       {378,476,122,576,404,270,301,665,343,562,553,701,552,0,347,515,665,681,474,400},
                       {43,292,244,528,414,429,41,490,212,191,262,572,639,347,0,192,618,358,151,619},
                       {142,401,412,684,570,597,221,599,369,187,280,681,939,515,192,0,765,299,125,787},
                       {656,439,575,101,257,404,609,523,403,648,586,292,233,665,618,765,0,698,785,319},
                       {308,263,578,598,568,576,387,303,396,135,145,479,793,681,358,299,698,0,404,830},
                       {111,408,371,695,581,556,180,606,380,284,371,688,898,474,151,125,785,404,0,745},
                       {647,566,394,357,258,264,570,757,473,705,689,509,168,400,619,787,390,830,745,0}};
    return matrix;
}

public double[][] generateCitiesMatrix(int numOfCities) {
    double[][] allCitiesMatrix = allCitiesMatrix();
    if(numOfCities==20){
        return allCitiesMatrix();
    }
    else{
        double[][] citiesMatrix=new double[numOfCities][numOfCities];
        citiesMap = FXMLController.getSelectedCitiesMap();
        ArrayList<Integer> keys=new ArrayList<Integer>(citiesMap.keySet());
        for(int i=0;i<numOfCities;i++){
            int x = keys.get(i);
            for(int j=0;j<numOfCities;j++){
                int y = keys.get(j);
                citiesMatrix[i][j] = allCitiesMatrix[x][y];
            }
        }
        return citiesMatrix;
    }
}

//FOR TESTING
public double[][] generateRandomMatrix(int n) {
    double[][] randomMatrix = new double[n][n];
    IntStream.range(0, n)
        .forEach(i -> IntStream.range(0, n)
            .forEach(j -> randomMatrix[i][j] = Math.abs(random.nextInt(100) + 1)));
    return randomMatrix;
}
```

```
public void optimize(){
    IntStream.rangeClosed(1, 3)
        .forEach(i -> {
            //TO DO GUI
            System.out.println("Attempt #" + i);
            solve();
        });
}

 public int[] solve(){
     setupAnts();
     clearRoutes();
     IntStream.range(0,iterations).forEach(i->{
         moveAnts();
         updateRoutes();
         updateBest();
     });
     FXMLController.setLengthResult(String.valueOf(bestLength-numOfCities));
     FXMLController.setOrderResult(String.valueOf(Arrays.toString(bestOrder)));
     return bestOrder.clone();
 }

 public void setupAnts(){
     ants.stream().
             forEach(ant -> {
                 ant.clear();
                 ant.visitCity(-1, random.nextInt(numOfCities));
             });
     currentIdx=0;
 }

 public void moveAnts(){
     IntStream.range(currentIdx,numOfCities -1).
             forEach(i->{
                 ants.forEach(ant-> ant.visitCity(currentIdx, selectNextCity(ant)));
                 currentIdx++;
             });
 }

public int selectNextCity(Ant ant){
    int randInt = random.nextInt(numOfCities - currentIdx);
    if(random.nextDouble() < randomFactor){
        OptionalInt cityIdx = IntStream.range(0,numOfCities).filter(i->i==randInt && !ant.getVisitedByIdx(i)).findFirst();
        if(cityIdx.isPresent()){
            return cityIdx.getAsInt();
        }
    }
    calcProbabilty(ant);
    double randDouble = random.nextDouble();
    double total =0;
    for(int i=0;i<numOfCities;i++){
        total +=probabilities[i];
        if(total >= randDouble){
            return i;
        }
    }
    return -1;
}
```

```
    public void calcProbabilty(Ant ant){
        int i = ant.getRouteByIdx(currentIdx);
        double pheromone = 0.0;
        for (int j=0;j<numOfCities;j++){
            if(!ant.getVisitedByIdx(j)){
                pheromone += Math.pow(routes[i][j], alpha) * Math.pow(1.0/graph[i][j], beta);
            }
        }
        for(int k=0;k<numOfCities;k++){
            if(ant.getVisitedByIdx(k)){
                probabilities[k]=0.0;
            }else{
                double numerator = Math.pow(routes[i][k], alpha) * Math.pow(1.0 / graph[i][k], beta);
                probabilities[k] = numerator/pheromone;
            }
        }
    }

    public void updateRoutes(){
        for (int i = 0; i < numOfCities; i++) {
            for (int j = 0; j < numOfCities; j++) {
                routes[i][j] *= evaporation;
            }
        }
        ants.forEach((ant) -> {
            double contribution = Q/ant.routeLength(graph);
            for(int i=0;i<numOfCities-1;i++){
                routes[ant.getRouteByIdx(i)][ant.getRouteByIdx(+1)] += contribution;
            }
            routes[ant.getRouteByIdx(numOfCities-1)][ant.getRouteByIdx(0)] += contribution;
        });
    }

    public void updateBest(){
        if(bestOrder == null){
            bestOrder=ants.get(0).getRoute();
            bestLength=ants.get(0).routeLength(graph);
        }
        for(Ant ant:ants){
            if(ant.routeLength(graph)<bestLength){
                bestLength = ant.routeLength(graph);
                bestOrder = ant.getRoute().clone();
            }
        }
    }

    public void clearRoutes(){
        IntStream.range(0, numOfCities)
            .forEach(i -> {
                IntStream.range(0, numOfCities)
                    .forEach(j -> routes[i][j] = c);
            });
    }
}
```

Klasa ACO, jest klasą implementująca algorytm mrówkowy. 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

```
public class FXMLController implements Initializable {

    @FXML
    private ComboBox<String> cityComboBox;
    @FXML
    private Button addCityButton;
    @FXML
    private Button startACO;
    @FXML
    private Button addAllCitiesButton;
    @FXML
    private Button removeCityButton;
    @FXML
    private Button removeAllCitiesButton;
    @FXML
    private Label addInfoLabel;
    @FXML
    private Label bestLen;
    @FXML
    private Label bestOrder;
    @FXML
    private ListView<String> citiesListView;
    @FXML
    private ListView<String> finalOrderListView;
    @FXML
    private TextField colonySize;
    @FXML
    private TextField iterations;
    @FXML
    private TextField alpha;
    @FXML
    private TextField beta;
    @FXML
    private TextField evaporation;
    @FXML
    private TextField qVal;


    private BidiMap<Integer, String> citiesMap = new DualHashBidiMap<>();
    private static BidiMap<Integer, String> selectedCitiesMap = new DualHashBidiMap<>();

    private static ObservableList<String> cities = FXCollections.observableArrayList();
    private static ObservableList<String> orderedCities=FXCollections.observableArrayList();

    private static String lengthResult;
    private static String orderResult;



   public static void setLengthResult(String res) {
       lengthResult = res;
   }

   public static void setOrderResult(String res) {
       orderResult = res;
   }
```

```java
public static BidiMap getSelectedCitiesMap() {
    return selectedCitiesMap;
}

@FXML
void addCityButton(ActionEvent event) {
    if (cityComboBox.valueProperty().getValue() != null) {
        if (!cities.contains(cityComboBox.getValue())) {
            cities.add(cityComboBox.getValue());
            selectedCitiesMap.put(citiesMap.getKey(cityComboBox.getValue()), cityComboBox.getValue());
            addInfoLabel.setText("city added");
        } else {
            addInfoLabel.setText("city already added");
        }
    } else {
        addInfoLabel.setText("select correct city");
    }
    refresh();
}

@FXML
void addAllCitiesButton(ActionEvent event) {
    if (!cities.isEmpty() || !selectedCitiesMap.isEmpty()) {
        cities.clear();
        selectedCitiesMap.clear();
    }
    for (int i = 0; i <= 19; i++) {
        cities.add((String) citiesMap.get(i));
        selectedCitiesMap.put(citiesMap.getKey((String) citiesMap.get(i)), (String) citiesMap.get(i));
    }
    addInfoLabel.setText("all cities added");
    refresh();
}


@FXML
void removeCityButton(ActionEvent event) {
    if (citiesListView.getSelectionModel().getSelectedItem() != null) {
        String cityToRemove = citiesListView.getSelectionModel().getSelectedItem();
        cities.remove(cityToRemove);
        selectedCitiesMap.values().removeIf(value -> value.contains(cityToRemove));
        //selectedCitiesMap.remove(cityToRemove);
        System.out.println(selectedCitiesMap.toString());
        addInfoLabel.setText("city removed");
    } else {
        addInfoLabel.setText("Nothing to remove");
    }
    refresh();
}
```

```
@FXML
void removeAllCitiesButton(ActionEvent event) {
    if (!cities.isEmpty() || !selectedCitiesMap.isEmpty()) {
        cities.clear();
        selectedCitiesMap.clear();
        System.out.println(selectedCitiesMap.toString());
        addInfoLabel.setText("All citites removed");
    }else{
        addInfoLabel.setText("Nothing to remove");
    }
    refresh();
}

@FXML
void startAcoButton(ActionEvent event) {
    ACO antColony = new ACO(cities.size(), Integer.parseInt(colonySize.getText()));
    antColony.setIterations(Integer.parseInt(iterations.getText()));
    antColony.setAlpha(Double.parseDouble(alpha.getText()));
    antColony.setBeta(Double.parseDouble(beta.getText()));
    antColony.setEvaporation(Double.parseDouble(evaporation.getText()));
    antColony.setQ(Double.parseDouble(qVal.getText()));
    antColony.optimize(); //TO DO testing
    bestLen.setText(lengthResult);
    bestOrder.setText(orderResult);
    int[] bestOrder=antColony.getBestOrder();
    orderedCities.clear();
    //System.out.println(Arrays.toString(xd));
    //System.out.println(cities.get(0)); //printuje 1 item z tabeli pierwszej
    for(int i=0;i<cities.size();i++){
        int firstElementOfArray=(int)Array.get(bestOrder,i);
        String cityInFirstTable=cities.get(firstElementOfArray);
        orderedCities.add(cityInFirstTable);
    }
    refresh();
}

void refresh() {
    citiesListView.setItems(cities);
    finalOrderListView.setItems(orderedCities);
}

public static ObservableList<String> getObservableList1() {
    return cities;
}
```

```
    public void insertCitiesMap() {
        citiesMap.put(0, "KRAKOW");
        citiesMap.put(1, "WARSZAWA");
        citiesMap.put(2, "WROCLAW");
        citiesMap.put(3, "GDANSK");
        citiesMap.put(4, "BYDGOSZCZ");
        citiesMap.put(5, "POZNAN");
        citiesMap.put(6, "KATOWICE");
        citiesMap.put(7, "BIALYSTOK");
        citiesMap.put(8, "LODZ");
        citiesMap.put(9, "SANDOMIERZ");
        citiesMap.put(10, "KAZIMIERZ DOLNY");
        citiesMap.put(11, "RESZEL");
        citiesMap.put(12, "MIELNO");
        citiesMap.put(13, "KARPACZ");
        citiesMap.put(14, "OLKUSZ");
        citiesMap.put(15, "KRYNICA ZDROJ");
        citiesMap.put(16, "HEL");
        citiesMap.put(17, "ZAMOSC");
        citiesMap.put(18, "ZAKOPANE");
        citiesMap.put(19, "SZCZECIN");
    }

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        insertCitiesMap();
        cityComboBox.setItems(FXCollections.observableArrayList(citiesMap.values()));
    }
}
```

Klasa FXMLController jest klasą odpowiedzialną 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 stworzenia algorytmu.

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

4. 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

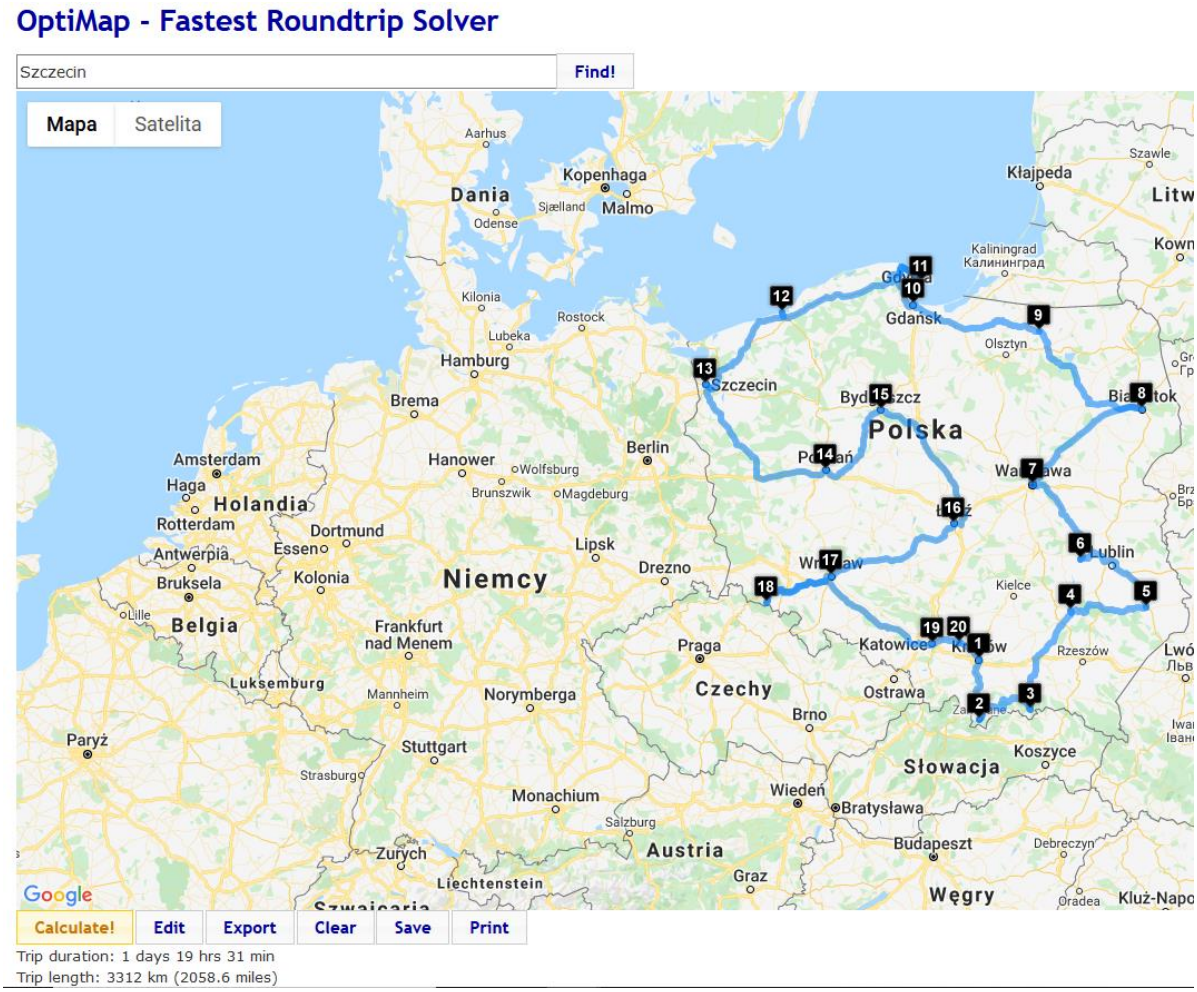
Badania dotyczyły sprawności algorytmu dla danych parametrów testowych przy obliczeniu trasy po wszystkich aktualnych 20 miastach Polski i ich macierzy odległości opisanej w wstępie teoretycznym.

L.p	Alfa	Beta	Liczba mrówek	Ilość iteracji	Współczynnik parowania	Parametr Q	Najkrótsza długość trasy
1	1.0	1.0	30	100	0.5	500	**4759**
2	1.0	1.0	30	200	0.5	500	**4794**
3	1.0	1.0	30	500	0.5	500	**4902**
4	1.0	3.0	30	100	0.5	500	**3515**
5	1.0	3.0	30	200	0.5	500	**3545**
6	1.0	3.0	30	500	0.5	500	**3567**
7	3.0	9.0	30	100	0.5	500	**3349**
8	3.0	9.0	30	200	0.5	500	**3343**
9	3.0	9.0	30	500	0.5	500	**3349**
10	9.0	12.0	30	100	0.5	500	**3349**
11	9.0	12.0	30	200	0.5	500	**3301**
12	9.0	12.0	30	500	0.5	500	**3349**
13	1.0	1.0	50	100	0.5	500	**4905**
14	1.0	1.0	50	100	0.3	500	**4985**
15	1.0	1.0	50	200	0.3	500	**4241**
16	1.0	3.0	50	100	0.5	500	**3592**
17	1.0	3.0	50	100	0.3	500	**3508**
18	1.0	3.0	50	200	0.3	500	**3442**
19	3.0	9.0	50	100	0.5	500	**3327**
20	3.0	9.0	50	100	0.3	500	**3327**
21	3.0	9.0	50	200	0.3	500	**3323**
22	9.0	12.0	50	100	0.5	500	**3349**
23	9.0	12.0	50	100	0.3	500	**3325**
24	9.0	12.0	50	200	0.3	500	**3349**
25	1.0	1.0	200	100	0.3	500	**4338**
26	1.0	1.0	200	500	0.3	500	**4499**
27	1.0	1.0	200	1000	0.3	500	**3899**
28	1.0	3.0	200	100	0.3	500	**3405**
29	1.0	3.0	200	500	0.3	500	**3421**
30	1.0	3.0	200	1000	0.3	500	**3418**
31	3.0	9.0	200	100	0.3	500	**3327**
32	3.0	9.0	200	500	0.3	500	**3301**
33	3.0	9.0	200	1000	0.3	500	**3301**
34	9.0	12.0	200	100	0.3	500	**3327**
35	9.0	12.0	200	500	0.3	500	**3327**
36	9.0	12.0	200	1000	0.3	500	**3327**

Wyniki prób dla powyższych rekordów:

L.p	Próba 1	Próba 2	Próba 3	Próba 4	Próba 5	Próba 6	Próba 7	Próba 8	Próba 9	Próba 10
1	5449	5336	5480	5102	4759	5518	4818	5232	5461	5269
2	4860	5310	5458	5771	5455	4892	5364	5770	4839	4794
3	5650	5105	5324	5557	4902	4944	5421	5369	4985	5743
4	4029	3653	3515	3728	3572	3909	3655	3697	3771	4036
5	3911	3867	3860	3780	3757	3627	3545	3979	3919	3949
6	3859	3855	3779	3717	3767	3648	3862	3567	4085	3747
7	3349	3349	3349	3406	3361	3349	3349	3349	3399	3415
8	3349	3459	3349	3349	3349	3349	3349	3406	3343	3349
9	3417	3349	3447	3391	3417	3349	3349	3349	3349	3417
10	3349	3349	3349	3349	3349	3383	3349	3448	3349	3365
11	3383	3349	3349	3448	3301	3373	3349	3343	3343	3391
12	3459	3349	3349	3349	3349	3349	3349	3349	3349	3349
13	5385	5032	4931	5217	5549	5008	4905	5368	5268	5015
14	5010	5338	5263	5153	5049	4985	5140	5525	5151	5066
15	4541	5085	5112	4241	4599	4785	5475	4865	4937	5209
16	3609	4003	3935	3744	3855	3592	3636	3664	3724	3822
17	3540	3736	3628	3608	3826	3734	3728	3851	3896	3508
18	3756	3768	3684	3442	3477	3688	3876	3842	3847	3697
19	3327	3369	3373	3459	3349	3349	3349	3339	3349	3349
20	3343	3349	3327	3383	3349	3383	3383	3349	3365	3343
21	3323	3349	3433	3349	3349	3413	3391	3365	3406	3349
22	3349	3349	3349	3349	3349	3349	3349	3349	3349	3349
23	3349	3349	3349	3349	3349	3349	3349	3349	3349	3325
24	3349	3349	3349	3349	3349	3349	3349	3413	3349	3349
25	4924	4957	4710	4844	4942	4725	5209	4338	4708	4856
26	4623	4499	5081	4882	4883	4921	4523	4926	4500	5228
27	4330	4951	4984	4952	3899	4794	4894	4982	4897	4286
28	3676	3664	3737	3840	3405	3615	3646	3627	3495	3506
29	3471	3539	3448	3421	3526	3689	3524	3602	3489	3654
30	3576	3724	3653	3418	3614	3698	3467	3588	3646	3640
31	3343	3327	3349	3343	3349	3349	3349	3349	3349	3349
32	3349	3327	3349	3349	3343	3339	3301	3325	3327	3317
33	3349	3349	3349	3343	3301	3327	3339	3349	3301	3349
34	3349	3349	3343	3349	3349	3349	3327	3339	3349	3349
35	3327	3349	3349	3349	3327	3349	3327	3349	3327	3339
36	3349	3349	3349	3349	3327	3349	3349	3349	3327	3327

5. Podsumowanie/ wnioski



➔ Porównaliśmy naszą implementację algorytmu z istniejącą implementacją lekko zmodyfikowanego algorytmu kolonii mrówek. Uzyskany tam wynik dla 20 miast, wynosi 3312 km, a przy tej implementacji wynik ten wynosi 3301 km.

➔ Poprzez próbę badawczą udowodniono, że program działa prawidłowo, a nawet jest dokładniejszy od istniejących implementacji.

➔ Współczynnik alfa odpowiada za eksploatacje[1] znanej trasy natomiast współczynnik beta odpowiada za eksplorację[2].

➔ Wejściowy parametr c odpowiada za oryginalną liczbę śladów na początku symulacji.

➔ Bazując na wykonanych badaniach największy wpływ na wynik algorytmu mają parametry alfa oraz beta, ponieważ najszybciej zbliżają do dobrego wyniku. Inne parametry, t.j.: zwiększona iteracja lub większa kolonia mrówek obciążą bardziej sprzęt, na którym są wykonywane badania niżeli poprawią wynik.

➔ Najistotniejszą częścią jest właściwy wybór następnego miasta do odwiedzenia. Należy pamiętać o tym, że każda mrówka stara się podążać śladem poprzednich.

[1] Eksploatacja- korzystanie z obecnej wiedzy lub zasobów,aby zdołać jak najdokładniej znaleźć optimum.
[2] Eksploracja- przeszukiwanie zgrubne całego obszaru, próba znalezienia optimów lokalnych

Miejsce to powinno zostać wybrane na podstawie logiki prawdopodobieństwa pamiętając, że mrówki preferują podążać trasami krótkimi o dużym nasileniu feromonu.

➔ Techniki sztucznej inteligencji rozwijają nowe metody rozwiązujące nierozwiązywalne problemy dawnych czasów oraz obecnych.

6. 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 projektu
  https://github.com/MCtyskie/AIProject