Прізвище: Магеровський

Ім’я: Андрій

Група: КНМ-14

Дисципліна: Методи нечіткої логіки та еволюційні алгоритми при автоматизованому проектуванні

Перевірив: Кривий Р.З.

**Звіт до лабораторної роботи № 5**

**«Рішення задачі комівояжера за допомогою генетичного алгоритму»**

**МЕТА РОБОТИ**

Ознайомитися з основними теоретичними відомостями про задачу комівояжера, обрати середовище розробки та мову програмування, реалізувати вирішення задачі комівояжера для [10; 50] міст за допомогою генетичного алгоритму.

**КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ**

Задача комівояжераполягає у знаходженні найвигіднішого маршруту, що проходить через вказані міста хоча б по одному разу. В умовах завдання вказуються критерій вигідності маршруту (найкоротший, найдешевший, сукупний критерій тощо) і відповідні матриці відстаней, вартості тощо. Зазвичай задано, що маршрут повинен проходити через кожне місто тільки один раз, в такому випадку розв'язок знаходиться серед гамільтонових циклів.

Існує маса різновидів узагальненої постановки задачі, зокрема геометрична задача комівояжера (коли матриця відстаней відображає відстані між точками на площині), трикутна задача комівояжера (коли на матриці вартостей виконується нерівність трикутника), симетрична та асиметрична задачі комівояжера.   
  
Прості методи розв'язання задачі комівояжера: повний лексичний перебір, жадібні алгоритми (метод найближчого сусіда), метод включення найближчого міста, метод найдешевшого включення, метод мінімального кістяка дерева. На практиці застосовують різні модифікації ефективніших методів: метод гілок і меж і метод генетичних алгоритмів, а так само алгоритм мурашиної колонії. 

**ІНДИВІДУАЛЬНЕ ЗАВДАННЯ**

За допомогою засобів мови програмування C# розробити програмне забезпечення для розв’язку задачі комівожера.

**Код програми:**

**CityHelper.cs**

using System;

using System.Collections.Generic;

using SalesmanGA.Models;

namespace SalesmanGA.Helpers

{

/// <summary>

/// CityHelper class

/// </summary>

public static class CityHelper

{

/// <summary>

/// Calculates the city distances.

/// </summary>

/// <param name="cityList">The city list.</param>

/// <param name="numberOfCloseCities">The number of close cities.</param>

public static void CalculateCityDistances(IList<City> cityList, int numberOfCloseCities)

{

foreach (City city in cityList)

{

city.Distances.Clear();

for (int i = 0; i < cityList.Count; i++)

{

city.Distances.Add(Math.Sqrt(Math.Pow(city.Location.X - cityList[i].Location.X, 2D) +

Math.Pow(city.Location.Y - cityList[i].Location.Y, 2D)));

}

}

foreach (City city in cityList)

{

city.FindClosestCities(numberOfCloseCities);

}

}

}

}

**RouteHelper.cs**

using System;

using System.Collections.Generic;

using SalesmanGA.Models;

namespace SalesmanGA.Helpers

{

/// <summary>

/// RouteHelper class

/// </summary>

public static class RouteHelper

{

#region Public properties

/// <summary>

/// Gets or sets the best route.

/// </summary>

public static Route BestRoute { get; set; }

#endregion

#region Public methods

/// <summary>

/// Create the initial set of random tours.

/// </summary>

/// <param name="populationSize">Number of tours to create.</param>

/// <param name="cityList">The list of cities in this tour.</param>

/// <param name="rand">Random number generator. We pass around the same random number generator, so that results between runs are consistent.</param>

/// <param name="chanceToUseCloseCity">The odds (out of 100) that a city that is known to be close will be used in any given link.</param>

public static List<Route> CreateRandomPopulation(int populationSize, List<City> cityList, Random rand, int chanceToUseCloseCity)

{

List<Route> population = new List<Route>();

for (int tourCount = 0; tourCount < populationSize; tourCount++)

{

Route route = new Route(cityList.Count);

// Create a starting point for this tour

var firstCity = rand.Next(cityList.Count);

var lastCity = firstCity;

for (int city = 0; city < cityList.Count - 1; city++)

{

int nextCity;

do

{

// Keep picking random cities for the next city, until we find one we haven't been to.

if ((rand.Next(100) < chanceToUseCloseCity) && (cityList[city].CloseCities.Count > 0))

{

// 75% chance will will pick a city that is close to this one

nextCity = cityList[city].CloseCities[rand.Next(cityList[city].CloseCities.Count)];

}

else

{

// Otherwise, pick a completely random city.

nextCity = rand.Next(cityList.Count);

}

// Make sure we haven't been here, and make sure it isn't where we are at now.

} while ((route[nextCity].Connection2 != -1) || (nextCity == lastCity));

// When going from city A to B, [1] on A = B and [1] on city B = A

route[lastCity].Connection2 = nextCity;

route[nextCity].Connection1 = lastCity;

lastCity = nextCity;

}

// Connect the last 2 cities.

route[lastCity].Connection2 = firstCity;

route[firstCity].Connection1 = lastCity;

route.DetermineFitness(cityList);

population.Add(route);

if ((BestRoute == null) || (route.Fitness < BestRoute.Fitness))

{

BestRoute = route;

}

}

return population;

}

#endregion

}

}

**SailsmanBrains.cs**

using System;

using System.Collections.Generic;

using SalesmanGA.Event\_Args;

using SalesmanGA.Helpers;

using SalesmanGA.Models;

namespace SalesmanGA

{

/// <summary>

/// This class performs the Travelling Salesman Problem algorithm.

/// </summary>

public class SalesmanBrains

{

#region Delegates

/// <summary>

/// Delegate used to raise an event when a new best tour is found.

/// </summary>

/// <param name="sender">Object that generated this event.</param>

/// <param name="e">Event arguments. Contains information about the best tour.</param>

public delegate void NewBestTourEventHandler(Object sender, SalesmanEventArgs e);

#endregion

#region Events

/// <summary>

/// Event fired when a new best tour is found.

/// </summary>

public event NewBestTourEventHandler FoundNewBestRoute;

#endregion

#region Public properties

/// <summary>

/// Random number generator object.

/// We allow the GUI to set the seed for the random number generator to assist in debugging.

/// This allows errors to be easily reproduced.

/// </summary>

public Random Rand { get; set; }

/// <summary>

/// The list of cities. This is only used to calculate the distances between the cities.

/// </summary>

public List<City> CityList { get; set; }

/// <summary>

/// The complete list of all the tours.

/// </summary>

public List<Route> Population1 { get; set; }

/// <summary>

/// The GUI sets this flag to true to stop the TSP algorithm and allow the Begin() function to return.

/// </summary>

public bool Halt { get; set; }

#endregion

#region Public methods

/// <summary>

/// Starts the TSP algorithm.

/// To stop before all generations are calculated, set <see cref="Halt"/> to true.

/// </summary>

/// <param name="populationSize">Number of random tours to create before starting the algorithm.</param>

/// <param name="maxGenerations">Number of times to perform the crossover operation before stopping.</param>

/// <param name="groupSize">Number of tours to examine in each generation. Top 2 are chosen as the parent tours whose children replace the worst 2 tours in the group.</param>

/// <param name="mutation">Odds that a child tour will be mutated..</param>

/// <param name="seed">Seed for the random number generator.</param>

/// <param name="chanceToUseCloseCity">The odds (out of 100) that a city that is known to be close will be used in any given link.</param>

/// <param name="cityList">List of cities in the tour.</param>

public void Begin(int populationSize, int maxGenerations, int groupSize, int mutation, int seed, int chanceToUseCloseCity, List<City> cityList)

{

Rand = new Random(seed);

CityList = cityList;

Population1 = RouteHelper.CreateRandomPopulation(populationSize, cityList, Rand, chanceToUseCloseCity);

DisplayRoute(RouteHelper.BestRoute, 0, false);

int generation;

for (generation = 0; generation < maxGenerations; generation++)

{

if (Halt)

{

break; // GUI has requested we exit.

}

var foundNewBestTour = MakeChildren(groupSize, mutation);

if (foundNewBestTour)

{

DisplayRoute(RouteHelper.BestRoute, generation, false);

}

}

DisplayRoute(RouteHelper.BestRoute, generation, true);

}

/// <summary>

/// Randomly select a group of tours from the population.

/// The top 2 are chosen as the parent tours.

/// Crossover is performed on these 2 tours.

/// The childred tours from this process replace the worst 2 tours in the group.

/// </summary>

/// <param name="groupSize">Number of tours in this group.</param>

/// <param name="mutation">Odds that a child will be mutated.</param>

public bool MakeChildren(int groupSize, int mutation)

{

int[] tourGroup = new int[groupSize];

int tourCount;

// pick random tours to be in the neighborhood city group

// we allow for the same tour to be included twice

for (tourCount = 0; tourCount < groupSize; tourCount++)

{

tourGroup[tourCount] = Rand.Next(Population1.Count);

}

// bubble sort on the neighborhood city group

for (tourCount = 0; tourCount < groupSize - 1; tourCount++)

{

var topTour = tourCount;

for (int i = topTour + 1; i < groupSize; i++)

{

if (Population1[tourGroup[i]].Fitness < Population1[tourGroup[topTour]].Fitness)

{

topTour = i;

}

}

if (topTour != tourCount)

{

var tempTour = tourGroup[tourCount];

tourGroup[tourCount] = tourGroup[topTour];

tourGroup[topTour] = tempTour;

}

}

bool foundNewBestTour = false;

// take the best 2 tours, do crossover, and replace the worst tour with it

var childPosition = tourGroup[groupSize - 1];

Population1[childPosition] = Route.Crossover(Population1[tourGroup[0]], Population1[tourGroup[1]], CityList, Rand);

if (Rand.Next(100) < mutation)

{

Population1[childPosition].Mutate(Rand);

}

Population1[childPosition].DetermineFitness(CityList);

// now see if the first new tour has the best fitness

if (Population1[childPosition].Fitness < RouteHelper.BestRoute.Fitness)

{

RouteHelper.BestRoute = Population1[childPosition];

foundNewBestTour = true;

}

// take the best 2 tours (opposite order), do crossover, and replace the 2nd worst tour with it

childPosition = tourGroup[groupSize - 2];

Population1[childPosition] = Route.Crossover(Population1[tourGroup[1]], Population1[tourGroup[0]], CityList, Rand);

if (Rand.Next(100) < mutation)

{

Population1[childPosition].Mutate(Rand);

}

Population1[childPosition].DetermineFitness(CityList);

// now see if the second new tour has the best fitness

if (Population1[childPosition].Fitness < RouteHelper.BestRoute.Fitness)

{

RouteHelper.BestRoute = Population1[childPosition];

foundNewBestTour = true;

}

return foundNewBestTour;

}

#endregion

#region Private methods

/// <summary>

/// Raise an event to the GUI listener to display a tour.

/// </summary>

/// <param name="bestRoute">The best tour the algorithm has found so far.</param>

/// <param name="generationNumber">How many generations have been performed.</param>

/// <param name="complete">Is the TSP algorithm complete.</param>

private void DisplayRoute(Route bestRoute, int generationNumber, bool complete)

{

FoundNewBestRoute?.Invoke(this, new SalesmanEventArgs(CityList, bestRoute, generationNumber, complete));

}

#endregion

}

}

**Отримані результати:**

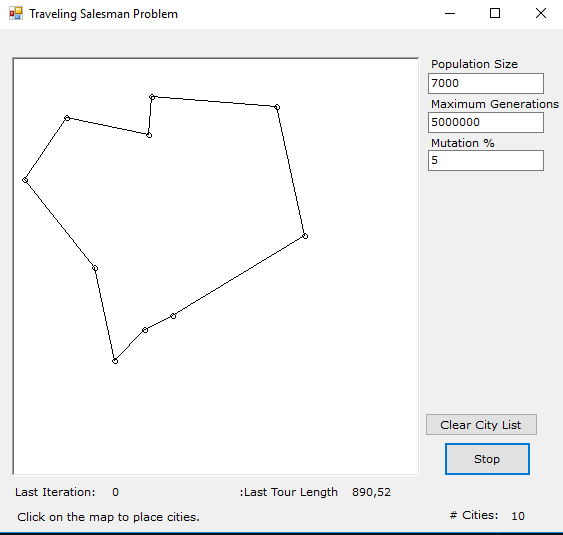


Рис 1. Розв`язок задачі при наявності десяти міст

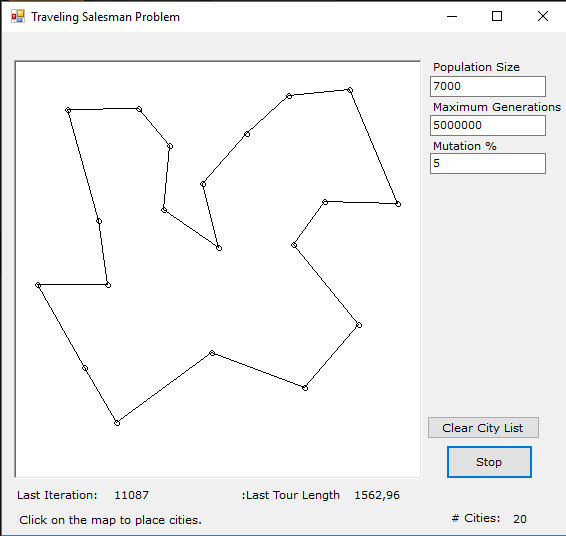


Рис 2. Розв`язок задачі при наявності двадцяти міст

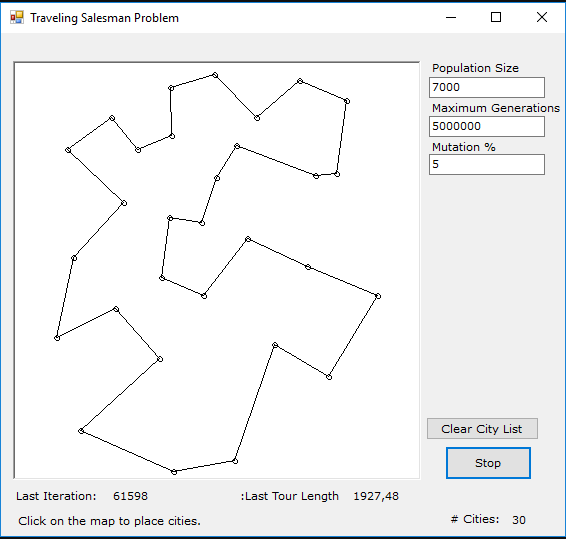


Рис 3. Розв`язок задачі при наявності тридцяти міст.

**Висновок:** виконуючи дану лабораторну роботу я ознайомився із теоретичними відомостями, а також, використовуючи засоби мови програмування C# та середовища розробки Visual Studio, розробив програмне забезпечення для розв’язку задачі комівояжера за допомогою генетичного алгоритму.