**Bio-Inspired Optimization Algorithm**

**Ant Colony Algorithm:**

The Traveling Salesman Problem is a problem of a salesman who, starting from his hometown, wants to find the shortest tour that takes him through a given set of customer cities and then back home, visiting each customer city exactly once." Each city is accessible from all other cities. Use ant colony algorithm for generating good solutions to both symmetric and asymmetric instances of the Traveling Salesman Problem. Use appropriate representation for graph and an appropriate heuristic that defines the distance between any two nodes of the graph. Use parallel approach to optimize solution

**Source Code:**

using System;

using System.Collections.Generic;

namespace AntColonySystem

{

class Program

{

static void Main(string[] args)

{

List<Point> points = TspFileReader.ReadTspFile(@"kroA100.tsp"); // Parse TSPlib file and load as List<Point>

Graph graph = new Graph(points, true); // Create Graph

GreedyAlgorithm greedyAlgorithm = new GreedyAlgorithm(graph);

double greedyShortestTourDistance = greedyAlgorithm.Run(); // get shortest tour using greedy algorithm

Parameters parameters = new Parameters() // Most parameters will be default. We only have to set T0 (initial pheromone level)

{

T0 = (1.0 / (graph.Dimensions \* greedyShortestTourDistance))

};

parameters.Show();

Solver solver = new Solver(parameters, graph);

List<double> results = solver.RunACS(); // Run ACS

Console.WriteLine("Time: " + solver.GetExecutionTime());

Console.ReadLine();

}

}

}

using System;

namespace AntColonySystem

{

public class Edge

{

public Point Start { get; set; }

public Point End { get; set; }

public double Length { get; set; }

public double Pheromone { get; set; }

public double Weight { get; set; }

public Edge() { }

public Edge(Point start, Point end)

{

Start = start;

End = end;

Length = Math.Round(Start.DistanceTo(End));

}

}

}

using System;

using System.Collections.Generic;

namespace AntColonySystem

{

public class Graph

{

public List<Point> Points { get; set; }

public Dictionary<int, Edge> Edges { get; set; }

public int Dimensions { get; set; }

public double MinimumPheromone { get; set; }

private bool IsSymetric { get; set; }

public Graph(List<Point> Points, bool isSymetric)

{

Edges = new Dictionary<int, Edge>();

this.Points = Points;

Dimensions = Points.Count;

IsSymetric = isSymetric;

CreateEdges();

}

/// <summary>

/// Create edges between all points.

/// NOTE: For every two points there is two edges between them in case of asymetric problem (1 -> 2, 2 -> 1).

/// </summary>

private void CreateEdges()

{

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

{

for (int j = 0; j < Points.Count; j++)

{

if (i != j)

{

Edge edge = new Edge(Points[i], Points[j]);

Edges.Add(Helper.HashFunction(Points[i].Id, Points[j].Id), edge);

}

}

}

}

/// <summary>

/// Return edge beetwen two points (their ID's) from Dictionary

/// </summary>

public Edge GetEdge(int firstPointId, int secondPointId)

{

return Edges[Helper.HashFunction(firstPointId, secondPointId)];

}

/// <summary>

/// Set specific pheromone to all edges

/// </summary>

public void ResetPheromone(double pheromoneValue)

{

foreach (var edge in Edges)

{

edge.Value.Pheromone = pheromoneValue;

}

}

public void EvaporatePheromone(Edge edge, double value)

{

edge.Pheromone = Math.Max(MinimumPheromone, edge.Pheromone \* value); // Math.Max is here to prevent Pheromon = 0

if (IsSymetric)

{

var secondEdge = GetEdge(edge.End.Id, edge.Start.Id);

secondEdge.Pheromone = Math.Max(MinimumPheromone, secondEdge.Pheromone \* value);

}

}

public void DepositPheromone(Edge edge, double value)

{

edge.Pheromone += value;

if (IsSymetric)

{

var secondEdge = GetEdge(edge.End.Id, edge.Start.Id);

secondEdge.Pheromone += value;

}

}

}

}

namespace AntColonySystem

{

public struct Point

{

public float X;

public float Y;

public int Id;

public Point(int id, float x, float y)

{

X = x;

Y = y;

Id = id;

}

public double DistanceTo(Point anotherPoint)

{

return Distance.Euclidean(X, Y, anotherPoint.X, anotherPoint.Y);

}

}

}

using System.Collections.Generic;

using System.Linq;

namespace AntColonySystem

{

public class Ant

{

#region Properties

public Graph Graph { get; set; }

public int Beta { get; set; }

public double Q0 { get; set; }

public int StartNodeId { get; set; }

public double Distance { get; set; }

public List<Point> VisitedNodes { get; set; }

public List<Point> UnvisitedNodes { get; set; }

public List<Edge> Path { get; set; }

#endregion

public Ant(Graph graph, int beta, double q0)

{

Graph = graph;

Beta = beta;

Q0 = q0;

VisitedNodes = new List<Point>();

UnvisitedNodes = new List<Point>();

Path = new List<Edge>();

}

public void Init(int startNodeId)

{

StartNodeId = startNodeId;

Distance = 0;

VisitedNodes.Add(Graph.Points.Where(x => x.Id == startNodeId).First());

UnvisitedNodes = Graph.Points.Where(x => x.Id != startNodeId).ToList();

Path.Clear();

}

public Point CurrentNode()

{

return VisitedNodes[VisitedNodes.Count - 1];

}

public bool CanMove()

{

return VisitedNodes.Count != Path.Count;

}

public Edge Move()

{

Point endPoint;

var startPoint = CurrentNode();

if (UnvisitedNodes.Count == 0)

{

endPoint = VisitedNodes[0]; // if ant visited every node, just go back to start

}

else

{

endPoint = ChooseNextPoint();

VisitedNodes.Add(endPoint);

UnvisitedNodes.RemoveAt(UnvisitedNodes.FindIndex(x => x.Id == endPoint.Id));

}

var edge = Graph.GetEdge(startPoint.Id, endPoint.Id);

Path.Add(edge);

Distance += edge.Length;

return edge;

}

private Point ChooseNextPoint()

{

List<Edge> edgesWithWeight = new List<Edge>();

Edge bestEdge = new Edge();

int currentNodeId = CurrentNode().Id;

foreach (var node in UnvisitedNodes)

{

var edge = Graph.GetEdge(currentNodeId, node.Id);

edge.Weight = Weight(edge);

if (edge.Weight > bestEdge.Weight)

{

bestEdge = edge;

}

edgesWithWeight.Add(edge);

}

var random = RandomGenerator.Instance.Random.NextDouble();

if (random < Q0)

{

return Exploitation(bestEdge);

}

else

{

return Exploration(edgesWithWeight);

}

}

private double Weight(Edge edge)

{

double heuristic = 1 / edge.Length;

return edge.Pheromone \* Helper.Pow(heuristic, Beta);

}

private Point Exploitation(Edge bestEdge)

{

return bestEdge.End;

}

private Point Exploration(List<Edge> edgesWithWeight)

{

double totalSum = edgesWithWeight.Sum(x => x.Weight);

var edgeProbabilities = edgesWithWeight.Select(w => { w.Weight = (w.Weight / totalSum); return w; }).ToList();

var cumSum = Helper.EdgeCumulativeSum(edgeProbabilities);

Point choosenPoint = Helper.GetRandomEdge(cumSum);

return choosenPoint;

}

}

}

using System;

namespace AntColonySystem

{

public class Parameters

{

/// <summary>

/// Relative importance of distance(default=2)

/// </summary>

public int Beta { get; set; }

/// <summary>

/// Global evaporation rate of pheromone(0..1, default=0.1)

/// </summary>

public double GlobalEvaporationRate { get; set; }

/// <summary>

/// Local evaporation rate of pheromone(0..1, default=0.01)

/// </summary>

public double LocalEvaporationRate { get; set; }

/// <summary>

/// Probability of choosing best ant path instead of random roulette

/// </summary>

public double Q0 { get; set; }

/// <summary>

/// Initial pheromone level along each Edge

/// </summary>

public double T0 { get; set; }

/// <summary>

/// Define how many ants will be used

/// </summary>

public int AntCount { get; set; }

/// <summary>

/// Number of iterations to perform (default=2500)

/// </summary>

public int Iterations { get; set; }

public Parameters(int Beta, double GlobalEvaporationRate, double LocalEvaporationRate, double T0, double Q0, int AntCount, int Iterations)

{

this.Beta = Beta;

this.GlobalEvaporationRate = GlobalEvaporationRate;

this.LocalEvaporationRate = LocalEvaporationRate;

this.T0 = T0;

this.Q0 = Q0;

this.AntCount = AntCount;

this.Iterations = Iterations;

}

/// <summary>

/// Default parameters

/// </summary>

public Parameters()

{

Beta = 2;

GlobalEvaporationRate = 0.1;

LocalEvaporationRate = 0.01;

Q0 = 0.9;

AntCount = 20;

Iterations = 10000;

T0 = 0.01;

}

public void Show()

{

Console.WriteLine("Beta: " + Beta);

Console.WriteLine("Global Evaporation Rate: " + GlobalEvaporationRate);

Console.WriteLine("Local Evaporation Rate: " + LocalEvaporationRate);

Console.WriteLine("Q0: " + Q0);

Console.WriteLine("AntCount: " + AntCount);

Console.WriteLine("Iterations: " + Iterations);

Console.WriteLine("T0: " + T0);

Console.WriteLine();

}

}

}

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

namespace AntColonySystem

{

public class Solver

{

public Parameters Parameters { get; set; }

private Ant GlobalBestAnt { get; set; }

private List<double> Results { get; set; }

private Graph Graph { get; set; }

private Stopwatch Stopwatch { get; set; }

public Solver(Parameters parameters, Graph graph)

{

Parameters = parameters;

graph.MinimumPheromone = parameters.T0;

Graph = graph;

Results = new List<double>();

Stopwatch = new Stopwatch();

}

/// <summary>

/// Main loop of ACS algorithm

/// </summary>

public List<double> RunACS()

{

Stopwatch.Start();

Graph.ResetPheromone(Parameters.T0);

for (int i = 0; i < Parameters.Iterations; i++)

{

List<Ant> antColony = CreateAnts();

GlobalBestAnt = GlobalBestAnt ?? antColony[0];

Ant localBestAnt = BuildTours(antColony);

if (Math.Round(localBestAnt.Distance, 2) < Math.Round(GlobalBestAnt.Distance, 2))

{

GlobalBestAnt = localBestAnt;

Console.WriteLine("Current Global Best: " + GlobalBestAnt.Distance + " found in " + i + " iteration");

}

Results.Add(localBestAnt.Distance);

}

Stopwatch.Stop();

return Results;

}

/// <summary>

/// Create ants and place every ant in random point on graph (warning AntCount < Dimensions)

/// </summary>

public List<Ant> CreateAnts()

{

List<Ant> antColony = new List<Ant>();

List<int> randomPoints = RandomGenerator.GenerateRandom(Parameters.AntCount, 1, Graph.Points.Count);

foreach (int random in randomPoints)

{

Ant ant = new Ant(Graph, Parameters.Beta, Parameters.Q0);

ant.Init(random);

antColony.Add(ant);

}

return antColony;

}

/// <summary>

/// This method builds solution for every ant in AntColony and return the best ant (with shortest distance tour)

/// </summary>

public Ant BuildTours(List<Ant> antColony)

{

for (int i = 0; i < Graph.Dimensions; i++)

{

foreach (Ant ant in antColony)

{

Edge edge = ant.Move();

LocalUpdate(edge);

}

}

GlobalUpdate();

return antColony.OrderBy(x => x.Distance).FirstOrDefault(); // find shortest ant tour (path)

}

/// <summary>

/// Update pheromone level on edge passed in parameter

/// </summary>

public void LocalUpdate(Edge edge)

{

double evaporate = (1 - Parameters.LocalEvaporationRate);

Graph.EvaporatePheromone(edge, evaporate);

double deposit = Parameters.LocalEvaporationRate \* Parameters.T0;

Graph.DepositPheromone(edge, deposit);

}

/// <summary>

/// Update pheromone level on path for best ant

/// </summary>

public void GlobalUpdate()

{

double deltaR = 1 / GlobalBestAnt.Distance;

foreach (Edge edge in GlobalBestAnt.Path)

{

double evaporate = (1 - Parameters.GlobalEvaporationRate);

Graph.EvaporatePheromone(edge, evaporate);

double deposit = Parameters.GlobalEvaporationRate \* deltaR;

Graph.DepositPheromone(edge, deposit);

}

}

public TimeSpan GetExecutionTime()

{

return Stopwatch.Elapsed;

}

}

}

using System;

namespace AntColonySystem

{

public static class Distance

{

public static double Euclidean(double X1, double Y1, double X2, double Y2)

{

return Math.Sqrt((Math.Pow(X1 - X2, 2) + Math.Pow(Y1 - Y2, 2)));

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

namespace AntColonySystem

{

/// <summary>

/// This class represents simple greedy algorithm for finding shorters tour in TSP. It always choose shortest distance beetwen two points.

/// </summary>

public class GreedyAlgorithm

{

#region Properties

private Graph Graph { get; set; }

private double TourDistance { get; set; }

private List<Point> VisitedNodes { get; set; }

private List<Point> UnvisitedNodes { get; set; }

private List<Edge> Path { get; set; }

#endregion

public GreedyAlgorithm(Graph graph)

{

Graph = graph;

TourDistance = 0;

UnvisitedNodes = graph.Points.ToList();

VisitedNodes = new List<Point>();

Path = new List<Edge>();

}

public double Run()

{

Point toPoint, fromPoint;

SetStartingPoint();

for (int i = 0; i < Graph.Dimensions; i++)

{

fromPoint = GetCurrentNode();

if (UnvisitedNodes.Count > 0) // if we have nodes to visit;

{

toPoint = ChooseNextPoint(fromPoint);

VisitedNodes.Add(toPoint);

UnvisitedNodes.Remove(toPoint);

}

else

{

toPoint = VisitedNodes[0]; // if visited every node, just go back to start

}

Edge edge = Graph.GetEdge(fromPoint.Id, toPoint.Id);

Path.Add(edge);

TourDistance += edge.Length;

}

return Math.Round(TourDistance);

}

private Point ChooseNextPoint(Point startPoint)

{

List<Edge> edges = new List<Edge>();

foreach (Point endPoint in UnvisitedNodes)

{

Edge edge = Graph.GetEdge(startPoint.Id, endPoint.Id);

edges.Add(edge);

}

Edge shortestEdge = edges.OrderBy(x => x.Length).FirstOrDefault(); // choosing shortest edge

return shortestEdge.End;

}

private Point GetCurrentNode()

{

return VisitedNodes[VisitedNodes.Count - 1];

}

private void SetStartingPoint()

{

Point startPoint = Graph.Points.First(); // We can set any point as starting point. I choosed first one.

VisitedNodes.Add(startPoint);

UnvisitedNodes.Remove(startPoint);

}

}

}

using System.Collections.Generic;

using System.Linq;

namespace AntColonySystem

{

public static class Helper

{

public static IEnumerable<Edge> EdgeCumulativeSum(IEnumerable<Edge> sequence)

{

double sum = 0;

foreach (var item in sequence)

{

sum += item.Weight;

item.Weight = sum;

}

return sequence;

}

public static Point GetRandomEdge(IEnumerable<Edge> cumSum)

{

var random = RandomGenerator.Instance.Random.NextDouble();

return cumSum.First(j => j.Weight >= random).End;

}

/// <summary>

/// This is faster then standard Math.Pow() function, but can power numbers only by INT type.

/// </summary>

public static double Pow(double num, int exp)

{

double result = 1.0;

while (exp > 0)

{

if (exp % 2 == 1)

result \*= num;

exp >>= 1;

num \*= num;

}

return result;

}

/// <summary>

/// My custom hash function for Edges Dictionary

/// </summary>

public static int HashFunction(int x, int y)

{

return (10000000 \* x) + y;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

namespace AntColonySystem

{

public sealed class RandomGenerator

{

public static RandomGenerator Instance { get; } = new RandomGenerator();

public Random Random { get; set; }

private RandomGenerator() => Random = new Random();

public static double GetDoubleRangeRandomNumber(double minimum, double maximum)

{

return Instance.Random.NextDouble() \* (maximum - minimum) + minimum;

}

/// <summary>

/// Generate List of unique numbers

/// </summary>

public static List<int> GenerateRandom(int count, int min, int max)

{

return Enumerable.Range(min, max).OrderBy(x => Instance.Random.Next()).Take(count).ToList();

}

}

}

using System.Collections.Generic;

using System.Globalization;

using System.IO;

namespace AntColonySystem

{

public static class TspFileReader

{

public static List<Point> ReadTspFile(string tspFilePath)

{

var file = File.ReadLines(tspFilePath);

List<Point> points = new List<Point>();

CultureInfo ci = (CultureInfo)CultureInfo.CurrentCulture.Clone();

ci.NumberFormat.CurrencyDecimalSeparator = ".";

bool readData = false;

foreach (var item in file)

{

if (item.Contains("NODE\_COORD\_SECTION"))

{

readData = true;

continue;

}

if (item.Contains("EOF"))

{

readData = false;

}

if (readData)

{

var spitted = item.Split(' ');

points.Add(new Point(int.Parse(spitted[0]), float.Parse(spitted[1], NumberStyles.Any, ci), float.Parse(spitted[2], NumberStyles.Any, ci)));

}

}

return points;

}

}

}

**Input File: kroA100.tsp**

NAME: kroA100

TYPE: TSP

COMMENT: 100-city problem A (Krolak/Felts/Nelson)

DIMENSION: 100

EDGE\_WEIGHT\_TYPE : EUC\_2D

NODE\_COORD\_SECTION

1 1380 939

2 2848 96

3 3510 1671

4 457 334

5 3888 666

6 984 965

7 2721 1482

8 1286 525

9 2716 1432

10 738 1325

11 1251 1832

12 2728 1698

13 3815 169

14 3683 1533

15 1247 1945

16 123 862

17 1234 1946

18 252 1240

19 611 673

20 2576 1676

21 928 1700

22 53 857

23 1807 1711

24 274 1420

25 2574 946

26 178 24

27 2678 1825

28 1795 962

29 3384 1498

30 3520 1079

31 1256 61

32 1424 1728

33 3913 192

34 3085 1528

35 2573 1969

36 463 1670

37 3875 598

38 298 1513

39 3479 821

40 2542 236

41 3955 1743

42 1323 280

43 3447 1830

44 2936 337

45 1621 1830

46 3373 1646

47 1393 1368

48 3874 1318

49 938 955

50 3022 474

51 2482 1183

52 3854 923

53 376 825

54 2519 135

55 2945 1622

56 953 268

57 2628 1479

58 2097 981

59 890 1846

60 2139 1806

61 2421 1007

62 2290 1810

63 1115 1052

64 2588 302

65 327 265

66 241 341

67 1917 687

68 2991 792

69 2573 599

70 19 674

71 3911 1673

72 872 1559

73 2863 558

74 929 1766

75 839 620

76 3893 102

77 2178 1619

78 3822 899

79 378 1048

80 1178 100

81 2599 901

82 3416 143

83 2961 1605

84 611 1384

85 3113 885

86 2597 1830

87 2586 1286

88 161 906

89 1429 134

90 742 1025

91 1625 1651

92 1187 706

93 1787 1009

94 22 987

95 3640 43

96 3756 882

97 776 392

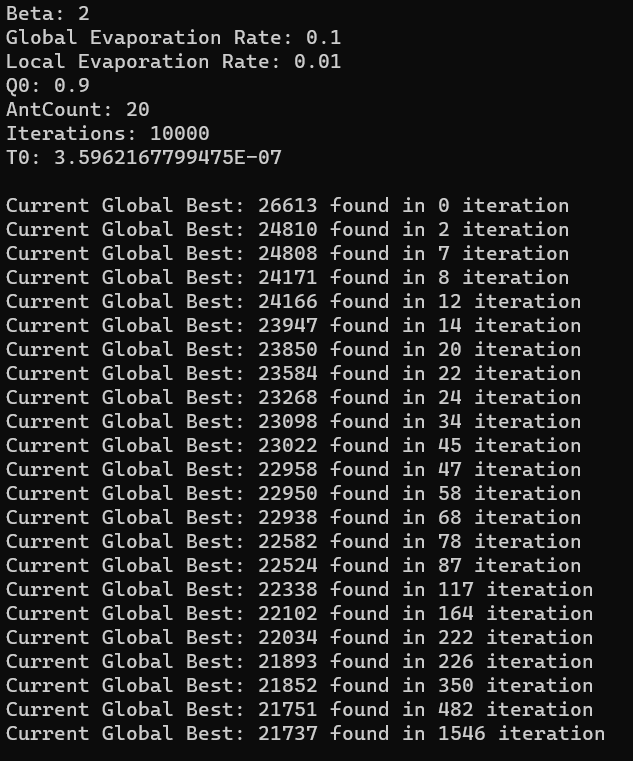
98 1724 1642

99 198 1810

100 3950 1558

EOF

**Output 1**

****

**Flowchart**: Ant Colony Algorithm

