**// Algorithm.cs**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

namespace Diploma

{

public abstract class Algorithm

{

protected List<Node> Nodes;

protected List<Node> Depots;

protected List<Node> Consumers;

protected int DepotsCount;

protected int ConsumersCount;

public int IterationNumber { get; private set; }

public abstract double Value { get; }

protected bool Stopped { get; private set; }

private const double Epsilon = 0.0001;

protected Algorithm()

{

IterationNumber = 0;

}

public void SetNodes(List<Node> nodesForSet)

{

Nodes = new List<Node>();

Depots = (from node in nodesForSet

where node.Type == Node.NodeType.Depot

select node).ToList();

Nodes.AddRange(Depots);

DepotsCount = Depots.Count;

Consumers = (from node in nodesForSet

where node.Type == Node.NodeType.Consumer

select node).ToList();

Nodes.AddRange(Consumers);

ConsumersCount = Consumers.Count;

}

public int LastChangedIteration { get; private set; }

public bool IsCalcLastChange { get; set; }

public string LogFileName { get; set; }

private StreamWriter writer = null;

private void OpenLogFile()

{

if (File.Exists(LogFileName))

{

writer = new StreamWriter(LogFileName, true);

writer.WriteLine(string.Format("Starting new algorithm: {0}. Time: {1}.", GetType().Name, DateTime.Now));

}

}

private double lastValue = double.PositiveInfinity;

private void Iteration()

{

if (Stopped)

return;

if (IterationNumber == 0)

{

OpenLogFile();

}

InnerIteration();

IterationNumber++;

double newValue = double.PositiveInfinity;

if (writer != null || IsCalcLastChange)

{

newValue = Value;

}

if (writer != null && !Stopped && (IterationNumber - 1) % 10 == 0)

{

writer.WriteLine(string.Format("{0}\t{1:0.00}", IterationNumber, newValue));

}

if (IsCalcLastChange && Math.Abs(lastValue - newValue) >= lastValue \* Epsilon)

{

LastChangedIteration = IterationNumber;

}

lastValue = newValue;

}

public void Iterations (int count = 1)

{

for (int i = 0; i != count; i++)

{

Iteration();

}

}

public void IterateToStop()

{

while (!Stopped)

{

Iteration();

}

}

public void Stop()

{

if (Stopped)

{

return;

}

if (writer != null)

{

writer.Close();

}

Stopped = true;

}

protected abstract void InnerIteration();

public abstract void DrawNodes();

public virtual string Info()

{

return "";

}

}

}

**// BeesColony.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

namespace Diploma

{

public class BeesColony : Algorithm

{

public enum ProblemType

{

VRP\_TSP,

CLUSTERING\_VRP,

CLUSTERING\_CVRP,

CLUSTERING\_CVRPP,

CLUSTERING\_CVRPP\_NNC

}

public ProblemType Problem;

public int ScoutsCount;

public int GoodSitesCount;

public int BestSitesCount;

public int NeighboursForGoodSites;

public int NeighboursForBestSites;

public int ClustersCount;

public int ClusterCapacityLimit;

public double KilometerCost;

private List<Site> sites;

private Site CreateNewSite ()

{

switch (Problem)

{

case ProblemType.VRP\_TSP:

return new SiteVrpTsp(Nodes, DepotsCount, ConsumersCount, ClustersCount);

case ProblemType.CLUSTERING\_VRP:

return new SiteClusteringVrp(Nodes, ClustersCount);

case ProblemType.CLUSTERING\_CVRP:

return new SiteClusteringCvrp(Nodes, ClusterCapacityLimit);

case ProblemType.CLUSTERING\_CVRPP:

return new SiteClusteringCvrpp(Nodes, ClusterCapacityLimit, ClustersCount, KilometerCost);

case ProblemType.CLUSTERING\_CVRPP\_NNC:

return new SiteClusteringCvrppNnc(Nodes, ClusterCapacityLimit, ClustersCount, KilometerCost);

default:

return null;

}

}

public void CreateSites ()

{

sites = new List<Site>();

Site.IsStartingInitialized = false;

CreateNewSite().StartingInitialize();

for (int i = 0; i != ScoutsCount; i++)

{

sites.Add(CreateNewSite());

}

}

public Site BestSite

{

get

{

sites.Sort();

return sites[0];

}

}

public override double Value

{

get { return BestSite.Price; }

}

protected override void InnerIteration ()

{

if (IterationNumber > 100 && (IterationNumber < 500 && IterationNumber > LastChangedIteration \* 3.0 || IterationNumber > 500 && IterationNumber > LastChangedIteration \* 1.5))

{

Stop();

return;

}

sites.Sort();

for (int i = 0; i != BestSitesCount; i++)

{

while (sites[i].GoToBestNeighbour(NeighboursForBestSites))

{}

}

for (int i = BestSitesCount; i != GoodSitesCount; i++)

{

while (sites[i].GoToBestNeighbour(NeighboursForGoodSites))

{}

}

for (int i = GoodSitesCount; i != ScoutsCount; i++)

{

sites[i] = CreateNewSite();

}

}

public override void DrawNodes ()

{

BestSite.DrawNodes();

}

public override string Info()

{

string str = "";

switch (Problem)

{

case ProblemType.CLUSTERING\_CVRP:

str = string.Format("Clusters count: {0}", (BestSite as SiteClusteringCvrp).Clusters.Count);

break;

case ProblemType.CLUSTERING\_CVRPP:

str = string.Format("Estimate length: {0:0.000}. Fines: {1}.", (BestSite as SiteClusteringCvrpp).EstimateLength, (BestSite as SiteClusteringCvrpp).Fines);

break;

}

return str;

}

}

}

**// Cluster.cs**

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

namespace Diploma

{

public class Cluster

{

public List<Node> Nodes = new List<Node>();

public Node Depot;

public int CapacityLimit = int.MaxValue;

private bool nodesUpdated = true;

private Node.Point center;

public Node.Point Center

{

get

{

if (!nodesUpdated)

{

return center;

}

if (Nodes.Count == 0)

{

return null;

}

double sx = 0;

double sy = 0;

foreach (Node node in Nodes)

{

sx += node.RealPosition.x;

sy += node.RealPosition.y;

}

center = new Node.Point(sx / Nodes.Count, sy / Nodes.Count);

nodesUpdated = false;

return center;

}

}

public double GetPrice ()

{

Node.Point currentCenter = Center;

if (Nodes.Count == 0)

{

return 0;

}

double distance = 0;

foreach (Node node in Nodes)

{

distance += Node.Point.SqrDistance(node.RealPosition, currentCenter);

}

if (Depot != null)

{

distance += 2 \* Node.Point.SqrDistance(Depot.RealPosition, currentCenter);

}

return distance;

}

public int Volume

{

get

{

int volume = 0;

foreach (Node node in Nodes)

{

volume += node.Volume;

}

return volume;

}

}

public Cluster() {}

public Cluster(Cluster cluster)

{

Nodes = new List<Node>();

foreach (Node node in cluster.Nodes)

{

Nodes.Add(node);

}

Depot = cluster.Depot;

CapacityLimit = cluster.CapacityLimit;

}

public List<Node> GetDrawingNodes(Color connectionsColor)

{

List<Node> drawingNodes = new List<Node>();

if (Nodes.Count == 0)

{

return drawingNodes;

}

Node center = new Node(-1, Node.NodeType.Auxiliary, (int)Center.x, (int)Center.y, Center.x, Center.y);

foreach (Node node in Nodes)

{

center.ConnectTo(node, connectionsColor);

drawingNodes.Add(node);

}

drawingNodes.Add(center);

return drawingNodes;

}

public void Merge(Cluster cluster)

{

Nodes.AddRange(cluster.Nodes);

Nodes = Nodes.Distinct().ToList();

nodesUpdated = true;

}

public void AddNode(Node node)

{

Nodes.Add(node);

nodesUpdated = true;

}

public void AddNodes(List<Node> nodes)

{

if (nodes != null)

{

Nodes.AddRange(nodes);

}

nodesUpdated = true;

}

public void RemoveNodes()

{

Nodes.Clear();

nodesUpdated = true;

}

public void RemoveNode(Node node)

{

Nodes.Remove(node);

nodesUpdated = true;

}

public override string ToString()

{

string str = "{";

foreach (Node node in Nodes)

{

str += node.Id + ",";

}

str += "}";

return str;

}

}

}

**// ClusteringAlgorithm.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

namespace Diploma

{

public abstract class ClusteringAlgorithm : Algorithm

{

public List<Cluster> Clusters { get; protected set; }

protected Node Depot;

protected int ClustersCount;

protected void AddCluster()

{

Cluster newCluster = new Cluster();

newCluster.Depot = Depot;

Clusters.Add(newCluster);

}

protected ClusteringAlgorithm(List<Node> nodes)

{

SetNodes(nodes);

if (Depots.Count != 0)

{

Depot = Depots[0];

}

}

protected ClusteringAlgorithm(List<Node> nodes, int clustersCount)

: this(nodes)

{

ClustersCount = clustersCount;

Clusters = new List<Cluster>();

for (int i = 0; i != ClustersCount; i++)

{

AddCluster();

}

}

public override void DrawNodes()

{

List<Node> drawingNodes = new List<Node>();

for (int i = 0; i != Clusters.Count; i++)

{

Cluster cluster = Clusters[i];

if (cluster.Nodes.Count == 0)

{

continue;

}

Node center = new Node(-1, Node.NodeType.Auxiliary, (int)Clusters[i].Center.x, (int)Clusters[i].Center.y, Clusters[i].Center.x, Clusters[i].Center.y);

if (cluster.Nodes.Count == 0)

{

drawingNodes.Add(center);

continue;

}

foreach (Node node in cluster.Nodes)

{

center.ConnectTo(node, Color.LightGray);

drawingNodes.Add(node);

}

drawingNodes.Add(center);

}

Node depot = Clusters[0].Depot;

if (depot != null)

{

drawingNodes.Add(depot);

}

TaskController.DrawNodes(drawingNodes);

}

public override double Value

{

get

{

double value = 0;

foreach (Cluster cluster in Clusters)

{

value += cluster.GetPrice();

}

return value;

}

}

}

}

**// KMeans.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

namespace Diploma

{

public class KMeans : ClusteringAlgorithm

{

public KMeans(List<Node> nodes, int clustersCount)

: base(nodes, clustersCount)

{

}

private Node.Point[] centers;

private void SetCenters()

{

if (centers == null)

{

centers = new Node.Point[Clusters.Count];

var xs = from node in Nodes

select node.RealPosition.x;

double xMin = xs.Min();

double xMax = xs.Max();

var ys = from node in Nodes

select node.RealPosition.y;

double yMin = ys.Min();

double yMax = ys.Max();

for (int i = 0; i != centers.Length; i++)

{

double cx = TaskController.Rnd.NextDouble() \* (xMax - xMin) + xMin;

double cy = TaskController.Rnd.NextDouble() \* (yMax - yMin) + yMin;

centers[i] = new Node.Point(cx, cy);

}

}

else

{

for (int i = 0; i != centers.Length; i++)

{

if (Clusters[i].Nodes.Count == 0)

continue;

centers[i] = Clusters[i].Center;

}

}

}

private void GenerateClusters()

{

foreach (Cluster cluster in Clusters)

{

cluster.RemoveNodes();

}

foreach (Node node in Nodes)

{

if (node.Type != Node.NodeType.Consumer)

continue;

if (IterationNumber == 0)

{

bool b = false;

foreach (Cluster cluster in Clusters)

{

if (cluster.Nodes.Count == 0)

{

cluster.AddNode(node);

b = true;

break;

}

}

if (b)

continue;

}

double minDist = double.PositiveInfinity;

Cluster closest = Clusters[0];

for (int i = 0; i < centers.Length; i++)

{

double dist = Node.Point.SqrDistance(centers[i], node.RealPosition);

if (dist < minDist)

{

minDist = dist;

closest = Clusters[i];

}

}

closest.AddNode(node);

}

}

protected override void InnerIteration()

{

if (IterationNumber - LastChangedIteration > 2)

{

Stop();

return;

}

SetCenters();

GenerateClusters();

}

}

}

**// NearestNeighbourChain.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Diploma

{

public class NearestNeighbourChain : ClusteringAlgorithm

{

private readonly int capacityLimit;

public NearestNeighbourChain(List<Node> nodes, int capacityLimit)

: base(nodes)

{

this.capacityLimit = capacityLimit;

GenerateClusters();

lastReviewedCluster = -1;

}

private void GenerateClusters()

{

Clusters = new List<Cluster>();

for (int i = 0; i != Consumers.Count; i++)

{

AddCluster();

Clusters[i].AddNode(Consumers[i]);

Clusters[i].CapacityLimit = capacityLimit;

}

}

private int lastReviewedCluster;

private int NearestCluster(int cluster)

{

double minDistance = double.PositiveInfinity;

int result = -1;

for (int i = 0; i != Clusters.Count; i++)

{

if (i == cluster)

continue;

double dist = Node.Point.SqrDistance(Clusters[i].Center, Clusters[cluster].Center);

if (dist < minDistance)

{

minDistance = dist;

result = i;

}

}

return result;

}

private bool merged = false;

protected override void InnerIteration()

{

lastReviewedCluster++;

if (lastReviewedCluster >= Clusters.Count)

{

if (!merged)

{

Stop();

return;

}

lastReviewedCluster = 0;

merged = false;

}

if (Clusters.Count <= 1)

{

return;

}

int startingCluster = lastReviewedCluster;

int currentCluster = startingCluster;

int lastCluster = startingCluster;

do

{

int nearest = NearestCluster(currentCluster);

if (nearest == lastCluster)

{

break;

}

lastCluster = currentCluster;

currentCluster = nearest;

} while (true);

if (Clusters[lastCluster].Volume + Clusters[currentCluster].Volume <= Clusters[lastCluster].CapacityLimit)

{

Clusters[lastCluster].Merge(Clusters[currentCluster]);

Clusters.RemoveAt(currentCluster);

merged = true;

}

else if (Clusters[lastCluster].Volume + Clusters[currentCluster].Volume <= Clusters[currentCluster].CapacityLimit)

{

Clusters[currentCluster].Merge(Clusters[lastCluster]);

Clusters.RemoveAt(lastCluster);

merged = true;

}

}

public override string Info()

{

return string.Format("Clusters count: {0}. Stopped: {1}", Clusters.Count, Stopped);

}

}

}

**// Site.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

namespace Diploma

{

public abstract class Site : IComparable

{

protected List<Node> Nodes;

public List<Node> DrawingNodes { get; protected set; }

protected Site (List<Node> nodes)

{

Nodes = new List<Node>();

for (int i = 0; i != nodes.Count; i++)

{

Nodes.Add(nodes[i]);

}

}

public abstract double Price { get; }

protected abstract void GoToNeighbour(Site site);

protected abstract Site GetNeighbour();

public abstract object Result { get; }

private List<Site> GenerateNeighbours(int count)

{

List<Site> result = new List<Site>();

for (int i = 0; i != count; i++)

{

result.Add(GetNeighbour());

}

return result;

}

public bool GoToBestNeighbour(int countOfNeightbours)

{

List<Site> neighbours = GenerateNeighbours(countOfNeightbours);

neighbours.Sort();

if (neighbours[0].Price < this.Price)

{

GoToNeighbour(neighbours[0]);

return true;

}

return false;

}

public static bool IsStartingInitialized { protected get; set; }

public virtual void StartingInitialize()

{

IsStartingInitialized = true;

}

public abstract List<Node> PrepareToDraw(Color connectionsColor);

public void DrawNodes()

{

DrawingNodes = PrepareToDraw(TaskController.ConnectionsPen.Color);

TaskController.DrawNodes(DrawingNodes);

}

public virtual int CompareTo(object obj)

{

Site site = obj as Site;

return Price.CompareTo(site.Price);

}

}

}

**// ClusteringToTsp.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

namespace Diploma

{

class ClusteringToTsp : Algorithm

{

private List<Cluster> clusters;

private List<Node> notClusteredNodes;

private List<double> values;

private List<BeesColony> colonies;

private double kilometerCost;

public ClusteringToTsp (List<Cluster> clusters, List<Node> notClusteredNodes, double kilometerCost)

{

this.clusters = clusters;

this.notClusteredNodes = notClusteredNodes;

this.kilometerCost = kilometerCost;

}

public void Calculate(int scoutsCount, int goodSitesCount, int bestSitesCount, int neighboursForGoodSites, int neighboursForBestSites)

{

values = new List<double>();

colonies = new List<BeesColony>();

foreach (Cluster cluster in clusters)

{

if (cluster.Nodes.Count == 0)

{

continue;

}

BeesColony colony = new BeesColony();

colony.Problem = BeesColony.ProblemType.VRP\_TSP;

colony.ClustersCount = 1;

colony.IsCalcLastChange = true;

colony.ScoutsCount = scoutsCount;

colony.GoodSitesCount = goodSitesCount;

colony.BestSitesCount = bestSitesCount;

colony.NeighboursForGoodSites = neighboursForGoodSites;

colony.NeighboursForBestSites = neighboursForBestSites;

List<Node> colonyNodes = new List<Node>();

colonyNodes.AddRange(cluster.Nodes);

if (cluster.Depot != null)

{

colonyNodes.Add(cluster.Depot);

}

colony.SetNodes(colonyNodes);

colony.CreateSites();

colony.IterateToStop();

values.Add(colony.Value);

colonies.Add(colony);

}

Stop();

}

private double SumValue

{

get

{

if (values == null)

{

return -1;

}

return values.Sum();

}

}

public override double Value

{

get

{

if (notClusteredNodes == null)

{

return SiteClusteringCvrpp.GetPrice(SumValue, kilometerCost, 0);

}

double fines = (from node in notClusteredNodes

select node.Fine).Sum();

return SiteClusteringCvrpp.GetPrice(SumValue, kilometerCost, fines);

}

}

protected override void InnerIteration()

{

//throw new NotImplementedException();

}

public override void DrawNodes()

{

List<Node> drawingNodes = new List<Node>();

for (int i = 0; i != colonies.Count; i++)

{

drawingNodes.AddRange(colonies[i].BestSite.PrepareToDraw(TaskController.GetDrawingColor(i)));

}

if (notClusteredNodes != null)

{

drawingNodes.AddRange(notClusteredNodes);

}

TaskController.DrawNodes(drawingNodes);

}

}

}