Source code documentation

June 2018

This is the documentation for the source code employed in the computational study in the master thesis *A matheuristic for the inventory routing problem with divisible pickup and delivery* at the Norwegian University of Science and Technology by Einar Aastveit, Simen Vadseth and Tuva Toftdahl Staver

The executing class is explained and a description of how to run the program is provided. An overview of how the program is designed is also presented. The classes of the program and the corresponding functions and attributes of each class are listed. An explanation of each class member is not provided. However, the source code is extensively commented. Also, only a selection of private members and helper functions are listed here. The most recent version of the source code is available in the master branch at github.com/einarea/IRPProject

The C++ files are compiled in Visual Studio 2015. The xpress files used by the optimization classes, must be properly linked to run the program. These are found under *include* and *lib* in the project repository. Also, the program uses gnuplot and boost libraries to plot the arc-flow network. The boost binaries can be found at
https://sourceforge.net/projects/boost/files/boost-binaries/. The branch *noPlot* runs the program without these external libraries.

Contents

1 Program design **2**

2 Executing Class **4**

2.1 IRPmodel 4

3 Classes for optimization models **6**

3.1 IRP 6

3.2 VRPmodel 7

3.3 RouteProblem 8

4 Dataholder classes **9**

4.1 Node 9

4.2 Node::Edge 11

4.3 NodeIRP 12

4.4 NodeIRP::EdgeIRP 14

4.5 NodeIRPHolder 15

4.6 NodeInstance 17

4.7 NodeInstanceDB 19

4.8 NodeStrong 20

4.9 Route 21

5 Classes to store solution information **24**

5.1 SolutionInfo 24

6 Miscellaneous classes **26**

6.1 Solution 26

6.2 GraphAlgorithm 29

6.3 ModelParameters 30

1 Program design

The program provides independent classes for each optimization model. The IRP class is capable of solving various types of IRP, in particular, the IRP-DPD studied in this thesis. It is also used to solve a version where the arc variables are relaxed. The VRPmodel class solves VRP problems for a set of input nodes with given quantities. The RouteProblem class solves various types of route problems, in particular, it is employed by all intensification operators and the final route selection problem.

The program employs various classes to store data in the solution process. These classes are illustrated in figure 1. The NodeInstance class is used to store predefined customer information such as position, demand and inventory limits. The class NodeIRP represents a customer node in a particular period. This class provides fields for storing variable solution values such as quantity served and inventory. All NodeIRP objects that represent a customer node in any period share the same NodeInstance object from where it accesses the predefined information about a customer.

The solution class is used to store complete information about a feasible solution to an IRP problem. It uses a vector of NodeIRPHolder objects to store data about all customer nodes for all periods. For an instance of n customer nodes the Solution class holds n NodeIRPHolder objects. For T periods each NodeIRPHolder holds a vector of T NodeIRP objects. Each NodeIRP object represents either a pickup node or a delivery node in a particular period. Also, each NodeIRP object may have one or several edges to other nodes. The edge class is used to store load and routing information. In an integer solution, only the depot node has more than one edge.

The solution class provides a set of functions which returns information about the current solution, such as objective values, holding cost and transportation cost. Feasibility checks, amount of pickup and delivery, type of visits and number of routes are other pieces of information available. Also, this class has various functions to print the solution in the terminal or plot the arc-flow network.

The solution class can also return a vector of Route objects that correspond to the routes of the solution. These routes can be used as input in the Route selection problem. In the matheuristic routes are collected from several solution objects during the iterations before the final route selection problem is solved.

The node and edge structure is well suited for graph traversal algorithms. Collection of routes, calculating costs and various algorithms to modify routes uses the network formed by nodes

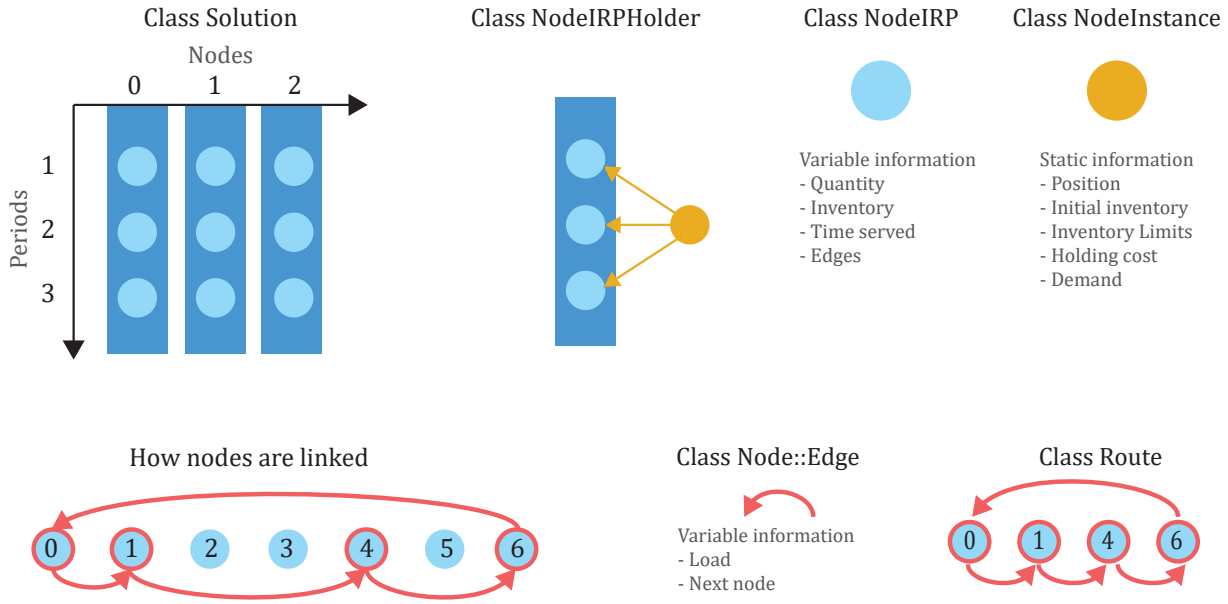

Figure 1: Illustration of classes that store data in the application

and edges. Especially the strong component algorithm in class Graph-Algorithm uses the Nodes and edges of a Solution Object to find the strong components in the solution. Subsequently, the corresponding edges are checked for subtour violation.

2 Executing Class

2.1 IRPmodel

IRPmodel is the executing class of the application. It allows to open and create instances. Also, the matheuristic and exact methods are run from this class. First a node instance is opened by calling NodeInstanceDB * openInstance which returns a pointer to a database of NodeInstance objects. The method use the path specified by getFileName which default path is set to ../Instances/ in the project repository.

To run the heuristic method Heurestic is called. The function takes the NodeInstance database as the input value. Also, optional arguments which are used to store solution data during the solution process can be passed to the function. If a pointer to a SolutionInfo::InstanceInfo object is passed to Heurestic as a second argument, this pointer is returned and may be used to print solution details.

The Heurestic function follows the framework established in the thesis. Various local time objects and solution holders are first defined. The two phases with a relaxed IRP and the subsequent VRPs are used to construct a solution. Then follows the intensification search before the updating of the tabu list and the adding of diversification constraints. Finally, when the iterative loop exceeds the Heuristic running time, the final route selection problem is solved.

The various model parameters are set in class ModelParameters. The time budget may be set for each phase of the matheuristic. Simultaneous is set to true if the relaxed IRP is to be solved with the simultaneous arc set. Further, SECs are included in the model if SubtourElimination is set to true. ALPHA and EDGE_WEIGHT are the parameters of the subtour algorithm described in the thesis. nVehicles sets how many vehicles the instance is to be solved with. Also, one and only one of the diversification constraints should be set to true. The proportion of required change and nodes to lock in the tabu list are options also included in the ModelParameters class. If the RecordImprovement parameter is set to true the final route selection problem is run in each diversification iteration to produce the solution that would emerge if the heuristic was to exit after that particular iteration. The data is stored in the SolutionInfo::InstanceInfo object passed to the Heuristic function.

The exact method employs the appropriate parameters from the ModelParameters class such as those related to the duration of the IRP, number of vehicles and subtour elimination. The valid inequalities proposed in the thesis can easily be included in the model by removing the comment marks in front of the corresponding inequality. It is also possible to solve the LP

relaxation of the exact model in this function. The exact function is also responsible for printing the solution to a file. However, it may also return a SolutionInfo::InstanceInfo object where more information can be stored.

When writing to files, the folder in which the file is to be placed in should exist. If the appropriate folder does not exist, the behavior of the program is undefined.

Functions

int main()

Instance * Heurestic(NodeInstancdDB&, InstanceData*)

Instance * ExactMethod(NodeInstancdDB&, InstanceData*)

3 Classes for optimization models

3.1 IRP

Description: Class to solve inventory routing problems

Attributes
double startTime
double bestBound
double bestSol
doulbe solutionTime
SolutionInfo::InstanceInfo * InstanceData
int nSubtourCuts = 0

Functions
Constructors
IRP(const NodeInstanceDB&, bool relaxed = false, SolutionInfo::InstanceInfo * instance = nullptr)

Solver functions
Solution * solveModel()
Solution * solveLPModel()

Functions to add constraints
void addVisitConstraint(double ** VisitMatrix, int selection, int relaxed = -1)
void addValidIneq(int ValidInequality)
void addHoldingCostCtr(double holdingCost)
void addRouteCtr(vector<Route*> routes, int relaxed = -1)

Functions for SECs
void buildGraph(vector <Node*> , int, bool includeDepot, double weight = 0.01)
void buildStrongGraph(vector <NodeStrong*> , int, bool includeDepot, double weight = -1)
bool sepStrongComponents(vector<XPRBcut>)
void addSubtourCut(vector<vector <Node *» , int t, bool , vector<XPRBcut>)
void printStrongCompAlgorithm()

Destructors
~IRP()

3.2 VRPmodel

Description: Class to solve vehicle routing problems. This particular class solves the problem for a set of NodeIRP objects with strictly positive quantity servings.

Attributes
time_t StartTime
double SolutionTime
time_t lastSolutionFound

Functions
Constructors
VRPmodel(const NodeInstanceDB db, vector<NodeIRP*> nodes, int period)

Solver functions
void solveModel(Solution * currentSol = 0)

Destructors
~VRPmodel()

3.3 RouteProblem

Description: Class to solve a routing problem for a set of routes. The problem is capable of solving the general route selection problem where all routes are free, however the class also solves more rigid versions of the problem. Various functions corresponding to the intensification operators are provided by the class.

Attributes
int ShiftPeriod
time_t StartTime
double SolutionTime

Functions
Constructors
RouteProblem(const NodeInstanceDB Instance, vector<Route*> routes)

Route information
vector<Route *> getSelectedRoutes()
void printRouteType()
void saveRouteType()
static void printRouteTypeToFile(string filename, double improvement)
int getnRoutes()
void printRouteMatrix()

Functions used by intensification operators
void addRestrictedShiftCtr(double nRoutes, double oldDel, double oldPick)
void formulateMIP()
void addChangeCtr()
int getShiftPeriod()
void formulateMinVisitProblem()
void shiftQuantityCtr(int quantity)
void lockRoutes(vector<int> Periods)

Destructors
~RouteProblem()

4 Dataholder classes

4.1 Node

Description: General parent class for all node classes. Provides the general structure of a node where each node may hold a set of edges which link them to other nodes. Also, the class provides several constants to describe the state of a node, which for instance are used to tabu nodes.

Attributes
int NodeID
int State
vector <Edge*> Edges
static const int TABU_EDGE
static const int FREE
static const int REMOVE
static const int TABU

Functions
Constructors
Node(int id, vector<Edge*> edges)
Node(const Node)
Node(int id)

Overloaded operators
virtual Node operator =(const Node)
bool operator==(const Node &) const
bool operator!=(const Node&) const

Accessor functions
Edge * getEdge(Node & n)
int getnEdges()
Edge * getEdge()
vector <Edge*> getEdges()
int getState()
int getId() const
bool isDepot() const
bool hasEdge(Edge *)

Modifier functions
void setId(int id)
void setState(int s)
Edge* addEdge(double value, Node * child)
Edge* addEdge(Edge *)
Edge* addEdge(Node * child)
void removeEdge(Node &child)
void deleteEdges()
void deleteEdge(Node *)
void removeEdges()

Destructors
virtual ~Node()

4.2 Node::Edge

Description: This class links nodes together in a network. The network is single-linked, that is only the next node in the network is known for a node.

Attributes
double Value
Node &EndNode

Functions
Constructors
Edge(Node endNode, double value)

Accessor functions
Node *getEndNode()
double getValue()

Modifier functions
void setValue(double)

Destructors
virtual ~Edge()

4.3 NodeIRP

Description: Child class of class Node. Represents a customer node and a customer's variable customer information.

Attributes
double Quantity
double Inventory
double TimeServed
const NodeInstance &NodeData

Functions
Constructors
NodeIRP(const NodeIRP&)
NodeIRP(const NodeInstance& data)

Overloaded operators
NodeIRP operator =(const NodeIRP& cpNode)

Accessor functions
NodeIRP* getNextNode()
bool inArcSet(const NodeIRP *) const
bool inArcSet(const NodeInstance *) const
double getTransCost(const NodeIRP * node) const
double getTravelTime(const NodeIRP * node) const
double getTransCost(const NodeInstance * node) const
double getTravelTime(const NodeInstance * node) const
bool isColocated(const NodeIRP * node) const
EdgeIRP * getEdge()
static NodeIRP * getNode(Node *)
vector <EdgeIRP*> getEdges()
double getOutflow()
double getPosX() const
double getPosY() const
double getHoldCost()
const NodeInstance getData() const
bool isDelivery()
void copyEdge(NodeIRP::EdgeIRP * edge, NodeIRP *)

Modifier functions
void addEdge(double loadDel, double loadPick, NodeIRP * child, double value)

Destructor
~NodeIRP()

4.4 NodeIRP::EdgeIRP

Description: Child class of Node::Edge. This class represents an arc in the arc-flow model. It store variable load information about an arc.

Attributes
double LoadDel
double LoadPick

Functions
Constructors
EdgeIRP(Node *child, double loadDel, double loadPick, double value)

Accessor functions
NodeIRP * getEndNode()

Destructors
~EdgeIRP()

4.5 NodeIRPHolder

Description: Class that organizes a customer node over all periods. It holds a vector of NodeIRP objects, where each NodeIRP object represents a particular period.

Attributes
vector< NodeIRP* > NodePeriods

Functions
Constructors
NodeIRPHolder(NodeInstance instance)
NodeIRPHolder(NodeIRPHolder cpNode)

Overloaded operators
void operator =(const NodeIRPHolder & cpNode)

Accessor functions
int getId() const
NodeIRP * getNode(int period)
int isInventoryFeasible()
int isInventoryFeasible(int period)
double getMinQuantity()
double getExcessQuantity()
int getXPos()
int getYPos()
int getHoldingCost()
int getInitialInventory()
int getDemand(int Period)
double getOutflow(int period)
vector <NodeIRP::EdgeIRP*> getEdges(int period)
NodeIRP::EdgeIRP * getEdge(int period)
double getHoldCost(int period)
double quantity(int period)
double inventory(int period)
double timeServed(int period)
bool isDelivery() const
const NodeInstance getData()

Modifier functions
void removeMinQuantity()
void addEdge(double loadDel, double loadPic, NodeIRPHolder * child, int period, double value)
void changeQuantity(int period, double quantity)

Destructors
~NodeIRPHolder()

4.6 NodeInstance

Description: Class to store static information about a customer node. Only one node instance needs to be created for each customer node, and can then be shared among all objects which needs access to this information.

Attributes
int NodeID
double PosX
double PosY
bool Delivery
int HoldingCost
int LowerLimit
int UpperLimit
vector<int> Demand
int InitInventory
int nPeriods
vector<NodeInstance*> ForbiddenNodes
NodeInstance* ColocatedNode

Functions
Constructors
NodeInstance(int NodeId, bool Del, int posX, int posY, int nPer, int inital, int holdingCost, int upperLim, int lowerLim, vector<int> Demand)
NodeInstance(int NodeId, bool Del, int posX, int posY, int nPer, int randSeed)
NodeInstance(int id, bool Del, int nPer, int randSeed, int positionType = RANDOM_POS, vector<int> clusterPosition = -1, -1)

Overloaded operators
bool operator==(const NodeInstance &) const
bool operator!=(const NodeInstance &node) const

Accessor functions
bool inArcSet(const NodeInstance * n) const
bool isDelivery() const
bool isColocated(const NodeInstance *) const
bool hasArc(NodeInstance * node)
double getXpos() const

double getYpos() const
int getId() const
double getDistance(const NodeInstance * node) const
NodeInstance * getColocatedNode()
double getTransCost(const NodeInstance * node) const
double getTravelTime(const NodeInstance* node) const

4.7 NodeInstanceDB

Description: Class that represents a problem instance. This is a container class to store all NodeInstance objects. The class is responsible for creating instances, writing instances to files and open instances. The class also defines the sets used by the various models. The class provides constant identifiers for the different instance types.

Attributes
vector <int> Periods
vector <int> AllPeriods
vector <NodeInstance *> AllNodes
vector <NodeInstance *> DeliveryNodes
vector <NodeInstance *> PickupNodes
vector <NodeInstance *> Nodes
vector <NodeInstance *> Depot
double Capacity

static const int DELIVERY
static const int COLOCATED
static const int SEPARATE
static const int CLOSENESS_TO_DEPOT
static const int CLUSTER

Functions
Services of the class
static NodeInstanceDB * createInstance(int nCustomers, int nPeriods, int version)
static NodeInstanceDB * createClosenessToDepotInstance(int nCust, int nPeriods, int ver)
static NodeInstanceDB * createPDInstance(int nCustomers, int nPeriods, int version)
static NodeInstanceDB * createDelInstance(int nCustomers, int nPeriods, int version)
static NodeInstanceDB * createClusterInstance(int nCus, int nClusters, int nPeriods, int ver)
static NodeInstanceDB * openInstance(int nCustomers, int nPeriods, int version)
static string getFilename(int nCustomers, int nPeriods, int version)

Accessor functions
int getNumNodes() const
int getnPeriods() const
void writeInstanceToFile(ofstream &instanceFile, string Filename)

4.8 NodeStrong

Description: Node class used by the strong component algorithm.

Attributes
int Index = -1
int LowLink = -1
bool onStack
vector <Node> Nodes

Functions
Constructors
NodeStrong(int id, vector <Edge*> edges)
NodeStrong(int id)
NodeStrong(Node * node)

Accessor functions
int getIndex()
int getLowLink()
static NodeStrong * getStrongNode(Node *)
bool isOnStack()

Modifier functions
void setIndex(int)
void setLowLink(int)
void setOnStack(bool)

Destructors
~NodeStrong()

4.9 Route

Description: Class that stores a route. It contains a vector of pointers to NodeIRP objects which appear in the route. The class provides a rich collection of route modifier functions and route generation algorithms. The class also includes constant identifiers to describe how the route was created.

Attributes
vector <NodeIRP*> route
int State
double constructionCost
int Id
int Period

States
static const int ORIG = 0
static const int SIMPLE_INSERTION = 1
static const int SIMPLE_REMOVAL = 7
static const int INSERTION_REMOVAL = 2
static const int DOUBLE_INSERTION_REMOVAL = 6
static const int LEAST_SERVED_REMOVAL = 3
static const int LEAST_SERVED_INSERTION = 4
static const int RESTRICTED_LEAST_SERVED_REMOVAL = 8
static const int MERGE = 5

Functions
Constructors
Route()
Route(vector <NodeIRP*> & path)
Route(const Route & r)

Overloaded operators
Route& operator = (const Route&)
Route& operator = (const Route * cpRoute)
bool operator < (const Route&) const
bool operator ==(const Route&)
bool operator !=(const Route& r)

NodeIRP * operator [](int i)

Accessor functions
int getPeriod()
bool isFeasible()
int getSize()
bool coincide(Route* r)
int getDirection() const
bool sameDirection(const Route *)
bool inRoute(const Node *)
vector<Route*> getSubgraphs(int n) const
bool isDuplicate(const Route * r)
double getTransCost() const
NodeIRP * front()
NodeIRP * back()
int getPosition(Node * node)
int getId() const
int** getRouteMatrix(int gridSize)
double getResidualCapacity(int capacity)
vector<NodeIRP*> getSubroute(int selection, double & minCost)

Modifier functions
void setId(int id)
void clearState()
void insertCheapestNode(vector<const NodeIRP*> nodes)
void removeSubgraph(vector <NodeIRP*>)
void generateRoute(const Route *, list<Route> & routeHolder, list<Route> & tabuRoutes)
void reverseRoute()
void setPeriod(int period)
int removeNode(NodeIRP*, Route *)
vector<NodeIRP*> cheapestInsertion(Route* subroute, double &minCost)
vector<NodeIRP*> cheapestRemoval(int subroutesize, double &maxCost)
int getTotalDelivery()
int getTotalPickup()
void insertSubgraph(Route* subgraph)
void insert(NodeIRP * start, NodeIRP * end, Route* subroute)
void removeNode(NodeIRP *)

void removeNodes()
double removeSubroute(int selection)

Information functions
void printPlot(string filename) const
void printRoute()

Destructors
~Route()

5 Classes to store solution information

5.1 SolutionInfo

Description: Class to store solution processs for several instances in order to calculate the average solution process for an instance type.

Attributes
vector<InstanceInfo> Instances
int MaxTime = -1

Functions
Accessor functions
double getMaxTime()
void printAllInstancesToFile(string name)
void printAverageInstancesToFile(string name)
InstanceInfo * newInstance(string name)

SolutionInfo::InstanceInfo

Description: Class to store the solution process for an instance. It contains a list of information objects representing each point in time during the solution process.

Attributes
double bestObjective = -1
int nRoutePool = -1
int nIterations = -1
int nIntIterations = -1
double relaxedObj = -1
int nDivisible = -1
int nSingleService = -1
int nSimultanouesService = -1
int nTotalNodesServed = -1
int nRoutes = -1
double solTime = -1
double bestBound = -1
double irpRel = -1
int MaxTime = -1
list<Information> infoHolder

string Name

Functions
Constructors
InstanceInfo(string name)

Accessor functions
double getBestSolutionTime()
double getPercentSolutionTime(double percent)
double getGap(int t)
double getAverageObjective(int t)
const Information * getInfo(int time) const
SolutionInfo::InstanceInfo * const printInstanceToFile(bool heurestic = true)
void printInstanceTimeSeriesToFile(bool heurestic = true)

Modifier functions
void fillInfo()
void addSolutionPoint(int state, double objectiveVal, double time)

SolutionInfo::Information

Description: Structure to store objective value and the solution's state at a particular point in time.

Attributes
string State
double ObjectiveVal
double Time

Functions
Constructors
Information(string state, double objectiveVal, double time)

6 Miscellaneous classes

6.1 Solution

Description: Class to store solutions. It stores NodeIRPHolder objects for each customer node. Several accessor functions provide solution information. Also it implements all intensification operators. Thus, many of the functions in this class modify the current solution object.

Attributes
vector<NodeIRPHolder *> Nodes
vector<Route *> TabuRoutes
vector<Route*> SelectedRoutes
vector<Route*> GeneratedRoutes
int SolID
double SolutionTime

Functions
Constructors
Solution(const NodeInstanceDB model)
Solution(const NodeInstanceDB model, vector<NodeIRPHolder*> nodes)
Solution(const Solution)

Overloaded operators
Solution operator = (const Solution)
bool operator ==(const Solution)
void updateSolution(Solution)

Accessor functions
int *** getRouteMatrix()
vector<NodeIRPHolder *> getNodes()
const NodeIRP* getLeastServed(int period) const
const NodeIRP* getLeastServed(vector<NodeIRP*>, int period) const
int getPeriodWithMinExcess(const vector <int> Periods)
vector <NodeIRP *> getVisitedNodes(int period)
vector <NodeIRP*> getNotVisitedNodes(int period)
static double ** Solution::getVisitDifference(const Solution sol1, const Solution sol2)
void getStrongComponents()
int getnDivisible()

int getnSingleService()
int getTotalNodesServed()
int getnSimultaneousVisits()
int getnRoutes()
bool isFeasible() const
int getMaxShiftPeriod()
double getNumberOfRoutes(int period) const
double getResidualCapacity(int period)
double getTotalDelivery(int period)
double getTotalPickup(int period)
double getNodeVisits(int period)
double ** getVisitedMatrix() const
double getDeliveryNodeVisits(int period)
double getPickupNodeVisits(int period)
vector <Route *> getRoutes(int period) const
int getInfeasiblePeriod()
vector<Route*> getAllRoutes()
double getObjective() const
double getTransportationCost() const
double getHoldingCost() const
double getHoldingCost(int period) const
double getTransportationCost(int period) const

Modifier functions
void solveVRP(int period)
void routeSearch(int REQUIRE_CHANGE = ModelParameters::NO_CHANGE)
void shiftQuantity(int SELECTION)
void shiftQuantityMIP(int PeriodSelection)
void addTabuRoutes(vector <Route*> routes, vector<Route*> origRoutes)
int selectPeriod(int selection)
Route* removeNodeFromPeriod(int period, const NodeIRP* remNode)
void generateRoutes(vector<Route* >routeHolder)
Route * insertNodeInPeriod(int period, const NodeIRP * insNode)
void solveInventoryProblem()

Information functions
void print(string filname, int weight)

void printSolution()
void plotPeriod(int t, string filename)
void plot(string filename)

Destructors
~Solution()

6.2 GraphAlgorithm

Description: This class implements the seperation by strong components algorithm. It provides functions to find the routes of a solution graph and calculate the similarities between two solutions. The plotting of the arc-flow network is also performed by this class.

Functions

Service functions

static void sepByStrongComp(vector<NodeStrong *> graph, vector<vector<Node*» result)

static void printGraph(vector<NodeIRP *> graph, string filename, int weight=0)

static double getSimiliarity(vector <NodeIRP*> graph1, vector <NodeIRP*> graph2)

static void getRoutes(vector<Node*> graph, vector<vector<Node*» routes)

6.3 ModelParameters

Description: This class defines various constant parameters and identifiers that are used by the program.

Time parameters
static int MAX_RUNNING_TIME_IRP
static int MAX_RUNNING_TIME_VRP
static int MAX_TIME_ROUTE_PROBLEM
static int INTENSIFICATION_TIME

static const int TERMINATE_IF_NO_NEW_SOLUTION
static const int HEURESTIC_TIME = 3600
static const bool RecordImprovement = true

Number of vehicles
static const int nVehicles

Subtour parameters
static const bool SUBTOUR_ELIMINATION = true
static const int EDGE_WEIGHT = 10
static const int ALPHA = 10

The simultaneous option
static const bool Simultanoues_RelaxedIRP
static bool Simultaneous = false

Diversification constraints
static const bool RouteChange = false
static const bool MinChanges = true
static const bool HoldingCost = false;

static const int ROUTE_LOCK = 50
static const int MIN_CHANGE = 60
static const int TabuLength = 2
static const int TABU_LOCK = 15
. static const int HOLDING_COST_INCREMENT = 5

static const int ExcessParameter = 30

Instance generation
static const int TRANSCOST_MULTIPLIER = 13
static const int SERVICECOST_MULTIPLIER =100
static const int TRAVELTIME_MULTIPLIER = 1
static const int SERVICETIME = 20
static const int maxTime = 480
static const int VehiclePenalty = 100000

static const int HoldingCostLow = 1
static const int HoldingCostHigh = 5

static const int DemandDelLow = 5
static const int DemandDelHigh = 50
static const int DemandPickLow = 5
static const int DemandPickHigh = 50

static const int LBDel = 5
static const int LBPic = 0

static const int UBDelLow = 75
static const int UBDelHigh = 100
static const int UBPickLow = 75
static const int UBPickHigh = 100

static const int InitInventoryDel = 15
static const int InitInventoryPick = 15

Identifiers
static const int MAX_SHIFT = 3
static const int RESTRICTED_SHIFT = 4
static const int MINIMIZE_VISITS = 5
static const int INFEASIBLE = 6
static const int SLACK = 10
static const int REQUIRE_CHANGE = 31
static const int NO_CHANGE = 46

static const int GLOBAL_CUTS = 33
static const int LOCAL_CUTS = 34

static const int LBPick = 0
static const int ForceVisits = 33
static const int ForceChanges = 34
static const int MinimumNodeVisit = 22
static const int MinimumFlow = 23
static const int MinimumInventory = 24
static const int SubsetSizeMinFlow = 3

static const int REMOVE_SERVICE = 0
static const int HIGHEST_HOLDINGCOST = 1
static const int HIGHEST_TRANSPORTATIONCOST = 22
static const int HIGHEST_RESIDUAL_CAPACITY = 23
static const int HIGHEST_TOTALCOST = 2

static const int LOAD = 1
static const int X = 2

static const int LOWER_LIMIT_BREAK = 51
static const int UPPER_LIMIT_BREAK = 52
static const int WITHIN_LIMITS = 53

static const int MIN_SERVICE = 33
static const int CLOCKWISE = 54
static const int COUNTER_CLOCKWISE = 53

static const int ROUTE_SEARCH = 76
static const int SHIFT_QUANTITY = 77
static const int IRP_REL = 78
static const int VRP = 79
static const int INTENSIFICATAION_END = 80
static const int FINAL_SOL = 81
static const int BBNode = 82