Pickup and Delivery Problem with Hub Reloading

Author: Li Li

Supervisor: Jesper Larsen

Informatics and Mathematical Modelling
Technical University of Denmark

Preface

This thesis is the final report in my scholastic career. I will end my master program in Technical University of Denmark by completing this project. At the time of finishing this thesis, I would like to thank to some people who have helped me during the process of doing this project.

Firstly, I would like to thank to my supervisor Jesper Larsen for his excellent ideas, unwearied edifications and endless patience. Secondly, I would like to thank to my parents for their auspices of my school works in Denmark. And I would also like to say thanks to my best friend, my boyfriend Zhenyu Yuan. He encouraged, comforted me when I met any problem doing this project and accompanied me passing this strenuous working process.

Content

Abstract

1 Introduction

1.1 Motivation……………………………………………………………………………6
1.2 Description of the working task and purpose…………………………………………...9

2 Routes with Central Depot

2.1 Problem formulation…………………………………………………………………….9
2.2 Mathematical formulation……………………………………………………………12
2.3 Literature review…………………………………………………………………………..14

3 Method Discussion

3.1 Forming routes

3.1.1 Constructive heuristics…………………………………………………………21
3.1.1.1 Insert_By_Distance heuristic …………………………………………………22
3.1.1.2 Sweep_By_Angle heuristic……………………………………………………26
3.1.1.3 Results comparison……………………………………………………………….28

3.1.2 Improved heuristics
3.1.2.1 Based on Sweep_By_Angle……………………………………………………30
3.1.2.1.1 Result from the modified Sweep_By_Angle…………………………….36
3.1.2.1.2 Method analysis about Sweep_By_Angle……………………………….39

3.1.2.2 Based on Insert_By_Distance…………………………………………………40
3.1.2.2.1 Result from the modified Insert_By_Distance…………………………...45
3.1.2.2.2 Method analysis about Insert_By_Distance……………………………...46
3.1.2.2.2.1 review the method and analyze it…………………………………..46
3.1.2.2.2.2 some new idea for modifying method……………………………...48

3.2 Connection routes
3.2.1 Searching relative routes in different parts…………………………………………..52
3.2.2 Rearrange the overload routes…………………………………………………….53

3.3 Experimental results

4 Conclusion
4.1 Based on Sweep_By_Angle and its pertinent approaches…………………………………66
4.1.1 disadvantage
4.1.2 advantage

4.2 Based on Insert_By_Distance and its pertinent approaches……………………………….66
4.2.1 disadvamtage
4.2.2 advantage

4.3 Comparison and analysis of results…………………………………………………………..67
4.4 Ideas of improving results……………………………………………………………………67
4.5 Comparison with the General Pickup and Delivery Problem……………………………..68

5
References……………………………………………………………………………………69

6 Appendixes…………………………………………………………………………………..70
6.1 program of constructive methods
6.2 program of modified methods
6.3 final algorithm

List of figures

2-1-1……………………………………………………………………………10
2-1-2……………………………………………………………………………10
2-1-3……………………………………………………………………………10
2-1-4……………………………………………………………………………11
2-3-1……………………………………………………………………………17
2-3-2……………………………………………………………………………17
2-3-3……………………………………………………………………………19
2-3-4……………………………………………………………………………19
2-3-5……………………………………………………………………………20
3-1-1……………………………………………………………………………21
3-1-2……………………………………………………………………………23
3-1-3……………………………………………………………………………23
3-1-4……………………………………………………………………………24
3-1-5……………………………………………………………………………26
3-1-6……………………………………………………………………………27
3-1-7……………………………………………………………………………28
3-1-8……………………………………………………………………………31
3-1-9……………………………………………………………………………31
3-1-10…………………………………………………………………………..32
3-1-11…………………………………………………………………………..32
3-1-12…………………………………………………………………………..33
3-1-13…………………………………………………………………………..35
3-1-14…………………………………………………………………………..36
3-1-15…………………………………………………………………………..39
3-1-16…………………………………………………………………………..40
3-1-17…………………………………………………………………………..42
3-1-18…………………………………………………………………………..43
3-1-19…………………………………………………………………………..44
3-1-20…………………………………………………………………………..47
3-1-21…………………………………………………………………………..47
3-1-22…………………………………………………………………………..49
3-1-23…………………………………………………………………………..49
3-1-24…………………………………………………………………………..50
3-1-25…………………………………………………………………………..51
3-2-1……………………………………………………………………………52
3-2-2……………………………………………………………………………54
3-2-3……………………………………………………………………………55
3-2-4……………………………………………………………………………56
3-2-5……………………………………………………………………………57
3-3-1……………………………………………………………………………65
3-3-2……………………………………………………………………………65
4-4-1……………………………………………………………………………68
4-5-1……………………………………………………………………………68

List of tables

2-1-1…………………………………………………………………………………....11
3-1-1……………………………………………………………………………………25
3-1-2……………………………………………………………………………………25
3-1-3……………………………………………………………………………………25
3-1-4……………………………………………………………………………………26
3-1-5……………………………………………………………………………………29
3-1-6……………………………………………………………………………………30
3-1-7……………………………………………………………………………………30
3-1-8……………………………………………………………………………………37
3-1-9……………………………………………………………………………………38
3-1-10…………………………………………………………………………………..45
3-1-11…………………………………………………………………………………..45
3-1-12…………………………………………………………………………………..46
3-2-1……………………………………………………………………………………52
3-2-2……………………………………………………………………………………53
3-3-1……………………………………………………………………………………59
3-3-2……………………………………………………………………………………60
3-3-3……………………………………………………………………………………60
3-3-4……………………………………………………………………………………60
3-3-5……………………………………………………………………………………61
3-3-6……………………………………………………………………………………61
3-3-7……………………………………………………………………………………62
3-3-8……………………………………………………………………………………62
3-3-9……………………………………………………………………………………62
3-3-10…………………………………………………………………………………..63
3-3-11…………………………………………………………………………………..64

Abstract

Along with the development of the society and the progress of the science, people around the world are devoting themselves to explore the better and better approaches to deal with all kinds of problems we are facing in the real life or our generations may confront to in future.

Nowadays, scientists are no more focusing on solving problems temporarily. They pay more close attention to adopt right and better ways and do their best to overcome problems permanently. This sort of the aspiration does not only belong to the developed countries. Since the explored and good methods have stimulated large batch of developed and developing countries, even the whole world attempt to apply the scientific means to solve problems effectively. Thus, the life of human being cannot go ahead well without science. Meanwhile, when we use the scientific way to help us settle problems, we also need to investigate right resolved approaches audaciously and continuously.

In the main content of this thesis, we will meet a new kind of transportation problem, which is a pickup and delivery problem with hub reloading at the central depot. In this pickup and delivery problem, every pickup source has a corresponding delivery terminations. We set each request includes the relative pickup and delivery actions and those two actions cannot be handled by the same vehicle. In other words, the pickup part and delivery part are separated in this project. My idea of resolving this problem can be described briefly as the way that: firstly, consider the pickup and delivery parts as two vehicle routing problems. Secondly, connect the relative pickup and delivery actions in each request under the time constraint. Along with the steps, I have displayed the relevant mathematical model. The goal is to find short routes and make every request to be finished in a limited time period are my work.

Key words
vehicle routing problem, pickup and delivery problem, mathematical model, heuristics, Insert_By_Distance, Sweep_By_Angle,

1 Introduction

In this section, a short introduction will be described. In the part of motivation, the development of transportation problem and the terminal purpose what people expect to touch will be given briefly. In order to guide readers go through this article and know the thought of composing methods smoothly, a mathematical model of vehicle routing problem will be described also. At the end of this section, the description of the current discussing problem will be introduced.

1.1 Motivation

In this thesis, a type of transportation problem will be presented. The purpose for doing such popular topic is the reason that transportation plays a significant role in the economy of most developed nations. For example, a National Council of Physical Distribution Study [1978] [1] estimated that transportation consumption occupied the rate of 15% of the U.S. gross national product. This economic importance has motivated both private companies and academic researchers to vigorously pursue the use of operation research and management science to improve the efficiency of transportation.

Various modes of transportation exist anywhere, such as the airlift, railway, water transportation, land transportation and so on. In different transportation fields, companies and researchers focus on different aspects in transportation problem. In the airlift, people pay primary attention to the arrangement of the crew scheduling. In the ordinary land transportation, the efficient use of a fleet of vehicles which must take some stops to pickup and delivery productions or passengers are concerned. The problem requires one to specify which customers should be delivered by each vehicle. And in what order so as to minimize total cost subjecting to a variety of constraints such as vehicle capacity and delivery time constraints. Those are also the content of this thesis that will be discussed later in more details.

This thesis is a pickup and delivery with hub reloading problem. This problem is a generation of the well-known vehicle routing problem, which is a generalization of the traveling salesman problem. The traveling salesman problem can be explained in short as [7] “A traveling salesman wants to visit each of a set of towns exactly once, starting from and returning back to his home town.” The traveling salesman problem can be seen as a trip. One of his problems is to find a shortest route for such trip.

The vehicle routing problem may be seen as the problem composed with some traveling salesman problems. It can be described as follows: given a fleet of vehicles with uniform capacity, a common depot, and several customer demands, finds the set of routes with overall minimum route cost which serve all the demands.

The characters of vehicle routing problem accord with the qualifications of the thesis. Because in each part of the project in this thesis, a fleet of vehicles without fixed numbers are located at the central depot, all the vehicles can only work from 6:00 in the morning to 22:00 in the evening and every vehicle has been fixed the same capacity.

Since the origin of our problem is the vehicle routing problem, there is a mathematical model of vehicle routing problem [2] displayed.

$K =$	number of vehicles in the fleet
$N =$	number of customers to which transportation must be made. Customers are indexed form 1 to n and index 0 denotes the central depot
$Q_k =$	the capacity of every vehicle
$d_i =$	the demand of load from every customer
$C_{ij} =$	cost of direct travel from i to j

The vehicle routing problem is to determine K vehicle routes. Every route has to start from the central deport with visiting a subset of N customers in a specified sequence, and then goes back to the central deport again. In every determined route, the total demand of a subset of customers should not exceed the vehicle capacity. All the routes should be determined to shorten the total travel distance.

Objective function: *minimize* $$\sum_{k\in K}\sum_{i\in N, j\in N} C_{ij} X_{ij}^{k} \quad (1)$$

Subject to: $$\sum_{k\in K}\sum_{j\in N} X_{ij}^{k} = 1, \quad \forall i \in N \quad (2)$$

$$\sum_{i\in N} d_i \sum_{j\in N} X_{ij}^{k} \le Q_k, \quad \forall k \in K \quad (3)$$

$$\sum_{j\in N} X_{oj}^{k} = 1, \quad \forall k \in K \quad (4)$$

$$\sum_{i\in N} X_{ih}^{k} - \sum_{j\in N} X_{hj}^{k} = 0, \quad \forall h \in N, \forall k \in K \quad (5)$$

$$\sum_{i\in N} X_{i,n+1}^{k} = 1, \quad \forall k \in K \quad (6)$$

$$\sum_{k\in K} X_{oj}^{k} - \sum_{k\in K} X_{jo}^{k} = 0 \quad (7)$$

$$X_{ij}^{k} = \begin{cases} 1 & \text{if vehicle } k \text{ drives from node } i \text{ to node } j \text{ directly} \\ 0 & \text{otherwise} \end{cases}$$

Note:

(2) Each customer must be assigned to exactly one vehicle

(3) No vehicle can serve more customers than its capacity permission

(4) For each vehicle, it can only start from depot to any nodes once

(5) For any two connected nodes, how much load comes out from the previous node equals to how much load enters into the forward node

(6) A flow constraint requiring that each vehicle k leaves node 0 once, leaves node i, if and

only if it enters that node, and returns to node $n+1$

(7) How many vehicles leaves the depot to serve customers outside should go back to the depot with the same number

1.2 Description of the working task and purpose

In the given task, we are required to consider a number of pickup-and-delivery orders. Under this process, the goods in each order should be picked up at one place called the source node and transported to another place denoted the terminal node. In this problem the goods do not travel directly from their sources to their terminations, instead they are firstly transported from the sources to a central depot where they are reloaded to another vehicle and then driven to the terminations. At the source, the destination and at the depot there are some costs associated with loading, moving or off-loading the goods. Each vehicle is bounded by a total driving time. The problem here is to plan the routes for the vehicles so that all goods get delivered but at a minimum transportation cost.

In association with the project proposal, a file containing realistic but not real life data is generated and supplied by the Danish company, Transvision.

2 Routes with Central Depot

In the context underlying, the given problem will be described carefully. Not only is the detailed train of thought about solving this problem included, but also the mathematical formulation fromed for solving the problem.

2.1 Problem formulation

The problem going to be solved later is a generation of ordinary vehicle routing problem. Comparing with the general pickup-and-delivery problem, let us see what the differences between the general case and our facing case are.

- ✧ *general case: vehicles have to transport goods from origins to destinations without transshipment at intermediate location*
- ✧ *our facing case: vehicles have to transport goods from origins to destinations with reloading at the single intermediate depot*

Obviously, the character of our facing case is this practical transportation problem composed of two parts. The first part includs a set of source/pickup nodes and one central depot. And the second part is composed of the same number of corresponding terminal/delivery nodes and the same central depot.

We may say that the general pickup-and-delivery problem is a vehicle routing problem in which either all the origins or all the destinations are located at the depot. Then the case we are facing now can be understood as an order combined with two vehicle routing problems. Such order includes two parts: the first one is to set all the destinations located at the depot, and the second one is to set all the origins located at the depot.

If we use three figure to show the differences of general case and our facing case visibly, then

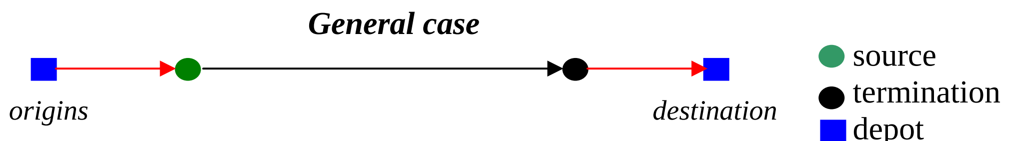


Figure 2-1-1

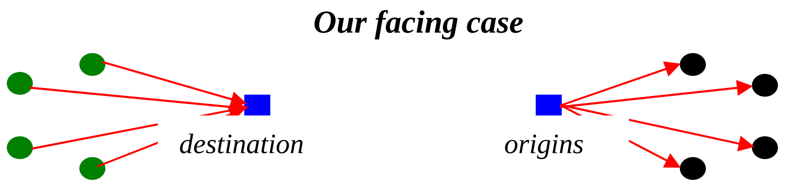


Figure 2-1-2

The aim of our facing problem is to adopt feasible routes. By those routes, a fleet of vehicles pick up plenty of goods from their source points to the depot, transship the goods to another fleet of vehicles, and then delivery them from the depot to their corresponding terminal points in a fixed working time period from 06:00 to 22:00.

Briefly speaking, the relevant information of the system can be described as a simple figure below
.

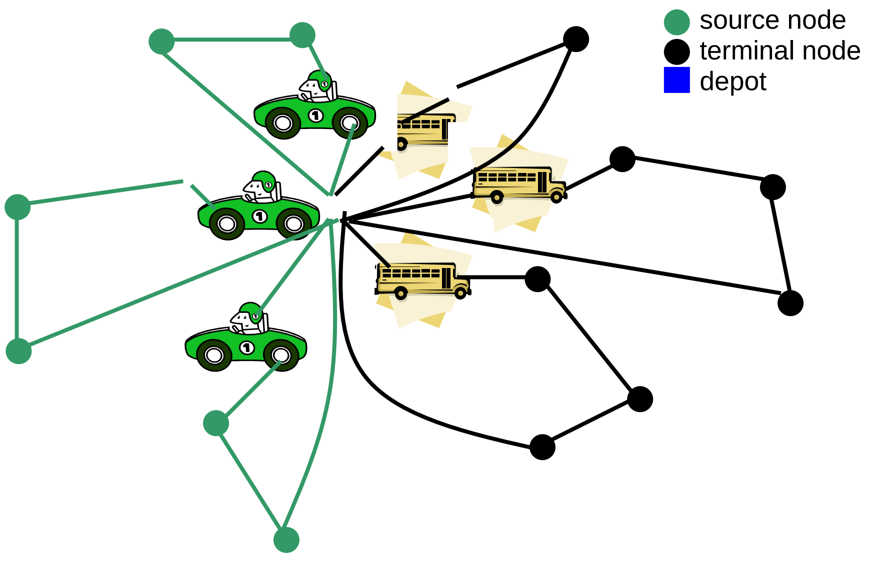


Figure 2-1-3

Evidently, from the picture above, we can see that there is a small light blue rectangle, which named as the central depot. Beside the depot, there are two sorts of nodes with

different colours. The light green ones are source nodes with goods, a fleet of green cars coming from the depot have to visit those nodes, pick up goods from them and go back to the depot. The black nodes are the places where the goods from the light green nodes have to be sent, we call them terminal nodes. At the end of this process, we can see that all the fleets of vehicles will come back to the depot again.
It is not difficult to find that each route in either patrs is only composed with source nodes or terminal nodes. The way of composing route means no route can include the source and terminal nodes together without passing by the central depot. When we plan to send some goods to their terminations, we must be sure those goods have been brought to the central depot first.

After getting the idea of picking up and delivering goods, how can we save the amount of the total cost obeying the rule of transporting those goods? Since the objective function is to find the shortest distance, the minimum cost of the entire process is decided by the whole travelling route length. And there is no constraint to combine several sorts of goods together on the same vehicle. Without exceeding vehicle capacity, we expect each vehicle can get as much as possible goods.
Comparing with normal transportation projects, this case is a little different except for the characters of problem per se. It is also a practical task. Despite purely focusing on the algorithm search, the Danish company, Transvision, has made a great deal for this case. They support amount of simulated data. All the data are separated into five weekdays.

There are 247, 214, 199, 229 and 212 orders assigned to those five weekdays from the first to the fifth respectively. For each order, there are some relevant information including the ID of order, the locations of source and terminal nodes, the week day and the corresponding demand. All these information can be seen clearly from the table below

ID	To X	To Y	From X	From Y	Week day	Dem and
1	538181	6086484	720415	6176264	1	16

Table 2-1-1

The value of X and Y are the coordinate value of the central depot on plane.
If we use the points with green colour to show the source nodes, the points with black colour to show the terminal nodes and the red circle to denote the position of the central depot, the distribution of all the data can be seen in the picture following

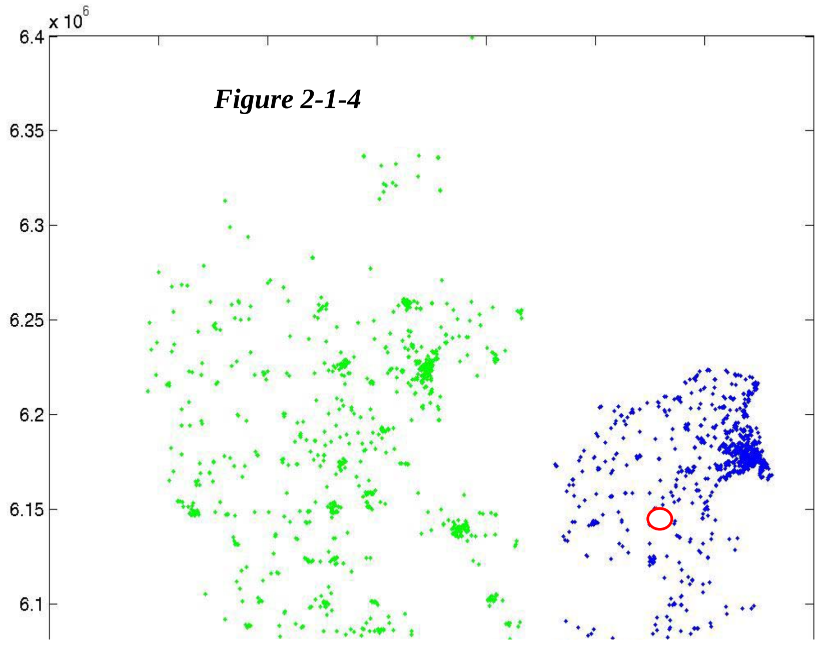


Figure 2-1-4

From the figure, the real amazing thing is that the picture filled with all the data in five workdays is the outline of Danish map. This is very interesting and inventive idea of making the simulated data.

Knowing the coordinate values of each point, it is not a problem for us to obtain the length between any two nodes. By applying the right angle equation, if we set the coordinate values of the first and second points are (x_1, y_1) and (x_2, y_2) separately, then the length between them can be calculated as

$$Length_{node1,node2} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

So far, we have got enough basic information about the problem. As I have mentioned, we set the pickup and delivery actions as vehicle routing problems. In this thesis, I focused on those two separate vehicle routing problems to get feasible routes. Then I connected relative pickup and delivery actions and make them to be an entirety. With such idea, a mathematical formulation for this problem has been formed also.

2.2 Mathematical formulation

The mathematical model is based on two parts: the part composed source nodes with a central depot and the part composed terminal nodes with the same central depot. The objective function of this model is to obtain the minimum transportation route length.

The data are:

- Sets (indices):
  - previous Nodes: i
  - current Nodes : j
  - vehicles : k
  - set of nodes for e . N
  - set of Vehicles in the first part : K_1
  - set of Vehicles in the second par K_2

- Data:
  - distance between two nodes : d_{ij}
  - demand from each node : R_i
  - Capacity of each vehicle : Q
  - Average velocity of each vehicle : v
  - Starting time for transporting : T_{start}
  - Ending time for transporting : T_{end}

- Variable:
  - Time of each vehicle : t_k
  - Earliest starting time of each vehicle: t_k^+
  - Latest starting time of each vehicle: t_k^-

- Binary variable:

$$X_{ij}^{k} = \begin{cases} 1 & \text{if vehicle } k \text{ drives from node } i \text{ and node } j \text{ directly} \\ 0 & \text{otherwise;} \end{cases}$$

The Model

The model is:

$$\text{Minimize } \sum_{k\in K_1}\sum_{i\in N}\sum_{j\in N} d_{ij}X_{ij}^{k} + \sum_{k\in K_2}\sum_{i\in N}\sum_{j\in N} d_{ij}X_{ij}^{k} \tag{1}$$

St.

$$\sum_{k}\sum_{j\in N} X_{ij}^{k} = 1 \quad \text{for } \forall i \in N,\ \forall k \in K_1 \text{ or } \forall k \in K_2 \tag{2}$$

$$\sum_{i\in N} R_i \sum_{j\in N} X_{ij}^{k} \le Q \quad \text{for } \forall k \in K_1 \text{ or } \forall k \in K_2 \tag{3}$$

$$\sum_{i\in N} X_{ih}^{k} - \sum_{j\in N} X_{hj}^{k} = 0 \quad \text{for } \forall h \in N, \forall k \in K_1 \text{ or } \forall k \in K_2 \tag{4}$$

$$\sum_{j\in N_1} X_{0j}^{k} = 1 \quad \text{for } \forall k \in K_1 \text{ or } \forall k \in K_2 \tag{5}$$

$$\sum_{i\in M_1} X_{i,n+1}^{k} = 1 \quad \text{for } \forall k \in K_1 \text{ or } \forall k \in K_2 \tag{6}$$

$$t_k^{+} \ge T_{start} \quad \text{for } \forall k \in K_1 \tag{7}$$

$$t_k^{-} + t_k \le T_{end} \quad \text{for } \forall k \in K_2 \tag{8}$$

$$t_{k\in K_1}^{-} + t_{k\in K_1} \le t_{k\in K_2}^{+} \tag{9}$$

$$t_{k\in K_2} + t_{k\in K_2} \le T_{end} - T_{start} \tag{10}$$

$$t_{k\in K_1} = \sum_{i\in M_1}\sum_{j\in N_1} d_{ij}X_{ij}^{k} / v + \left(\sum_{i\in M_1} R_i \sum_{j\in N_1} X_{ij}^{k} + 10 * \sum_{i\in M_1}\sum_{j\in N_1} X_{ij}^{k}\right)/60 \quad \text{for} \begin{cases} \forall k \in K_1 \\ \text{or } \forall k \in K_2 \end{cases} \tag{11}$$

Note:

(1) the objective function is calculated to get the shortest distance of the whole process since the cost here depends on the entire distance;

(2) each node can be visited by only one vehicle;

(3) each vehicle cannot transported more pallets than its capacity constraints;

(4) flow constraints, which means how many come into one node then how many come out;

(5) there is only one vehicle for each route;

(6) there is only one current node can be connected to only one previous node;

(7) the earliest staring time should later than the starting time of the system;

(8) the transportation started at the latest starting time in the second part should be finished earlier than the ending time of the system;

(9) the transportation started at the latest starting time in the first part should be finished earlier than the earliest starting time of the second part;

(10) the time spent in both parts should less than the total time bounded to each vehicle;

(11) time spent in the route of the first part or in the second part

2.3 Literature review

Some basic information and the mathematical formulation for handling the solving case have been discussed. But the problems of obtaining routes and connecting them still have not been solved. In order to deal with these problems, I have referred some pertinent literatures.

Laporte and Semet [3] said "Several families of heuristics have been proposed for the vehicle routing problem. All those can be widely classified into two main classes: classical heuristics developed mostly between 1960 and 1990, and meta-heuristics which have been grown since the last decade. But most standard construction and improvement procedures in use today belong to the first class."

The Heuristics are comparatively more popular than the meta-heuristics, mainly due to the characters of those two methods:

Heuristics	● Perform a relative limited exploration of search space and they can typically produce good solutions with modest computing time;
Meta-heuristics	● Emphasis on performing a deep exploration of the most promising regions of the solution space, but it is not so easy to cope with as people imagined
	● The implements of their sophistications relay on too much computing time.

So far, there is some guy has calculated good results for mostly 100 customers in VRP. Comparing with the solved issue, the number of customers in our problem are nearly more than 200 in every weekday. Being restricted with plenty of customers and expecting to get feasible results probably. I decided to apply heuristics to resolve the problem in this thesis.

Some classical VRP heuristics can be broadly classified into three categories.

Classical VRP heuristics

- *constructive heuristics*
  gradually build a feasible solution while keeping an eye on solution cost but do not contain an improvement phase per se

- *two-phase heuristics*
  - *cluster-first, route-second*
  - *route-first, cluster-second*

  the problem is decomposed into its two natural components: clustering of vertices into feasible routes and actual routes construction, with possible feedback loops between the two stages

- *improvement methods*
  attempt to upgrade any feasible solution by performing a sequence of edge or vertex exchanges within or between vehicle routes.

Before starting any practical action to the problem, I have referred some useful articles in order to find suitable and feasible constructive methods. Some literatures are necessary to be introduced.

Most of the previous works in this section are about the reason why I chose the heuristics methods. There are a lot of famous heuristics explored for solving VRP. Such as:

In the article written by Laporte and Semet [3], the main classical heuristics for VRP have been reviewed. They are constructive methods, two-phase methods and improvement heuristics. Because my aim was to find suitable constructive methods, I mainly paid my attention to the section of constructive methods in this article. There are two main techniques are used for constructing VRP solutions. The first one is to use a saving criterion to merge existing routes. And the second one uses an insertion cost to assign vertices to vehicle routes gradually.

The first constructive method comes from the idea of Clarke and Wright [4]. This algorithm is the most well-known heuristic for the VRP. The notion of saving can be explained as when two routes $(0,\ldots,i,0)$ and $(0,j,\ldots,0)$ can be merged into a single route $(0,\ldots,i,j,\ldots,0)$. A distance saving $s_{ij} = c_{io} + c_{oj} - c_{ij}$ is generated. The second insertion method comes from two algorithms based on sequential insertions. The first, due to Mole and James [5], expands one route at a time. The second, due to Christofides, Mingozzi and Toth [6], applies in turn sequential and parallel. Both of these insertion algorithms are applied to problem with an unspecified number of vehicles.

- ***Phase one: sequential route construction***

Step1 Set a first route index $k = 1$

Step2 Select any unconnected point p_k to initialize route k. Calculate $\delta_i = c_{0i} + \lambda c_{ii_k}$ for every point i

Step3 Let $\delta_{i^*} = \min_{i \in S_k}\{\delta_i\}$. S_k is the set of points which can be feasibly inserted into route k . Optimize route k with the feasible points by using 3-opt algorithm. Repeat step 3 until mo more point can be assigned to route k

Step4 If all points have been inserted into route, stop. Otherwise, set $k = k + 1$ and go to step2

➢ ***Phase two: parallel route construction***

Step5 Initialize k routes and $R_t = (0, i_t, 0)(t = 1,..., k)$. k is the number of routes obtained from phase one. Let $j = \{R_1,..., R_k\}$

Step6 For each route $R_t \in J$ and for every point i which has not been connected to route. Compute $\varepsilon_{ti} = c_{0i} + \mu c_{iR_t}$ and $\varepsilon_{t^*i} = \min_{R_t \in J}\{\varepsilon_{ti}\}$. Connect point i to route R_t and repeat step6 until all points have been connected to routes

Step7 Take any route $R_t \in J$ and set $J := J / \{R_t\}$. For every point i connected to route R_t , compute $\varepsilon_{t'i} = \min_{R_t \in J}\{\varepsilon_{ti}\}$ and $\tau_i = \varepsilon_{t'i} - \varepsilon_{ti}$

Step8 If point i satisfies $\tau_{i^*} = \max_{i \in S_t}\{\tau_i\}$. S_t is the set of points which have been inserted in to route R_t . Optimize route R_t using 3-opt algorithm. Repeat step8 until no more points can be inserted in to route R_t

Step9 If $|J| \neq \phi$, go to step6. Otherwise, if all the points are in routes, stop. If uninserted point exists, create a new route starting from the step1 of phase one

Since the heuristics of saving and insertion play an import role among the constructive methods for VRP, I have referred a relevant article presented by Junger, Reinelt and Rinaldi [7]. This paper has introduced the methods of saving and insertion very carefully and basically.

✧ *Insertion Heuristics*

It starts with cycles visiting only small subsets of the nodes and then extends these cycles by inserting the remaining nodes until all nodes are inserted and a circle is found

```
Procedure INSERTION
(1)  Select a starting circle on $n$ nodes $n_1, n_2, ... n_n$ ($n \geq 1$) and set $W = N / \{n_1, n_2, ... n_n\}$
(2)  As long as $W \neq \phi$ do the following:
     (2.1) Select a node $j \in W$ according to some criterion;
     (2.2) Insert $j$ at some position in the cycle and set $W = W / \{j\}$
```

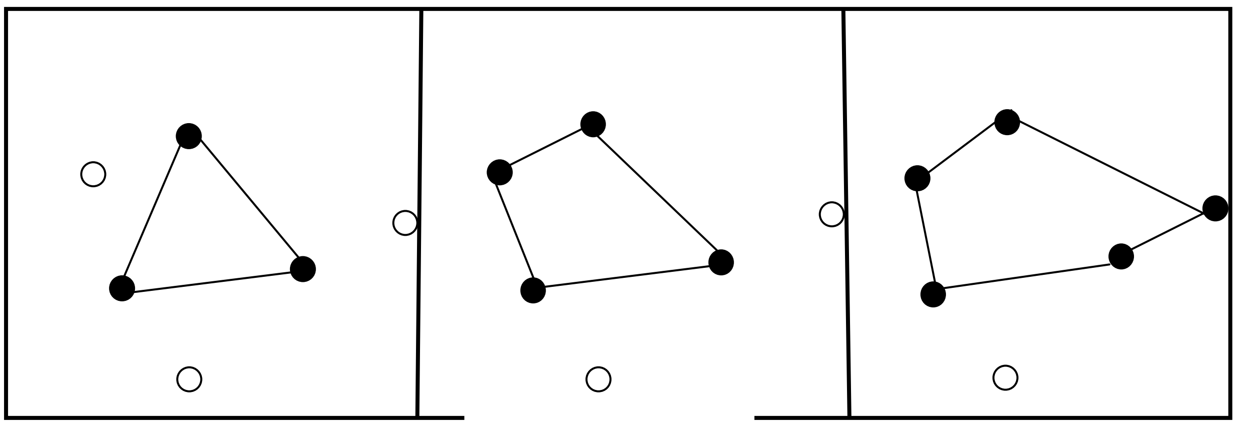


Figure 2-3-1

✧ *Saving Heuristics*

This heuristic was originally developed for vehicle routing problems. It successively merges sub tours to eventually obtain a single one, if the vehicle routing problem is supposed to be special that it involves only one vehicle with unlimited capacity.

Procedure SAVING

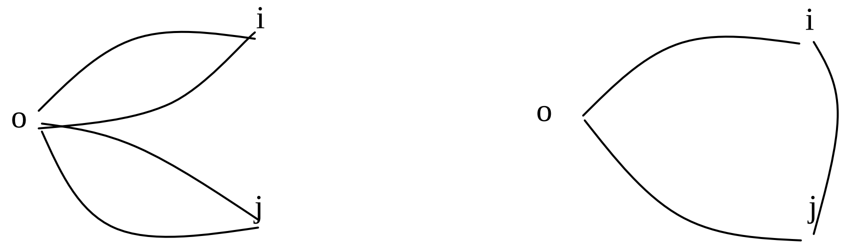
(1) Select a base node $z \in V$ and set up the $n-1$ sub tours (z, v), $v \in V / \{z\}$ consisting of two nodes each;

(2) As long as more than one sub tours is left perform the following steps:

(2.1) For every pair of sub tours T_1 and T_2 compute the saving that is achieved is they are merged by deleting in each of them an edge to the base node and connecting the two open ends ;

(2.2) Merge the two sub tours which provide the largest savings.

Merging process:



$s_1 = d_{oi} + d_{io} + d_{oj} + d_{jo}$ $\quad$ $s_2 = d_{oi} + d_{ij} + d_{jo}$

$s_{saving} = s_1 - s_2 = d_{io} + d_{oj} - d_{ij}$

If $s_{saving>0}$ *, then some distance will be saved*

Otherwise, merging process failed

Figure 2-3-2

Beside literatures above, there are also other four articles supported me some great idea when I was in the trouble of researching goods methods to solve problem.

In the article written by Lau and Liang [8], a two-phase method for solving pickup and delivery problem with time windows has been presented. In the this paper, two new constructive methods have a relationship to my structure of presenting constructive methods. This article makes me realize that the algorithms adopted by me are reasonable.

Two constructive methods discussed by Lau and Liang [8] can be adopted to show as

- ***Insertion Heuristic***

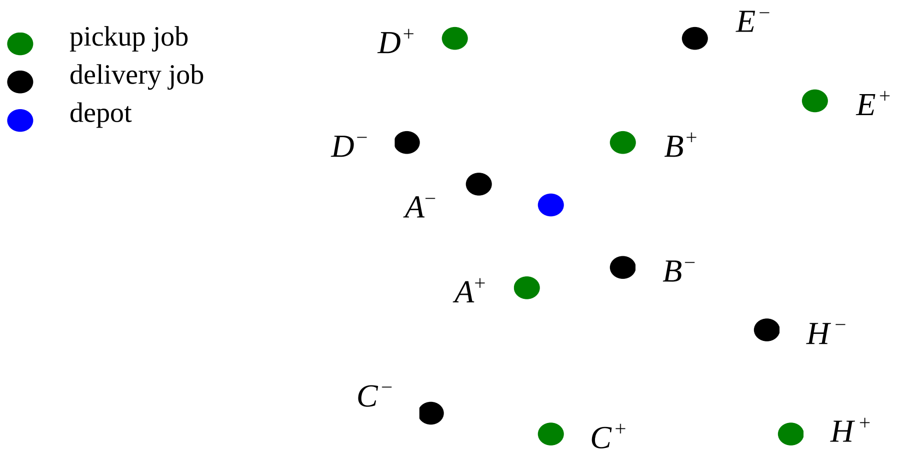
1. let all vehicle have empty routes
2. let L be the list of unassigned requests
3. take a job pair v in L
4. insert v in a route at a feasible position where there is the least increase in cost
5. remove v from L
6. if L is not empty, go to 3

- ***Sweep Heuristic***

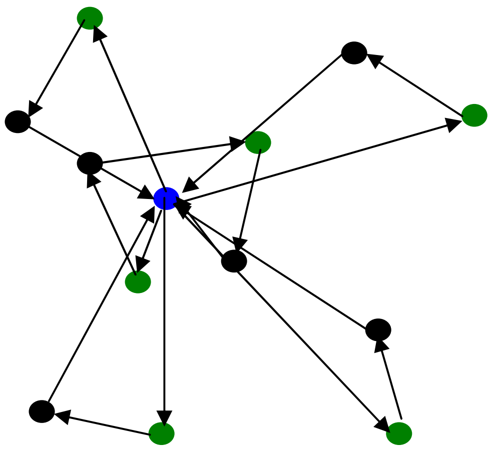
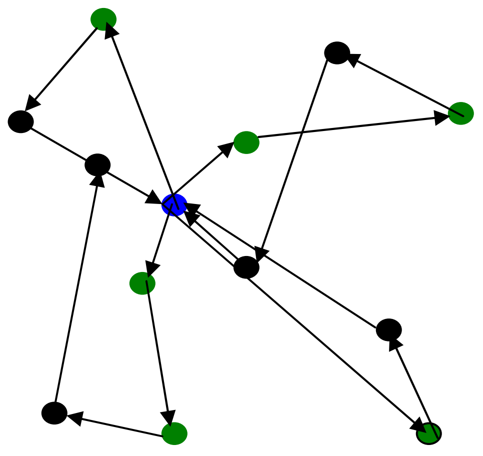
1. let O be a site from which vehicles leave, and let A (different from O) be another location, which serves as a reference.
2. sort pickup jobs by increasing angle $\angle AOS$ where S is the job location. Put result in a list L
3. pick a pickup job in L with location I and its delivery job with location J and create an new route with this job pair
4. until no more jobs can be added to the route, do

   a. if there are uninserted pickup jobs located in the sector $\angle IOJ$, insert the pair that is best feasible. Otherwise, insert an uninserted pickup and delivery job pair, in which the pick up job is at location K , where $\angle JOK$ is smallest and all the constraints are respected

   b. remove this pickup job from L
5. if L is not empty, go to 3



Example PDPTW instance

Solution using Insertion Heuristic
Solution using Sweep Heuristic

Figure 2-3-3

The second paper was written by Fisher and Jaikumar [9]. They have introduced the readers a good assignment heuristic to solve VRP. The main idea they composed to form routes in a feasible way, can be described into two steps. Firstly, they set several furthest points as seed customers among all the points and connected those seed customers directly to the depot to make routes. Secondly, they inserted other unconnected points into the formed routes to make feasible and short routes, under the constraints such as the vehicle capacity.

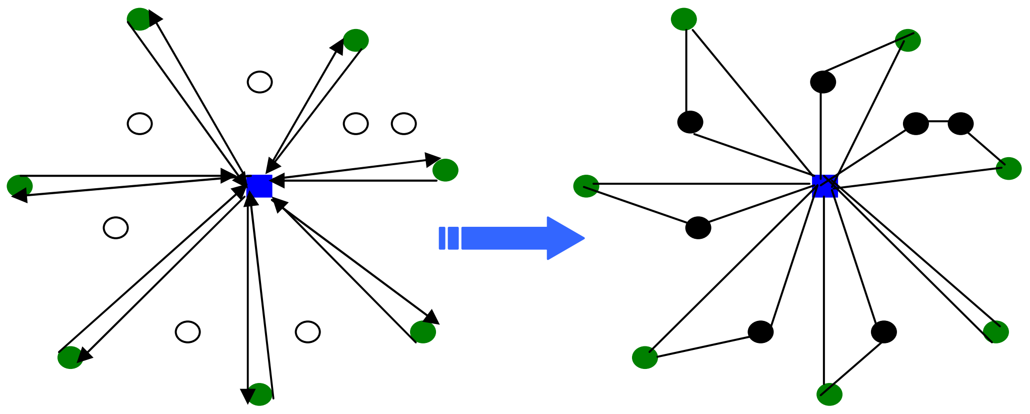


Figure 2-3-4

The figure above describes the process of forming routes in article Fisher and Jaikumar [9]. Their idea absolutely inspirited me since my ideas of composing method of Sweep_By_Angle was produced from it. But I did not try to find seed customers, because my project has been given too many points and I could not have any way to define seed customers from them. I chose to form routes by sweeping the tangent value of each point. The more details about Sweep_By_Angle will be descried in the next section.

The problem handled in the third one presented by Røpke [10] is general pickup and delivery problem with time windows. Røpke has systematically introduced some well-known methods for solving pickup and delivery problem with time windows. He has also implemented most of them and made a conclusion to compare those methods. I have made some comparisons between the problem in such paper and mine. When I went through this article, I learned more about PDP. Røpke's way of researching algorithms helped me to modify my methods in a right and better way.

Finally, a thesis coming from a Finnish guy, Braysy [11] displayed plenty of feasible and optimal algorithms. The content of this report almost crowns all the algorithms that I have mentioned above. There is very useful approach used for optimizing routes arose my attention when I read it, which is Ejection Chain. The method of EC gave me a good direction when I was dealing with the modification part of my constructive methods.

The method of EC can be briefly described as a procedure of moving points among routes. This is not an unfamiliar method in VRP field. People use this method to eliminate routes when they do optimization. The basic idea is to pick up first some customers c_i from route r_i and insert some other customers c_j currently served by route r_j into the partial route r_i. If the insertion is possible, the customers c_i have to be inserted into another route r_k, $r_k \neq r_i$. If all insertions are feasible, the ejection chain is completed and next chain will start. When all the possibility of using EC has been tried and the last one customer can be inserted its neighbour completed route, the procedure of EC is finished. The process of using EC can be depicted by figures

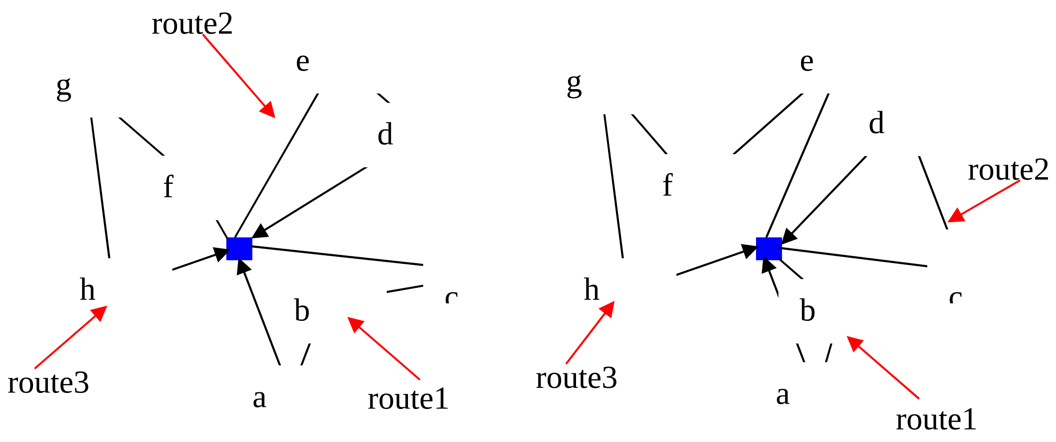


Figure 2-3-5

Ejection chain. There are three routes in the figures above. The figure on the left shows the routes will be reformed and the figure on the right shows the new routes formed from the routes in the left figure. Focusing on the figure on the left, we assume to move point c from route1 to route2. If point c can not be inserted into route2 feasibly by some limitations such as time or vehicle capacity, we could first move points from route2 to other route except route1. Assuming point picked up from route 2 is point e, three new routes got are showing in the right figure. This is the procedure of applying ejection chain. For the other points i and j, the same way can be used until no route can be eliminated.

3 Method Discussion

In order to find out feasible solution in this project, I divided the whole process of solving problem into two steps. The first step was responsible to form feasible routes both in pickup and delivery parts. The second one was to connect relative routes formed in the first step. Since there was time bounded to each vehicle, some trouble would be met. The total time of two relative routes might exceed the bounded time in the process of the second step. Once the problem happened, such relative routes had to be modified. But the way of overcoming the trouble will be described later.

3.1 Forming routes

This section is the process of getting short and feasible routes. In this section, I will first describe two basic constructive heuristics: Insert_By_Distance and Sweep_By_Angle, which have helped me get the original solutions. Then I will discuss the modifications based on those two constructive methods.

3.1.1 Constructive heuristics

From the discussion of classical heuristics above, I know that there are mainly two well-known heuristics for solving VRP, *saving* and *insertion* algorithms. Since the objective function of my case has been given more than 200 customers, it will not be a good way to adopt the saving heuristic to form routes. The process of getting saving and comparing them are a huge and complicated works.

By considering the shortage of using saving heuristic in this thesis, I chose the insertion method as my beginning and I made the method of Insert_By_Distance. The insertion method has helped me get feasible solutions but there also some dissatisfied instances existed. For example, some points in the same route are far from each other, which make the route length to be longer as the figures shown below.

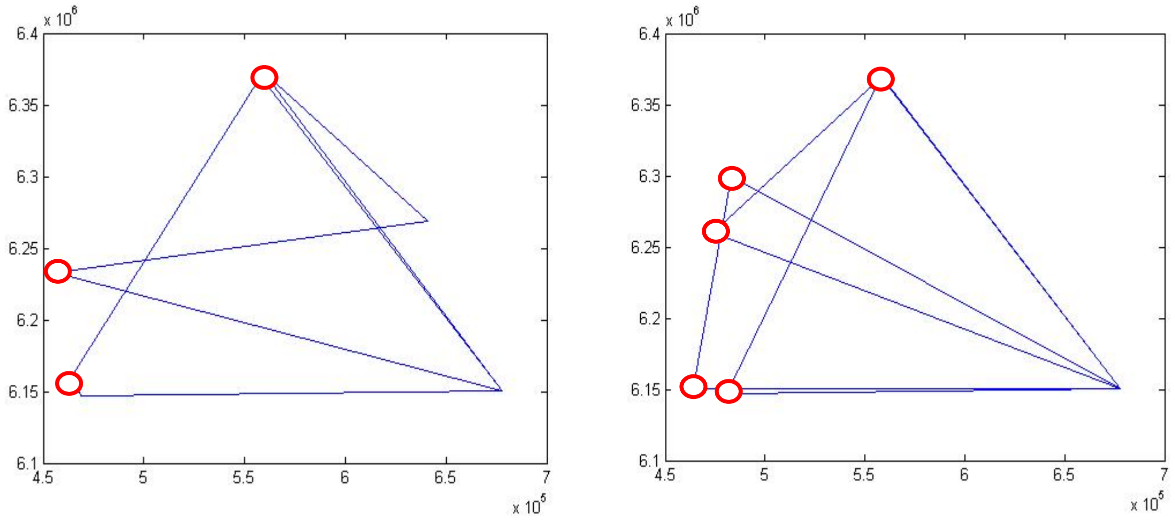


Figure 3-1-1

When I realized this problem, the idea of Fisher and Jaikumar [9] inspired me. Actually, it just gave me information that I should connect the points around the same place. If those points were connected, then the above phenomenon would not happen. According to this idea, I made the method of Sweep_By_Angle. But it was surprising that the results from Sweep_By_Angle were not as good as results from Insert_By_Distance. Therefore, I did some modifications.

Anyway, I will display the basic methods of Insert_By_Distance and Sweep_By_Angle in details firstly.

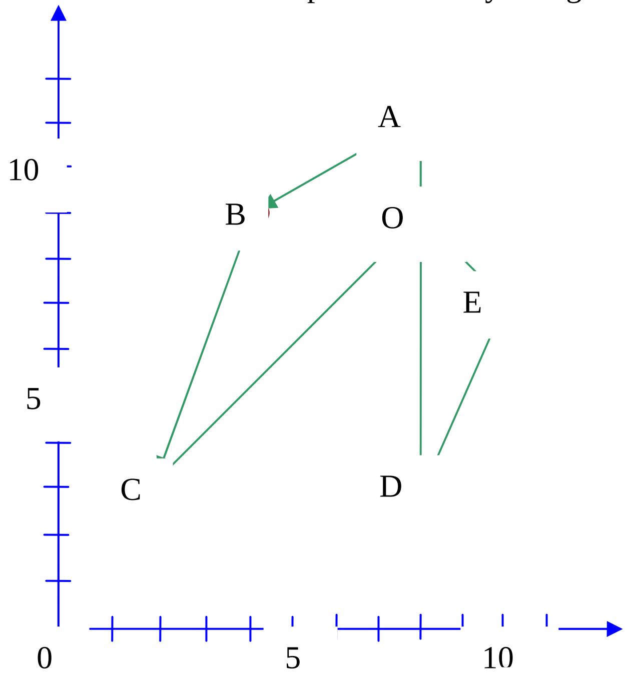
3.1.1.1 Insert_By_Distance heuristic

Since in every route, the vehicle has to start from the depot, visit its customers and go back to the depot at the end. We need build an original circle only with the depot, which means the depot can be seen as the starting point and also the ending in such circle. The next step of work is to insert the rest of the points into the original circle gradually in order to form route. Once we meet the instance that in some route the total demands of connected points exceed the vehicle capacity if we insert just one more point. We have to give up continuing inserting the last point but go back to the depot.
Every time in each circle, we insert point sequentially. And the unconnected point that needs to be inserted should be close-by the last connected point in the circle. Such method is called as Insert_By_Distance. The pseudo code of this method can be written as

```
Set:  $N = \{1,...,n\}$: all the points except for the depot;
      $k = 0$ : the ID of routes;
      $Route[k] = \phi$ ;
      $i = 0$;
      $selectedNode = 0$;
1  function Insert_By_Distance: all points
2        all points:  $n$
3        bool: finished = false
4        while not finished do
5              for( $j \in N$ )
6                    if ( $\sum_{i \in M} d_i \sum_{j \in N} x_{ij}^k \le Q$ )
7                          if (selectedNode == 0)
8                                selectedNode == j;
9                                shortestDistance = Distance(i,j);
10                               else
11                                     if (Distance(i,j) < shortestDistance)
12                                           selectedNode = j;
13                                                 shortestDistance = Distance(i,j)
14                 if (selectedNode != 0)
15                       N = N – {j};
16                       Route[k] = Route[k] + {j};
17                       i = j;
18                       selectedNode = 0;
19                 else
20                             $k++, Route[k] = \phi$ ;
21                             i = 0;
22                             selectedNode = 0;
23                     while ( $N != \phi$ )
```

When I was in the procedure of inserting points, I found a problem that when more than one point having the same qualification are waiting to be inserted, which one should be supposed to my choice?

In order to select the right points, I thought if we choose the point with more demand but give up the points with less demand when this problem exists, the rest of the points with less demand will have more chance to be inserted into other circles. And fortunately, the test and results have proved that my thought was reasonable.



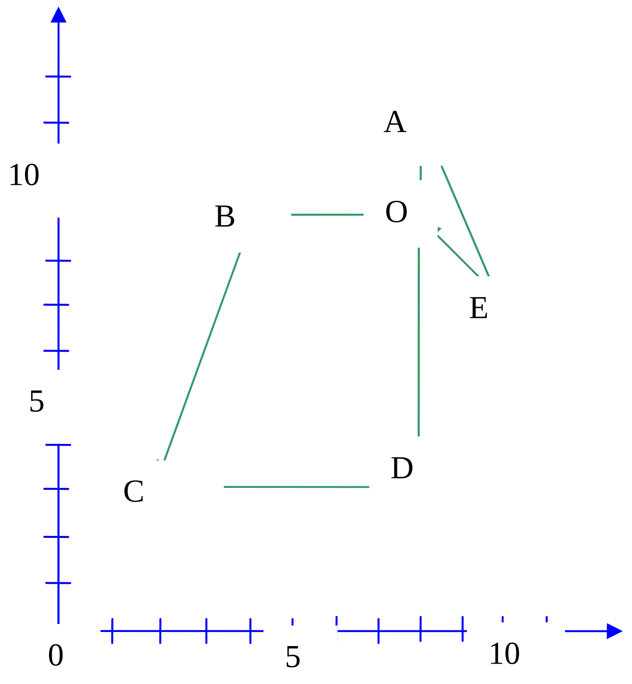
Demand: Capacity: 30

A:	12
B:	8
C:	7
D:	13
E:	16
O:	0

Route1: O, A, B, C, O
Route2: O, E, D, O

Length_Route1 = 21.28
Length_Route2 = 13.3
Total Length = 34.58

Figure 3-1-2



Demand: Capacity: 30

A:	12
B:	8
C:	7
D:	13
E:	16
O:	0

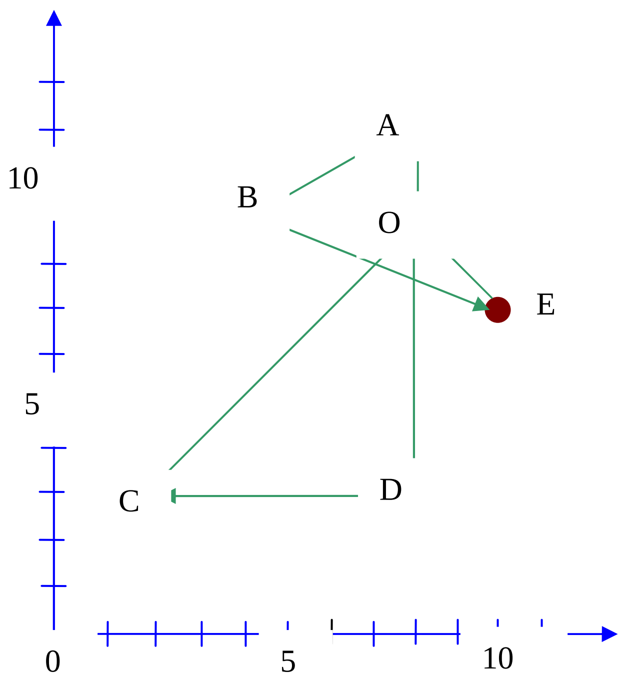
Route1: O, A, E, O
Route2: O, B, C, D, O

Length_Route1 = 9.3
Length_Route2 = 22.32
Total Length = 31.62

Figure 3-1-3

Two pictures above are a part of my test. In those two pictures I set all the same points with the same demand. In the first picture, point B and point E have the same distance to point A after point O and point A have been connected. Should I connect B to A? Or I should connect E to A?

First I connected the points with less demand. In this way I connected point B first. And I got the total route length in the first picture was 34.58. Later in the second figure, I chose to connect point with more demand first, point E instead of point B. And I got 31.62 for the total route length, which was less than the solution in the first figure.
I also considered other instance. What would happen if the information such as the demand of some points has been changed? In order to make a test, I changed the demand of some points leaving all the points at their original positions as the figure shown above. And I connected the point with more demand first again and got the total route length was 33.94, which was less than the first one again.



Demand: Capacity: 30

A:	12
B:	9
C:	7
D:	13
E:	8
O:	0

Route1: O, A, B, E, O
Route2: O, D, C, O

Length_Route1 = 15.62
Length_Route2 = 18.32
Total Length = 33.94

Figure 3-1-4

On the other hand, for the sake of insuring the validity of the connecting way, I tested it with the given data in the pickup part. When I did the test, I made three groups of solutions. Under the instance that when more than one point having the same qualification are waiting to be inserted:

- First group: select the point with more demand
- Second group: select the point with less demand
- Third group: select the point according to their ID order

Capacity = 33, results : route length

weekday	number of points	First group $\times 10^7$	Second group $\times 10^7$	Third group $\times 10^7$
Day1	247	**2.6603**	2.7011	2.7011
Day2	214	**2.3584**	2.5001	2.5001
Day3	199	2.1858	**2.1518**	2.2071
Day4	229	**2.5817**	2.6197	2.5904
Day5	212	**2.3298**	2.3680	2.3680

Table 3-1-1

Capcacity = 33, results : computing time
Unitage : millisecond

weekday	number of points	First group	Second group
Day1	247	969	**640**
Day2	214	609	**203**
Day3	199	1359	**735**
Day4	229	**391**	921
Day5	212	1047	**500**

Table 3-1-2

The tables above include the solution got from the original data. In the original data, the number of customers has not been changed. We can see that almost the better results got by selecting the points with more demand first, when there are more than one point has the same qualification waiting to be inserted. But the time spent is more than applying the way of selecting point with less demand.

Tables below include the solution under changing the data, which means only 100 customers will be tested every weekday.

Capacity = 33, number of customers = 100, results : route length

weekday	First group $\times 10^7$	Second group $\times 10^7$
Day1	**1.1305**	1.1305
Day2	**1.2229**	1.2229
Day3	**1.1424**	1.4039
Day4	**1.1521**	1.1521
Day5	**1.1977**	1.1977

Table 3-1-3

Capcacity = 33, number of customers = 100, results : computing time
Unitage : millisecond

weekday	First group	Second group
Day1	**93**	125
Day2	531	**422**
Day3	**94**	453
Day4	141	**93**
Day5	**78**	422

Table 3-1-4

Seeing the solution from the table, the smallest numbers in those tables are boldfaces. Not only the test but also the solutions got from data of pickup part are all proved that we should priority the points with more demand when more than one point having the same qualification are waiting to be inserted.

But as I said before, there is a bad phenomenon existing when we use the method of Insert_By_Distance. Because some points in the same route formed by Insert_By_Distance are far from each other, those points make the route length to be long.

3.1.1.2 Sweep_By_Angle heuristic

The method of Sweep_By_Angle was basically produced from the idea of Fisher and Jaikumar [9]. In their idea, they knew they should choose the furthest points among the total points as seed customers. And they inserted other unconnected points to form routes in the way that I have described previously in the section of reference.

They set the furthest points as seed customers, because the total number of points in their case was not as many as mine. When I met my case, I could not get any good idea to resolve my case applying the same way as theirs. The only thing what I could do was to do the best of my abilities to focus on the points around the same place and tried to form route by connecting them. In the method of Sweep_By_Angle, I found the sequential distribution of all the points in each weekday.

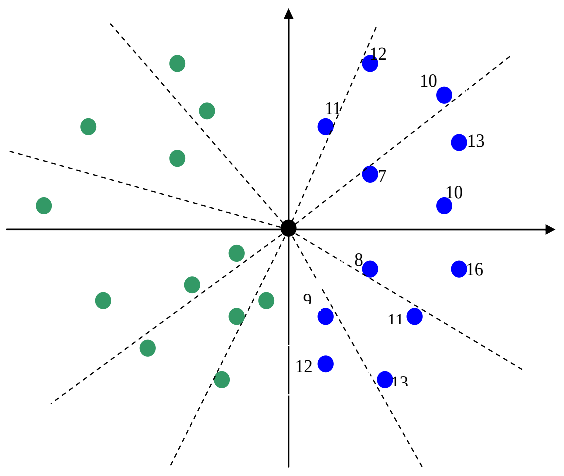


Figure 3-1-5

From the figure full of the points above, we can see there is a central depot and all the other points are just distributed around such central depot. On the plane, we may set this depot as the origin in the planar coordinates. For the other points, we may use lots of radial starting from the depot to connect those points respectively. These radials with the horizontal line crossing the depot can make a lot of angles. My way of finding the sequential distribution depends on the tangent value of those angles.

We know the angles of all the points distributed around the depot are from 0 to 2π . The way of rearranging the ID of points by angles depends on the tangent function, because this function increases monotonously when the angles are distributed at the sections of $[-\pi/2, \pi/2]$ or $[\pi/2, 3\pi/2]$. The idea will be understood more clearly from the figure following.

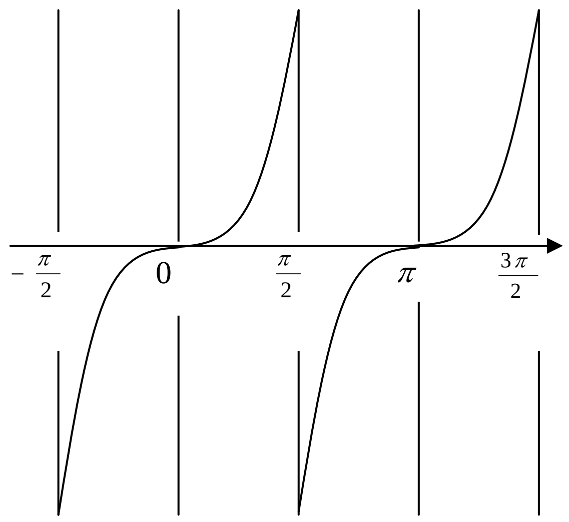


Figure 3-1-6

More detail of sorting points in the sequential way by angle is written in pseudo code shown below

```
function Rearranging Points: all points
        all points: $n \in N$
        bool: finished = false
        while not finished do
                for( $j \in N$ )
                    if ( $x_j < x_{depot}$ )
                        tangent_small = 0;
                        store such points in unite $G$ ;
                        for ( $j \in G$ )
                            if (tangent( $j$ ) $\geq$ tangent_small)
                                store point $j$ in part 1;
                                $G = G/\{j\}$;
                                tangent_small = tangent( $j$ )
                            else
```

```
15                          tangent_small = tangent( $j$ )
16                  else
17                      tangent_small = $-\infty$
18                      store such points in unite $B$
19                      for ( $j \in B$ )
20                          if (tangent(j) $\geq$ tangent_small)
21                              store point $j$ in part 2;
22                              $B = B/\{j\}$;
23                              tangent_small = tangent( $j$ )
24                          else
25                              tangent_small = tangent( $j$ )
26                              release the point $j$ in part 2;
27              while ( $N! = \phi$ )
28          All the points are sequenced by the sum $S$ of pulsing part1 and part2.
```

So far the work of rearranging the points in each day is completed. Once we have got the sequence of all the points, how can we divide those points to different groups and how to connect points in each group to form route?

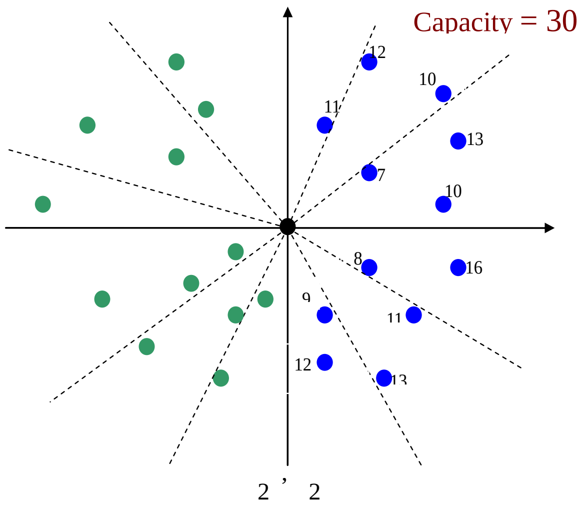


Figure 3-1-7

From the figure we can see some points with green colour on the left hand of the central depot. The x coordinate values of these points are less than the x coordinate value of the central depot. For the points with blue colour on the right hand of the central depot, their x coordinate values are bigger than the x coordinate value of the depot. Nearby the blue points, there is information of demand corresponding to those points.

Starting from the vertical line under the horizontal line, we sweep the blue points on the plane in anticlockwise direction. We can form five routes as the capacity of vehicle is 30. The total demand of points in the first route is

$$\text{demand\_route1} = 12 + 9 = 21 < 30$$

In this route, even the total demand of points is far away from 30, but we cannot insert next point. Since the total demand will exceed the vehicle capacity if we insert the next one with demand of 13.

The pseudo code of grouping points following rearrangement part can be written down as

```
29    while($S \neq \phi$)
30        for ($i = 0; i++; S-1$)
31            totalDemand = $\sum_{i=0} d_i$
32            if (totalDemand $> Q$)
33                break;
34                store points in specified group
35            else
36                go to step 30;
```

From the grouping way of Sweep_By_Angle, we need to know that if lots of routes exist with total demands that are much less than vehicle capacity, it will be difficult to find short routes. Anyway, the results from Sweep_By_Angle can tell us whether this method is feasible or not.

When I tried to connect the points in each group, I connected them in two ways:

➢ First	Connect points according to their sequence by their tangent value
➢ Second	Insert the points by method of Insert_By_Distance.

And I got two groups of solutions:

weekday	Number of customers	First way $\times 10^7$	Second way $\times 10^7$
Day1	247	**2.9398**	3.0758
Day2	214	**2.5753**	2.7219
Day3	199	**2.2364**	2.3630
Day4	229	**2.8021**	2.9904
Day5	212	**2.4978**	2.7079

Table 3-1-5

3.1.1.3 Results comparison

Until now, I did not come out with the conclusion that which constructive method is better. Even two constructive heuristics have been displayed, but both of them had shortages. The solutions of two methods are shown below

Capacity = 33, results : route length $\times 10^7$

Method	Day1	Day2	Day3	Day4	Day5
Insert_By_Distance	**2.6603**	**2.3584**	**2.1858**	**2.5817**	**2.3298**
Sweep_By_Angle	2.9398	2.5753	2.2364	2.8021	2.4978

Table 3-1-6

Capacity = 33, results : computing time
Unitage : millisecond

Method	Day1	Day2	Day3	Day4	Day5
Insert_By_Distance	1406	1219	**578**	**594**	756
Sweep_By_Angle	**969**	**203**	735	921	**500**

Figure 3-1-7

From the content of tables, it is very interesting to find that the results got by applying Sweep_By_Angle are more than the results got by applying Insertion_By_Distance.

Based on the results, it seems that it is more necessary to improve Insert_By_Distance method. But there is also room to improve Sweep_By_Angle, since some routes from it are not "full" enough. Points released from other routes, which will be destroyed, can be inserted into such un-full routes.

3.1.2 Improved heuristics

In this section, the methods of Insert_By_Distance and Sweep_By_Angle have been improved separately. For both of them, I used the conception of ejection chain referred from Braysy [11]. Thus they all have to meet two problems when they are going to be improved.

- What sort of route need to be destroyed?
- How can we insert those released points into other routes?

3.1.2.1 Based on Sweep_By_Angle

In the method of Sweep_By_Angle, the main concept of this method is to make the routes and let them distributed in some cone sections with their tops at the depot. Because from the solutions got by applying Insert_By_Distance, plenty of routes overlapped each other. And the most perfect thought of forming routes is to avoid overlapping and even ***intersection*** except the point of depot.

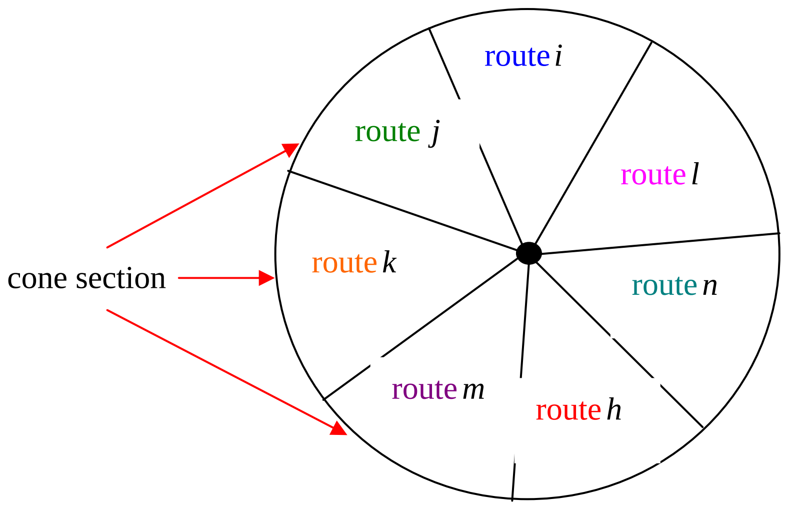


Figure 3-1-8

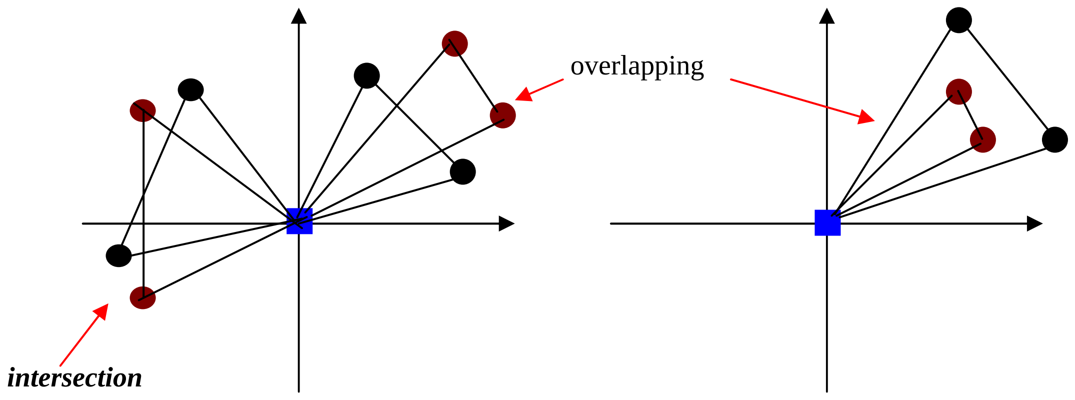


Figure 3-1-9

The routes formed by using Sweep_By_Angle had no overlapping and they were close to each other. And those routes were distributed in some cone sections on plane.

According to the theory of saving algorithm, the elimination of routes can help us shorten the route length. When I thought about the first problem mentioned above, I thought the route with lest total demand of points should be destroyed. Because there were maybe several points in such route or all the points were with little demand. After destroying such route, points with little demand could be easily inserted into other routes. Also, several points released out could be easier to be inserted to their neighbor routes.

At beginning I have got nearly 70 routes each weekday by using Sweep_By_Angle method, how to deal with several points released out and insert them into right routs among 70 routes in short time?

In order to test whether this ejection chain could bring significant results and also for optimizing routes easily, I chose to pick up just one point from route every time and try to insert it into its nearest neighbor route. The detailed action of completing this work is described below.

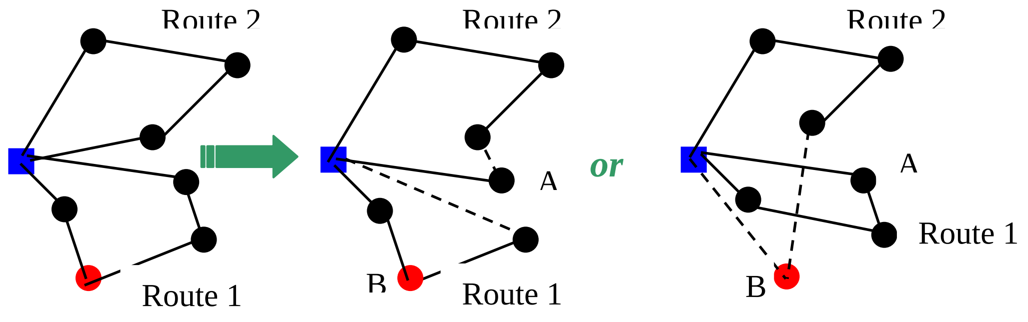


Figure 3-1-10

As we can see that the red point in the first route is very important, because if we pick up this red point from route1 and insert it to route2, new route 2 will overlap the new route1. This phenomenon can be seen from ***Figure 3-1-10***.

From the second and third figures, we will see the difference of inserting point A to Route 2 and inserting point B to Route 2. Obviously, when point B is inserted to Route 2, there is overlapping between two new routes. But inserting point A to route 2 will not produce any overlapping even the so called ***intersection*** which phenomenon I have displayed in previous page. How can we figure out the difference of point A and point B?

Actually, if we make two radial lines from the depot to point A and point B respectively, two angles will be made by those radial lines and horizontal line crossing the depot o, $\angle Aox$ and $\angle Box$.

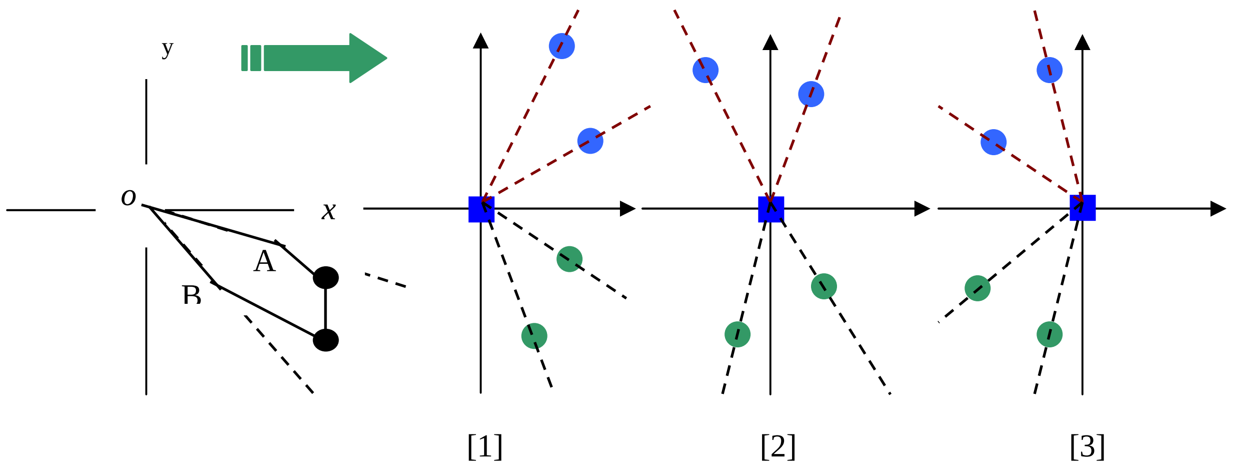


Figure 3-1-11

Looking at the figures above, there are four pictures. The first one shows the angles formed by point A and B. The others are three distributions of pair of points on the plane. Those pairs of points are distributed only on the left hand side of the vertical line as in figure [3], or on the right hand side of the vertical line as in figure [1] or crossing the vertical line as in figure [2].

Firstly, we focus on the first figure and explain how I know that we should choose A instead of B even if there are several points can be chosen. As we can see that point A and B with the depot make two angles, $\angle Aox$ and $\angle Box$ .Since the tangent function creases monotonously when the value of angles are at the sections of $[-\pi/2, \pi/2]$ and $[\pi/2, 3\pi/2]$. We know that the tangent value of $\angle Box$ is less than the tangent value of $\angle Aox$. And we know we should pick up point A and insert it to route 2 if we want to avoid overlapping. But can we find a way to deal with general cases?

Actually, when we move points and change routes in the modified way which is talking now, only one point is considered to be moved every time. In order to avoid route overlapping, such point is special. And we know each route has to start from the depot and end at the depot again, which means there are at most two points connected to the depot. We set x_{start} and x_{end} as the x coordinate values of those points. Certainly, if there is only one point in such route, we need not consider the work in such complicated way. But when there are two points, two instances will exist

Prerequisite	***Figure out the point***
➢ $(x_{start} - x_{depot}) * (x_{end} - x_{depot}) \geq 0$	Point need to be picked up is the point which has the bigger tangent value.
➢ $(x_{start} - x_{depot}) * (x_{end} - x_{depot}) < 0$	Point need to be picked up is the point which has the smaller tangent value.

So far, the rule of moving one point has been finished. In order to ensure this rule can be used to improve the Sweep_By_Angle, first we do some thing for the routes, which have been formed

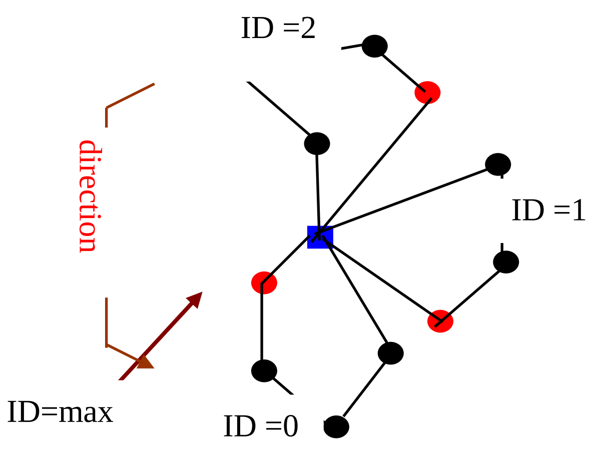


Figure 3-1-12

From the simulated route figure above, there are some routes have been shown completely and others are omit. For each route, an ID has been given. All the routes are sorted in the anticlockwise direction. The arresting things are the red points in this figure.

Those points play a very import role among the routes. They decide the sequential distribution of all the routes in anticlockwise direction.

We see all those red points are connected to the central depot directly. The way of finding out such points is almost the same as the method of figuring out the picked up points previously. But

Prerequisite	***Figure out the point***
➢ $(x_{start} - x_{depot}) * (x_{end} - x_{depot}) \geq 0$	Point need to be picked up is the point which has the smaller tangent value.
➢ $(x_{start} - x_{depot}) * (x_{end} - x_{depot}) < 0$	Point need to be picked up is the point which has the bigger tangent value.

In such way, we find out those points, save them, and sort those points to make a sequence by the way of Sweep_By_Angle.

The ID of those red points can be used for the routes in which the red points exist. After giving each route an ID, the steps of work will be displayed

➢ ***Note:***

● circulation:	start to pick up and insert point from the first route and the last route until no point is not connected;
● unchanged route:	route that has been picked up and inserted point
● rearrange points	connect points by the way of ***Insert_By_Angle*** method

➢ ***Steps of improved Sweep_By_Angle heuristic***

step1: Start from the route ID=I or ID=max with least total demand among all the routes;

step2: Pick up one point from the starting route according to the requirement of picking point mentioned before. Try to insert the point into next route ID=i+1 or ID=0;

step3: If it is feasible to insert such point to route ID=i+1 or ID=0, go to step5;

step4: If it is not feasible to insert such pint to route ID=i+1 or ID=0, go to step8;

step5: If the total demand exceeds capacity without picking up point from route ID=i+1 or ID=0, go to step7;

step6: If the total demand does not exceed capacity without picking up point from route ID=i+1 or ID=0, go to 9;

step7:	Try to pick up the point in route ID=i+1 or ID=0 with least demand and new total demand does not exceed capacity after picking up such point, go to step10;
step8:	Save such point in same point section and leave it to be considered finally after going through all the routes in the current circulation;
step9:	Rearrange all the points in new route ID=i+1 or ID=0 and denote the original route ID=i+1 or ID=0 has been changed, go to step1;
step10:	If there is no unchanged route left fort the point to move in, go to step8

The step works displayed above discuss the problem of how to pick up and insert point. From the eighth step, we know some points which cannot be inserted to any routes should be saved in some point section. Now, I will talk about way of handling those points and connecting them to make new routes.

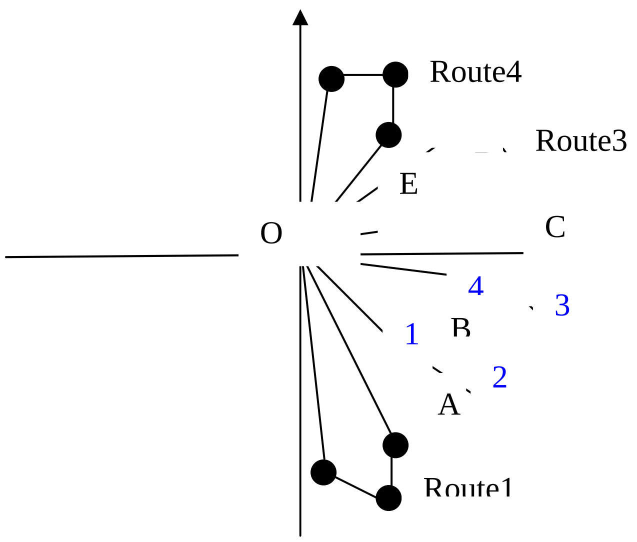


Figure 3-1-13

From the figure, we see that all the black points are connected points and the purple red points are the points, which have been saved in some point union. How to connect those purple red points? If we assume to choose the point A as the starting point, the next step of work will be described below.

First, it is clear to see that all the purple red points are sorted according to their tangent value. The sequential distribution is A, B, C, D, E. Starting from point A we try to find the next point in the sequence, which is point B. If we check all the points in the formed routes, there are two connected point1 and point2 in the region of $\angle AOB$. Since point1 and point2 with depot O cannot compose an entire route, which means depot – 1 – 2 – depot is not a route. We calculate the demand of point A and point B, and check out whether the total demand of them exceeds the vehicle capacity or not.

- ***Yes*** Connect point A to the depot and make a new route: depot – A – depot. Later start from point B and find the next new route
- ***No*** Check whether there is unconnected point left and continue finding out the next points.

We may assume that the total demand of point A and point B does not exceed the capacity. We find the next point is point C. Between point A and C, there are four connected points: poin1, point2, point3 and point4. Those four connected points form a integrate route: depot – 1 – 2 – 3 – 4 – depot. Thus we can only connect point A, point B and the depot to make route: depot – A – B – depot.

Until now, all of my idea about how I improved Sweep_By_Angle heuristic has been introduced completely. The most import and complicated works in this section are to avoid route overlapping and ***intersection*** except the depot. The phenomenon of route overlapping and ***intersection*** can be depicted clearly and visibly in the following figures

3.1.2.1.1 Result from the modified Sweep_By_Angle

In this section, results got by applying the modified Sweep_By_Angle are listed. Those results compare with the solution coming from the original method of Sweep_By_Angle. When I tried to get results from the modified Sweep_By_Angle, I made some iteration to run the modified algorithm. The iteration helped points move from one route to another route. I thought the solution of modified Sweep_By_Angle in each weekday could form a curve. This curve depicts the distribution of final results in one weekday after doing iterations.

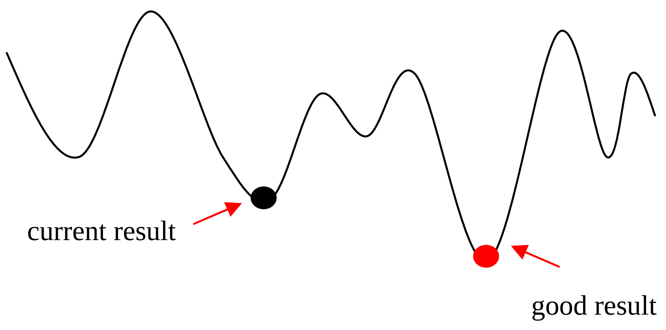


Figure 3-1-14

We see good result can be easily found out from the curve, since I could figure out when I get good results by analyzing the curve. When I did the iteration, I set the number of iteration as 100. Actually the number of 100 has no warrant to the algorithm iteration process. But by doing 100 iterations, the results reminded me the way of modified Sweep_By_Angle had some problem.

- original results: Results got from the original Sweep_By_Angle without improving
- current results: First five results after doing 100 iterations

Based on Insert_By_Distance

workday	Number of points	original result $\times 10^7$	current results $\times 10^7$
Day1	247	**2.9398**	**2.9398**
			3.1554
			3.3425
			3.4030
			3.3626.
Day2	214	**2.5753**	**2.5753**
			2.8799
			2.9000
			3.0442
			3.0454
Day3	199	**2.2364**	**2.2364**
			2.4453
			2.4770
			2.5943
			2.6556
Day4	229	**2.8021**	**2.8021**
			2.9612
			3.1113
			3.2041
			3.1665
Day5	212	**2.4978**	**2.4978**
			2.7157
			2.8265
			2.8508
			2.8753

Table 3-1-8

From the results displayed in the table, I was very surprised that the modified algorithm actually has not helped me to shorten the route length. On the contrary, it made the results be even worse than the results obtained originally. At that moment, I still did not want to give up my mind, because I wanted to use the idea of modified Sweep_By_Angle to modify the method of Insert_By_Distance. And I expected to get better solution after doing so.

The way of modifying Insert_By_Distance here is more or less the same as the modified behavior described in the previous section, but there is a change in the action of **rearranging points**. Once the original routes have been changed, the points inside them need to be connected by applying Insert_By_Distance. Since ejection chain is to change the position of points in routes, points in the original routes got from Insert_By_Distance have to be connected in the same way used the original one.

When I used the way to Insert_By_Distance, I still use 100 iterations as the steps of moving points among all the routes. Before starting to iterate the modified behavior, I sorted all the routes and gave the distributed sequence to the routes formed in Insert_By_Distance. The results are

- original results: Results got from the original Insert_By_Distance without improving
- current results: First five results after doing 100 iterations

Based on Insert_By_Distance

workday	Number of points	original result $\times 10^7$	current results $\times 10^7$
Day1	247	2.6603	**2.6253**
			3.1226
			3.2955
			3.4299
			3.3973
Day2	214	2.3584	**2.3347**
			2.8790
			2.9393
			3.1399
			3.2622
Day3	199	2.1858.	**2.1612**
			2.5844
			2.6466
			2.7086
			2.6362
Day4	229	2.5817	**2.5733**
			2.9358
			3.1548
			3.2058
			3.2220
Day5	212	2.3298	**2.3169**
			2.7593
			2.8887
			2.9251
			3.0362

Table 3-1-9

Unfortunately, the ejection chain cannot be used to improve Insert_By_Distance further, even though better results could be got at the beginning. This can be referred from the results above. We see the first results in bold-face are little less than the original results, but the iteration goes up, no better results exist. Even though we should not say the method of ejection chain has no effect when it used to eliminate routes.

The failure of using ejection chain here belongs to the modified method per se. The ejection chain could have improved the algorithm and helped to get better results actually, since some bad phenomenon that could use the ejection chain to resolve:

- Some points in the same route but they are far from each other
- Some points in different routes but near by each other

The method of ejection chain could help us to rearrange the distribution of points in routes. The failure of modified behavior is the way of using ejection chain above did not associate with other important constraint. I only concerned with moving points without thinking whether such movement was good or it would make the route length to be longer.

3.1.2.1.2 *Method analysis about Sweep_By_Angle*

So far, some failure experience of modifying method has been collected. But the failure of modifying Sweep_By_Angle should not be ignored, because the failure from them helped me improve Insert_By_Distance in more feasible way later.

In my point of view, the first reason of failure is in the part of modification procedure. In this part, I just wanted to decrease the number of routes and removed the points among routes expecting to get better results. But I did not set shortening route length as the prerequisite when I moved points. If I only moved points when short routes could be produced, the modification procedure would help me to get less value of route length at the end.
In my mind, I thought the best perfect instance of forming routes is to avoid route overlapping and ***intersection***. The imagined perfect routes can be felt from the figure following

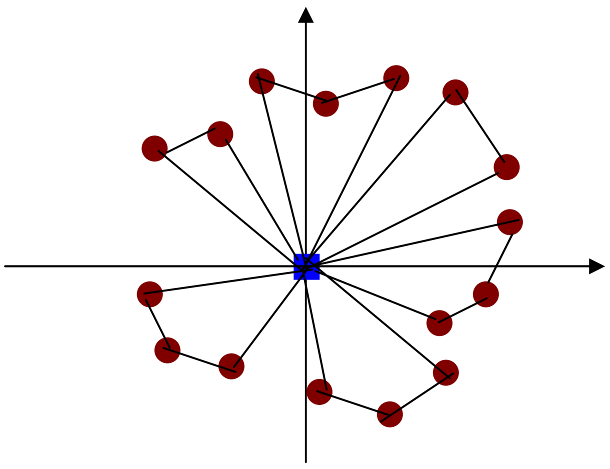


Figure 3-1-15

Actually, the expectations of avoiding route overlapping and ***intersection*** are right. But when there are plenty of clustered customers distributed on the plane, such expectation is luxurious. If we assume that lots of clustered points have to be assigned to different vehicles with capacity constraint, how we can ensure that all the point will be arranged averagely without route overlapping. The fact is that it is difficult to avoid route overlapping and ***intersection***, if points are clustered. So this is the second reason why modification is failed.

Third, picking up and moving only one point every time is not a good way to modify method. Firstly, it is a bad choice from the point of time consumption. Secondly, the choice of picking up point is limited. Only choosing the point connected to the depot directly will badly block other points to be picked up.

3.1.2.2 Based on Insert_By_Distance

In the part of introducing constructive heuristics, I have set two directions to form routes: Sweep_By_Angle and Insert_By_Distance. So far, I have explained the way of modifying the Sweep_By_Angle and I also used the same way to modify Inser_By_Distance, but as we see that I did not get effective solution. Thus I turned to Insert_By_Distance, tried to modify it and expected to get feasible solution.

The process of my thought can be described as the flow chart

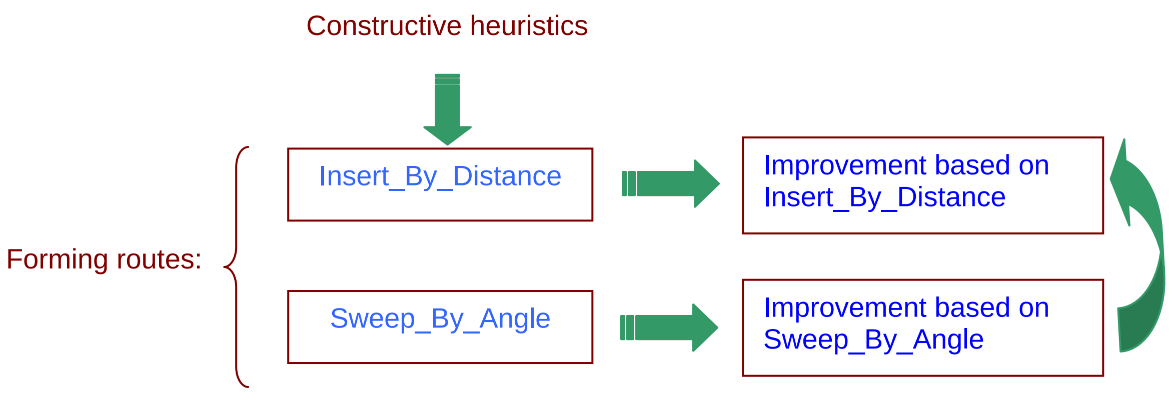

Constructive heuristics
Forming routes:
Insert_By_Distance
Sweep_By_Angle
Improvement based on Insert_By_Distance
Improvement based on Sweep_By_Angle

Figure 3-1-16

Fortunately, I have got excellent solution. At least the results were much better than the old ones. Since I had the experience of working with modifying Sweep_By_Angle, I mainly emphasized two important rules when I focused on modifying Insert_By_Distance. This rule was not only experiential words, but it also brought some significant breakthrough to the results.

The first rule of modifying Insert_By_Distance only allowed inserting point but not replacing point. From the experience of previous work, I knew that if the replacing action was successful, it might help us to move points and get better position for them. But if it was failed, the points replaced outside without being accepted by other routes would form some new routes, which disobey the idea of *saving heuristic*.

Actually, my new idea of modifying the Insert_By_Distance can be described generally in two steps. The first step is to pick up every route i from the formed route section and optimize such route with its next neighborhood route $i+1$. The second step of work will start after finishing optimizing all the routes. In the second step, the work is to set some constraint and make iteration to continue optimizing results. The second step is also the second rule I will talk about later

Now, Let us talk about the first step:

Function: modified Insert_By_Distance

```
pSelectedInR1 = -1
largestPDecrease = -1
orderly pickup each point p from route i
{
    lengthDecrease = the length decrease from route i if take off point p
    If (point p can be inserted into route i+1)
    {
        lengthIncrease = the length increase from route i+1 if insert point p
        if ( lengthIncrease < lengthDecrease)
        {
            Pdecrease = lengthDecrease - lengthIncrease
            if (largestPDecrease == -1 )
            {
                largestPDecrease = Pdecrease
                pSelectedInR1 = currentPoint p
            }
            else if ( Pdecrease > largestPDecrease )
            {
                largestPDecrease = Pdecrease
                pSelectedInR1 = currentPoint p
            }
        }
    }
 }
 if ( pSelectedInR1 != -1 )
 {
     move point pSelectedInR1 from route i to route i+1
 }
```

After passing the first step of work, some iteration made to optimize the solution in the second phase will be presented. In this iteration part, we will start to optimize the route got from the constructive method of Insert_By_Distance and then use the way of modified Insert_By_Distance in the anticlockwise direction. But we won’t only follow the anticlockwise direction; the reason is if some routes are bad at the beginning, the new formed routes will be worse and worse. Such judgment will be explained more clearly by using figures

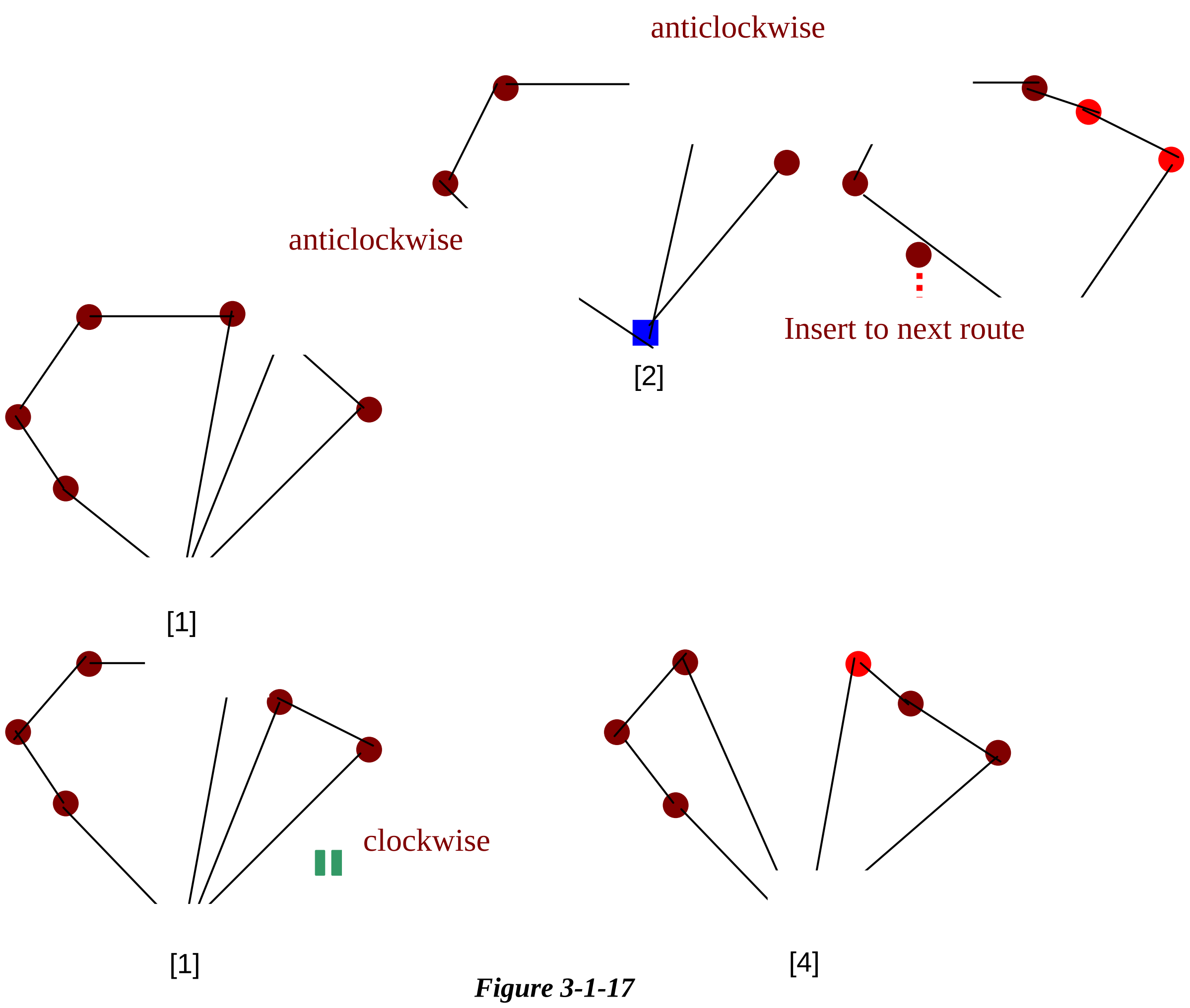
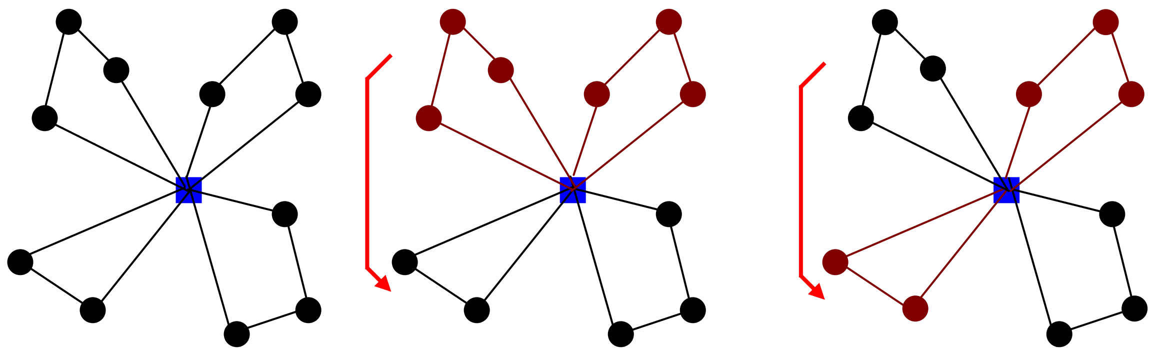


Figure 3-1-17

From the figures above, we can see that two groups of figures are displayed. The actions in both figures are to pick up point and insert it to another route in the anticlockwise and clockwise directions respectively. In the anticlockwise direction, figure [2] includes the optimized routes coming from figure [1]. And figure [3] includes the new routes got from figure [2]. In the clockwise direction, figure [4] includes the routes optimized from the figure [1].

The way of optimizing routes has been found. But if we continue only optimizing two nearby routes, there must be no routes can be optimized each other after several iterations. When we meet this problem, we should select the routes to optimize without only considering two nearest ones. A variable stepLength will be set in the program to guide the algorithm choose routes and optimize them. For instance, if we set the stepLength is 1, which means we only optimize the current route with its next nearest neighbour route. There are some simple figures drawn to show the work of stepLength.



original routes *stepLength = 1* *stepLength = 2*

Figure 3-1-18

The original route with black points has been displayed on the left. In order to show the use of stepLength, points in the chosen routes going to be optimized are in purple red in second and the third figures. When stepLength equals to 1, the ID of the next route just equals to the ID of current route plus 1. In the same way, when stepLength is 2, the ID of next route is the ID of the current route plus 2. Other value of stepLength will follow this way.

The introduction of stepLength makes us know the meaning of doing so. But the method that how we use the way of changing stepLength to optimize routes still have not been explained completely. The way of iterating algorithm is to optimize routes by applying the method in anticlockwise direction first and then continue optimizing routes in clockwise direction.

So when we use the way of stepLength, there are three aspects we need to concern.

- ➢ First: When we set the stepLength, every iteration is the action to optimize routes both in anticlockwise and clockwise directions. We can only stop and change the stepLength when there is no route can be optimized;
- ➢ Second: The optimization way only includes inserting but not replacing ;
- ➢ Third: The value of stepLength cannot be increased without limitation after several times of iteration. We need to set a value for maxStepLength aiming at different instances. The purpose of setting max value to stepLength is that all the routes are distributed just at one part of the planar, as

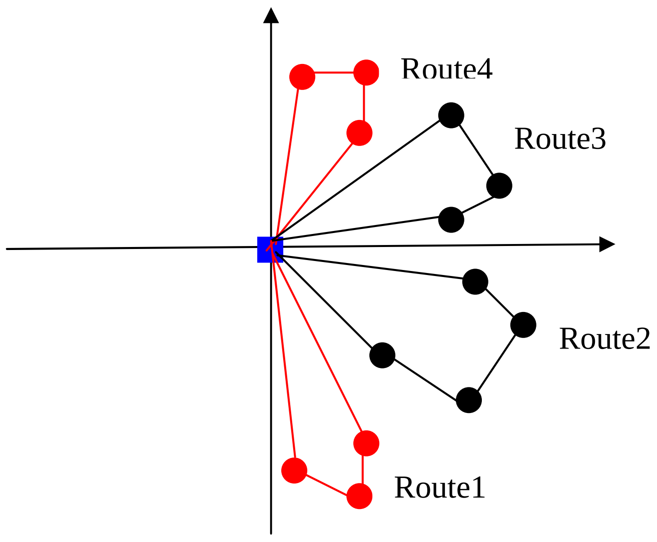


Figure 3-1-19

In the figure, as we can see that route optimization starting from route1 and trying to minimize it with route 4 is not a good choice, since those two routes are far from each other. The maximum stepLength is not the same value as the number of routes, the maximum stepLength need to be tested out. And the maximum stepLength can help us to avoid the bad choice mentioned here. This is not only theoretic advice; it actually came from the experience that I have done. And it is also tenable in theory.

By going through the test part, I found that the value of stepLength should not be increased up to the number of routes, because routes will stop being optimized after the stepLength reach to some value. Even though the stepLength can be increased bigger and bigger, but those bigger values of stepLength will not help routes to be optimized any further.

We should know that nothing can be fixed and only be considered in one way. Every thing will impress different meanings under different instances. Thus I have used the stepLength in two different ways. The use of stepLength plays a very important role in this thesis. The different ways of using it have helped me get two different solutions.

As the case stands, there are two directions we can choose. The first one is we increase the stepLength continuously in the process of optimizing routes until reach the maxStepLength. The second one is to change the stepLength in discontinuity way. The process is

- Step1: Under the instance: *stepLength > 1*
- Step2: Focusing on the current stepLength, optimize routes until there is no change in routes
- Step3: Reset the *stepLength = 1* and try to optimize any two nearest routes since some routes must have been changed and reformed in the previous step, which bring bigger probability to optimize some nearest new routes.

- Step4 Increase the stepLength until there is no change in routes. Go to step1.

3.1.2.2.1 Result from the modified Insert_By_Distance

So far, the whole work of forming routes is nearly finished. During the procedure, first I built two constructive heuristics as the start of searching method. Then I modified the Sweep_By_Angle method and got some useful and reasonable conclusions that how to modify methods and optimize the solutions in more effective way.

After having done some work to modify the method of Sweep_By_Angle, I have improved the way of improving methods. I realize that there are two rules need to obey if we expected to get better results in valid way comparing with the methods used in Sweep_By_Angle.

- Only allow to inserting point but not replace point;
- Using the circulatory approach effectively to optimize results;

In the table, there are two groups of results, they are list below

➢ First group	results by increasing the stepLength continually
➢ Second group	results by increasing stepLength in discontinuity way

Results of first part by using modified Insert_By_Distance:

Capacity = 33;

weekday	Number of points	Original results $\times 10^7$	First group $\times 10^7$	Second group $\times 10^7$
Day1	247	2.6603	2.6131	**2.6040**
Day2	214	2.3584	2.3233	**2.3023**
Day3	199	2.1858	2.0940	**2.0881**
Day4	229	2.5817	**2.5339**	2.5359
Day5	212	2.3298	2.2401	**2.2232**

Table 3-1-10

Results of second part by using modified Insert_By_Distance:

Capacity = 33

weekday	Number of points	Original results	First group	Second group
Day1	247	9744405	9271989	**9085027**
Day2	214	8708209	8570935	**8379432**
Day3	199	7983595	7965560	**7914716**
Day4	229	8611727	8519576	**8483032**
Day5	212	8081702	7812657	**7789271**

Table 3-1-11

Along with the route length in each weekday, there are computing time consumed by running the algorithms in different way of changing stepLength are listed below

- First results by increasing the stepLength continually
- second results by increasing stepLength in discontinuity way

time unitage : millisecond

Group Name	*Day1*	*Day2*	*Day3*	*Day4*	*Day5*
First	**1641**	**2422**	**4682**	**2969**	4734
Second	3172	7453	4953	4813	**4500**

Table 3-1-12

In tables above, the results displayed are coming from the solutions in different parts by using modified Insert_By_Distance. The fist and the second tables show the results in the pickup part and delivery part got by changing the value of stepLength continually or just changing it in discontinuity way respectively. More details about changing stepLength continually or in discontinuity way will be explained later in method analysis. And the third table shows the consuming time spent in every weekday by changing the stepLength in different ways.

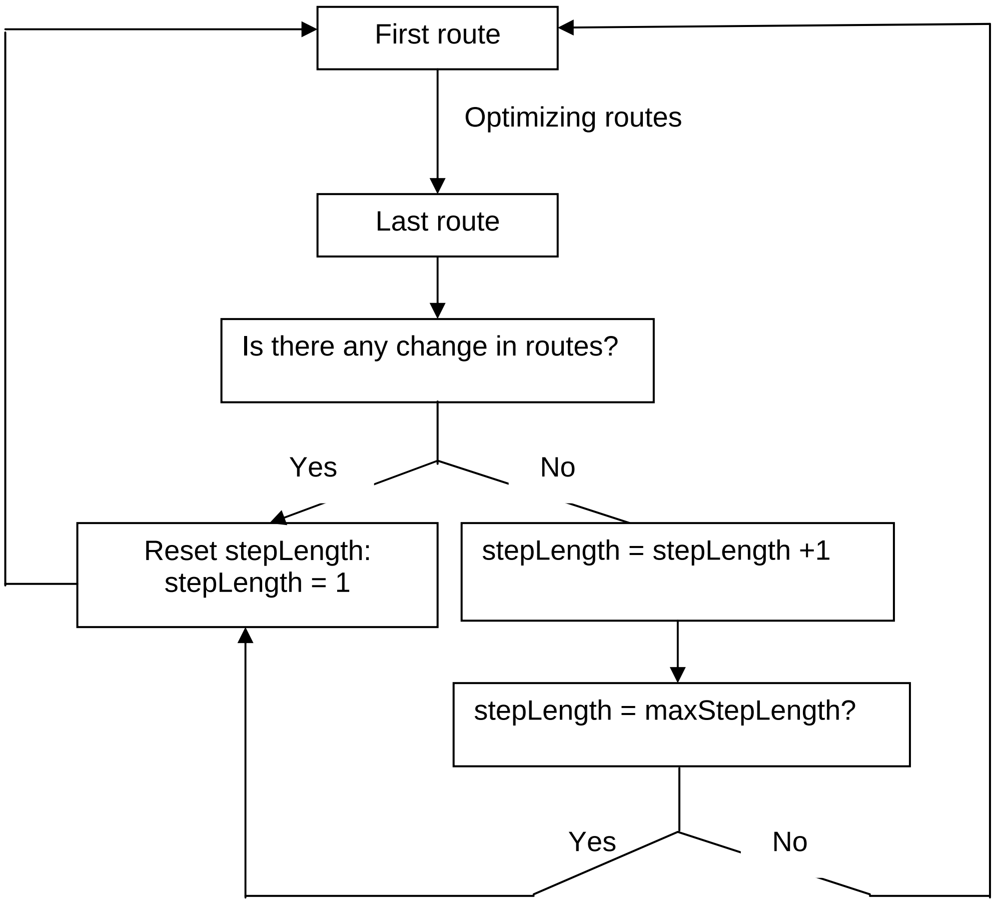
The results in bold-face are the results with smaller value. And we found that the way of changing stepLength in discontinuity way help us to get the best results so far. But the computing time is much.

***3.1.2.2.2** Method analysis about Insert_By_Distance*

3.1.2.2.2 .1 review the method and analyze it

From all the results, I discovered that the rule of only allowing to insert point brings a significant improvement to solution. And the way of changing stepLength also makes a great effect. Although the results got both in changing the stepLength continually and in discontinuity way are not much different, those two manners of changing stepLength exist big distinctness in the algorithm design.

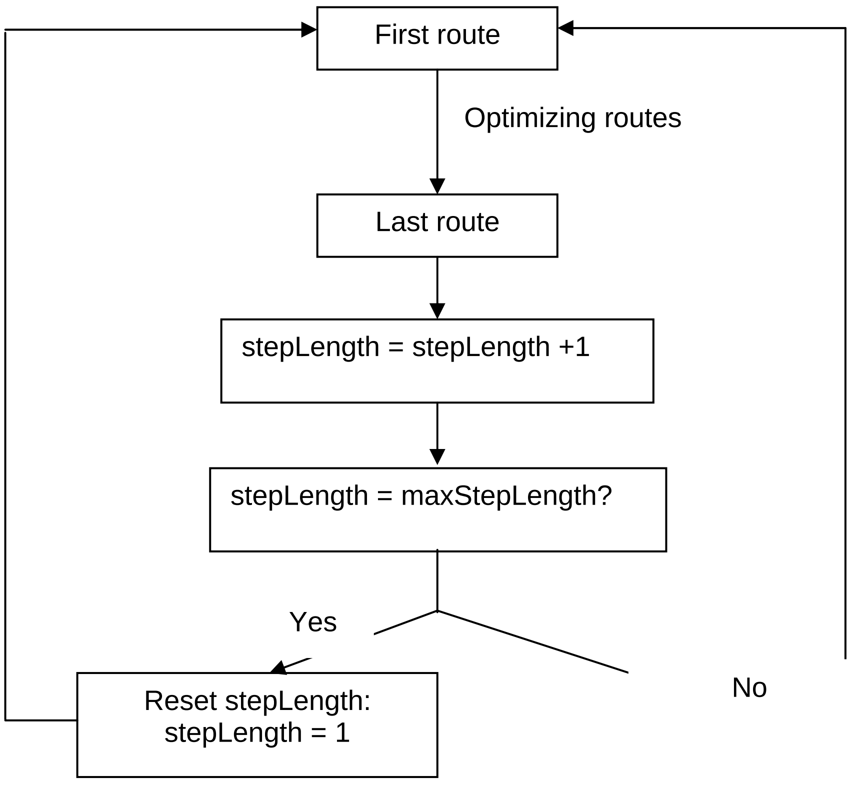
If we use flow chart to describe those two manners, they should be designed as



First route
Optimizing routes
Last route
Is there any change in routes?
Yes
No
Reset stepLength:
stepLength = 1
stepLength = stepLength +1
stepLength = maxStepLength?
Yes
No

Flow chart of changing the stepLength in discontinuity way

Figure 3-1-20



Flow chart of changing the stepLength continually

Figure 3-1-21

Looking at those two flow charts above, there are some steps of work we need to review.

➢ Optimizing routes :	The procedure here is try to optimize one pair of routes every time
➢ stepLength = stepLength +1 :	This is the way of choosing the route to optimize
➢ maxStepLength :	The value need to test according the given data

As we see that the difference existing in two changing ways of stepLength is time, which can refer the value from the table in previous section. And we could also see the complexity. If we set the average complexity is O under each iteration with some fixed stepLength and the maximum stepLength is m , the total complexity holden by method of changing stepLength continually and the method of changing stepLength in discontinuity way are mO and $m(m+1)O$ respectively.

3.1.2.2.2.2 some new idea for modifying method

The VRP is so popular. People have been being spent their time doing its relevant researches since they start to know the importance of such problem. Because of researchers' contributions, there are a lot of approaches discovered. In this thesis, some algorithms are also carried out and listed by referring the avant-couriers' good idea.

From the work I had done, I realize that it is necessary to do some brave attempts if we have some ideas and think them to be reasonable. The efficiency and perfection of one method relies on the accurate exhibition of results at the end. This is also the attitude recognized by the great philosophy conception: the enchantment of maths is that the results got from it are perfect because of their accuracy.

Now that we should not give up any reasonable ideas to solve problems, every such instance is worthy to trying. In this thesis, plenty of thought came to my mind, some of them have been implemented as work shown previously. And for the others, there is no more time to implement them. But I trustfully think that those thoughts have not been worked with may bring uncommon results. Thus I would like to write some new idea down; perhaps it will give people some information and help them to make more general conclusions if it can be considered to work with.

The new method I would like to talk is based on the results got from the Insert_by_Distance. If we have the original source of results as

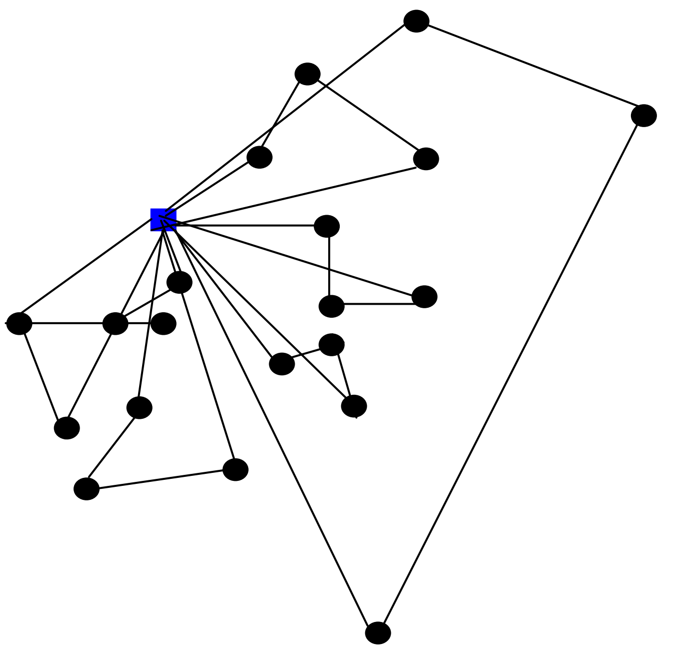


Figure 3-1-22

The new method is the combination of inserting and replacing. In this method, we pay our main attention to the demand of point, because the distance of the total routes dose not only rely on the length of every route, but also is constrained under the vehicle capacity. And the results got from Insert_by_Distance have a character that the procedure of forming route is mainly restricted to the constraint of vehicle capacity.

Owing the figure above, we know that ID has set for each route when routes were formed. But those ID of routes are not sequenced in anticlockwise direction. So at the beginning of modifying Insert_By_Distance by using the new idea, we have to resort all the routes and make them to distribute in the anticlockwise direction. The way of handling that is the same as the method described in the previous section. After sorting the routes, if we use a simple picture to show the information of routes formed from Insert_By_Distance, the figure should be described as

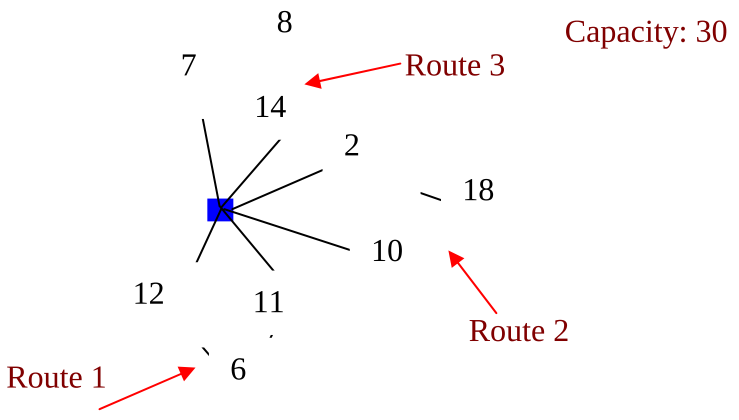


Figure 3-1-23

The figure almost includes basic information we need. As we can see that the capacity of vehicle fixed for the routes in this example is 30 and three routes are supposed to be formed by applying the method of Insert_By_Distance. For instance, we focus on the route 1 and route 2, if we want to pickup point from route 1 and insert such point into route 2 or use this point to replace the point in the original route 2. Then we get

	route1	route2
demand	18	12, 11, 6
demand	10	6

18 out $\longrightarrow$ 12 insert
18 out $\longrightarrow$ 11 insert
18 out $\longrightarrow$ 6 insert
10 out $\longrightarrow$ 6 insert

Figure 3-1-24

In this figure, since most of the routes are full, we need to pickup one point/points from one route and use such point/points to replace point/points in next route. In order to be sure that the behavior can be carried out completely, ***the demand of point which will be replaced out should be more than the demand of point used to replace such point***.

According to obey the rule in boldface, we see there will be four possible new route2 if we let the points with 18 and 10 out from route2 respectively. From those new route2, we all compare them with the original route2, and then there are two instances will happen

➢ first choice	if some of the new route is shorter than route2, then we choose such new shorter route;
➢ second choice	if none of them are shorter than route2, then we give up doing so but consider route2 and route3 in the same way

We are familiar with the basic method now, but if we suppose the first choice happened, and what we should do next is a problem. There are some steps can help us to solve the problem if the first choice has been achieved.

Step1: one point from route2 will be picked up

Step2: if such point can be inserted, check whether new route 3 is shorter comparing with the original route3

- *shorter or no change?* insert point
- *bigger?* do not insert, consider route3 and route 4

Step3: if such point cannot be inserted, just leave it

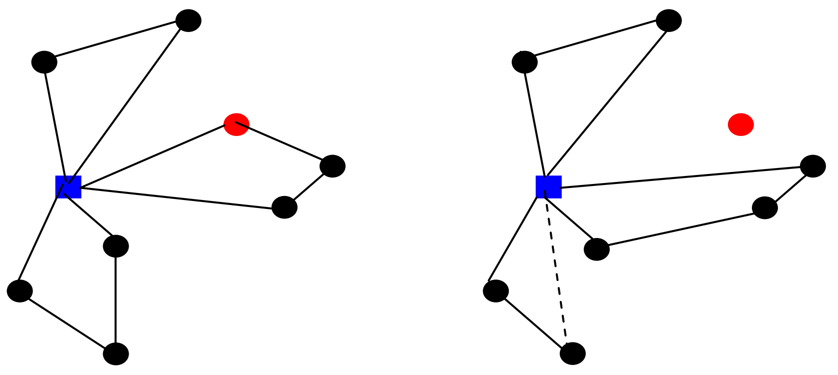


Figure 3-1-25

Step4	After going through all routes in anticlockwise direction. There are some points have been pushed aside the connected points	
	Try to insert such sort of unconnected points to their nearest route neighbors	
Step5	Can be inserted?	Yes, calculate the distance and compare the distance of route neighbor and the distance of the route formed only by connecting the point to the depot **Choose the route with shorter distance**
	Cannot be inserted?	Yes, just connect the point to the depot, make route with such point and the depot
Step6	Until all the points have been connected, then make circulation and find the shortest one	

Until now, the description of new idea of modifying method is finished. In this new idea, the conception of optimizing routes does not refer the experience summarized from the modified Insert_By_Distance such as replacement. The new idea allows the existence of replacement.

This new idea of modifying method may not lead the route length to be shorter enough. But as the optimizing way is very tight, it is still worthy to implementing and testing whether there is any improvement to the results.

3.2 Connecting routes

In this part, step of work connecting the relative formed routes will be described. First, we need to find out the relationship of routes both in pickup part and the delivery part. Second, all the relative routes will be tested relying on the working time. Since there is a time constraint given in the problem that every vehicle has to work from 06:00 to 22:00. If the time is invalid, we have to rearrange the troublesome routes.

3.2.1 Searching relative routes in different parts

Now, all the points have been connected to compose routes. Because the parts of pickup and delivery are split, in order to ensure every order can be picked up and transported to its termination successfully. The work we need to do now is to find out the relative routes in both parts, which means we have to connect all the relative sources and terminations together.

First we discuss the way of making connections for relative routes between pickup and delivery parts.

For the sake of describing the method, a figure including routes in both parts is displayed. It is clear to see that this simple figure composed with one central depot, some points and several routes formed with links between two nodes. The routes in the figure are generally distributed on both sides of the central depot.

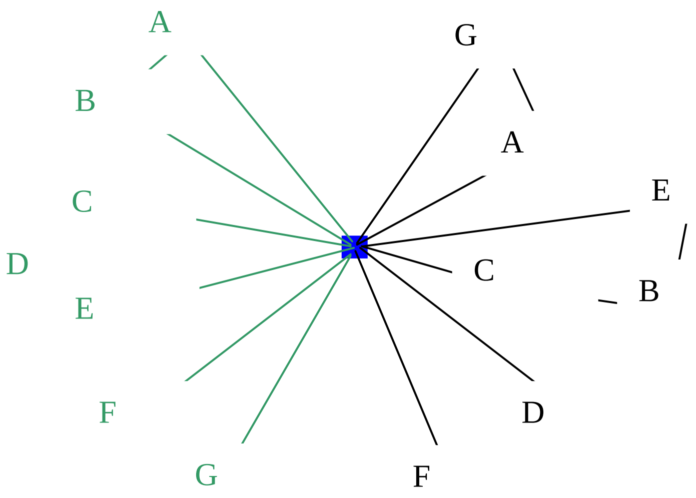


Figure 3-2-1

We may say routes on the left hand side of the depot come from pickup part. Routes on the right hand side of the depot come from delivery part. In every route, there are some points inside it. And it is obvious to see that the points on both sides are given the same ID for the source points and their corresponding terminal points, which can help us to find out the relative routes by checking the ID of points from different parts.

First, for both side we clarify the nodes in each routes:

Left side	Right side
$R1 = \{B, A\}$	$RI = \{A, G\}$
$R2 = \{E, D, C\}$	$RII = \{C, B, E\}$
$R3 = \{F, G\}$	$RIII = \{D, F\}$

Table 3-2-1

Second, we test whether there is union between any two routes from different parts:

$R1$	$R2$	$R3$
$R1 \bigcap RI = \{A\};$	$R2 \bigcap RI = \phi;$	$R3 \bigcap RI = \{G\};$
$R1 \bigcap RII = \{B\};$	$R2 \bigcap RII = \{C, E\};$	$R3 \bigcap RII = \phi;$
$R1 \bigcap RIII = \phi;$	$R2 \bigcap RIII = \{D\};$	$R3 \bigcap RIII = \{F\};$

Table 3-2-2

From the table we see that when we focus on different routes in pickup part and check their relative routes in delivery part, there are some different results come out. For $R1$ in pickup part, we find that routes RI and RII have union with $R1$, which can be explained that the goods from source points in $R1$ have to be delivered to terminal points in RI and RII separately. For route $R2$ and $R3$, they can be understood in the same way as route $R1$.

By using the way above, we can find out all the relative routes coming from different parts.

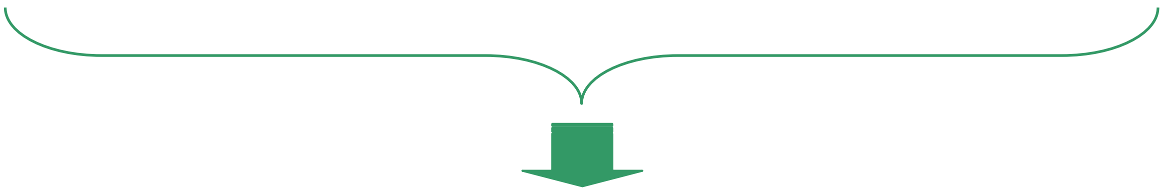
3.2.2 Rearrange the overload routes

As we have mentioned that every vehicle has been bounded with a working time period. Once there is a route union, we have to check whether the total time spent in any two relative routes exceeds the time constraint or not. In this thesis, the problem has been taken starting at 06:00 in the morning and finishing at 22:00 in the evening, so time period is bounded as 16 hours every day. The work we need to do next is to test time spending in routes.

In order to be convenient to understand, we set I as the collection of routes from the pickup part, every route has an ID named $i \in I$ and the collection of routes in second part set as J, routes with ID inside called $j \in J$. Since all the routes are formed by connecting points, point $k \in K_i$ means nodes in pickup part, and point $k \in K_j$ in delivery part. K_i and K_j are the gathers of points in different parts. And the demand of points in different part, we set as $d_i \leq 33$ and $d_j \leq 33$.

In this problem procedure, every order need to be uploaded at the source node, offloaded and uploaded again at the central depot, and then offloaded at the terminal node. For every process of uploading or offloading, it will take 10 minutes and the one minute for every demand of goods. The average speed of each vehicle is 60 km/hour. Once we find the relative routes, the total time spent in them is

$$TotalTime = TotalTransportingTime + 2 * (uploadingTime + offloadingTime);$$
$$TransportingTime = (Length\_i + Length\_j) / 60000;$$
$$uploadingTime = 10 + d_i + 10 + d_i;$$
$$offloadingTime = 10 + d_j + 10 + d_j;$$



$$TotalTime == (Length\_i + Length\_j)/60000 + 40 + 2*(d_i + d_j)$$

Knowing the way of calculating total time, the steps underlying which are adopted will help us to check whether any two routes are feasible to be connected

```
1.    start from $i = 0$, $i$ is the ID of route coming from the pickup part
2.    for ($i = 0, i++, i < \sum i$)
3.        go through all the routes in delivery part
4.        if $k \in K_i \cap k \in K_j \neq \phi$ means routes coming from different parts have the
5.           same ID
6.           $TotalTime == (Length\_i + Length\_j)/60000 + 40 + 2*(d_i + d_j)$
7.           if $TotalTime > 16$
8.              record the route $i$ and its relative route $j$, save them in special collection
9.           else
10.                routes from pickup and delivery parts can be connected and such connection
11.                is feasible
```

In the step8, we save the relative routes whose spending time exceeded bounded time period. For the sake of understanding simply, we set the process of uploading, transporting and offloading the same amount goods in the pickup part as the first part. Similarly, we set the process of uploading, transporting and offloading the same amount of goods as the second part. Then the time spent in routes which need to be rearranged can be visually represented as

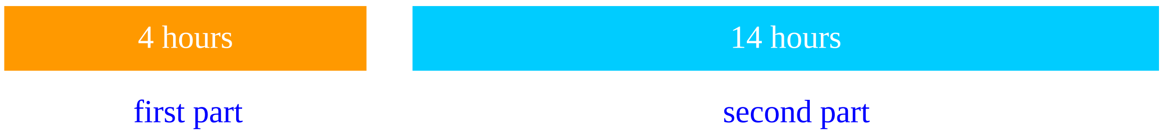


Figure 3-2-2

From the figure, we can see that the time spent in different parts is 4 and 14 hours respectively, which exceeds the total time bounded as 16 hours. The only problem make this phenomenon happen is some route take too much time to finish work such as the second part above. There is an idea jump into my mind that if we can use one more vehicles to finish the same work which has been done by only one vehicle but using twice time value, why we should not choose more vehicles.

Once we plan to use more vehicles to handle the same work burdened by several vehicles before, there are three problems need to be considered firstly:

- How can we split the longer route?
- How many new routes should be formed?
- Knowing the number of the new routes, how can we distribute the points which have been released from the longer route?

From those problem, we know that if we want to split route, firstly we should know how many number of new routes will be engendered. The way of dealing such problem is

$routeLength$ = length of the longer route
n = number of new routes

$$n = \left\lfloor \frac{\frac{routeLength}{speed}}{bestTime} \right\rfloor + 1$$

$$bestTime = \frac{16 - \frac{40 + 2 * capacity}{60}}{2}$$

The bestTime here plays a very important role, first it helps us to get the number of new routes, second it supplies us a measure to test whether we should split formed route or not. Then we know if $routeLength > bestTime$, it is necessary to rearrange the routes.

As we have talked just now, there is a job given to be handled only by one vehicle in T time period, when we wan to shorten the working time, we can rearrange this job to two vehicles to finish. If every vehicle uses the same time period, then each of them will use $T/2$, which means we can complete the whole work in a half of the original time. So

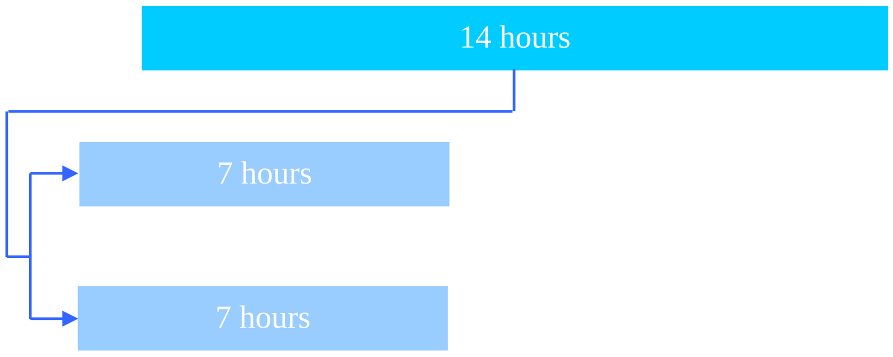


Figure 3-2-3

The imagination of splitting routes is perfect, but we cannot ensure this perfection exist. Because one route cannot be slipt and completed by several vehicles in the some *bestTime* period. Every vehicle starts from the central depot and it has to come back to the central depot again after visiting its customers, which means it will be impossible for us to split a routes into several average new routes if we want the sum of total split time equals to the time spent in the original route. But there is an eclectic instance that we could try our best to make sure *newRouteLength* nearly reach the value of $routeLength/n$. Then we come to meet the problem that:

- ***Knowing the number of the new routes, how can we distribute the points which have been released from the longer route?***

In the way of rearranging released points and connecting them to form routes, we will adopt some new idea.

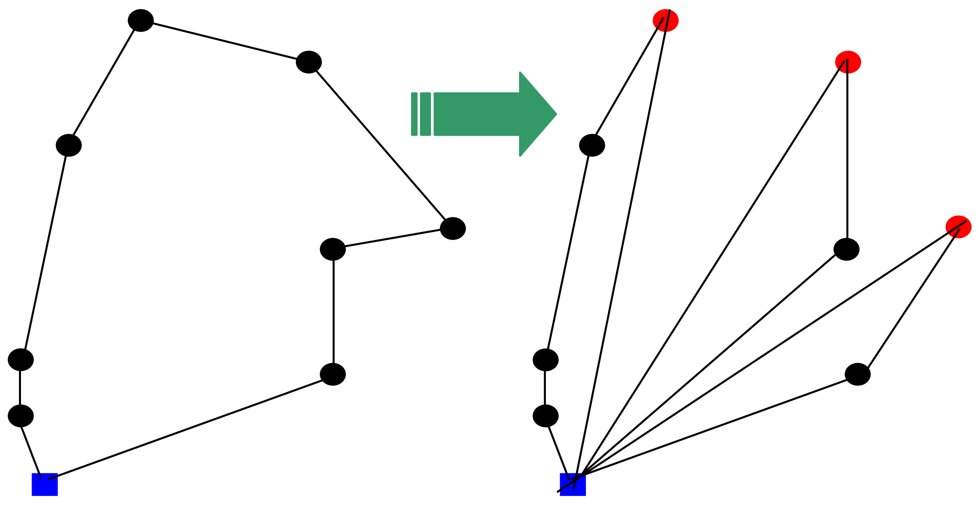


Figure 3-2-4

The figures above describe a process of rearranging. The figure on the left is the original route and the one on the right shows the new routes which have been split from the original one. The procedure of splitting such long original route can be described as the steps following

➢	Step 1	release all the points in the rearranging long route and save these points in a point union W
➢	Step 2	start from the central depot o, search and connect the furthest point i and go back to point o. Pick up point i from W
➢	Step 3	Check the route composed by the point/points and the central depot o
➢	Step 4	if $newRouteLength < routeLength/n$, search and connect the nearest point j to previous point i and go back to the central depot to make route. Pick up point j from W and go to step 4
➢	Step 5	else if $newRouteLength$ nearly equals to $routeLength/n$, go back to

the depot. Pick up point j from W and check whether $W = \phi$. If $W \neq \phi$, go back to step 2

Let us go back to the figure, and go through the procedure again. As we see that there are three red points in right figure. They are the first point of every route beside the central depot. We call those red points seed customers. And the seed customers here are the points which are far from the central depot. Once we connect such seed customer with the central depot, the next work is to find the nearest point to this seed customer. Every time when we search point, we need to suppose those searched points can be composed to form new route and check whether the time spent in the route is nearly equals to the value of $routeLength/n$. If it is, we stop searching next point. But if it is not, we have to continue until there is no satisfied point.

It seems that there is no problem to arrange those points which have been released out from the long original route. But since every time we compose route only by picking up the points satisfy the time limitation, there must be some point which is not valid to be picked up to form route when we finish composing routes. How do we handle those separate points?

There is a figure underlying, it is very clear to see from the figure that the left points can be inserted into their neighborhood routes. Because if we only connect them with the central depot separately, this way disobeys to the saving method as we have proved in the previous section. But inserting points will help us to avoid increasing total route length. And inserting points to their nearest neighborhood route will not add the routeLength too long comparing with the value of $routeLength/n$ firstly, and secondly the consideration of the phenomenon of exceeding vehicle capacity can be omitted.

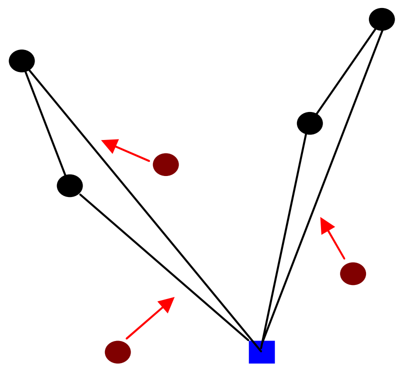


Figure 3-2-5

In conclusion, the procedure of rearranging the long route from the relative pair of routes can be described amply as the code below

```
1  **function** connection() : solution
2          solution : s;
3          bool : finished = false;
4          **while not** finished **do**
```

```
5               bestLength = 0;
6               bestTime = ( 16 – ( 40 + 2 * capacity ) / 60 ) / 2;
7               for each pair routes in solution do
8                   find the routes which spend time exceeding bestTime
9                   if the number of routes > =1
10                      choose the longer route in such pair routes and split it into n new
11                      smaller routes
12                      n =int [(routeLength/speed) / bestTime] + 1
13                      for each splitting route do
14                          release all the m points in such route
15                          while m > 0 do
16                                for each point in the point union do
17                                    start from the depot and find the best insertion of the
18                                    points to form new routes
19                                    if newRouteLength < = routeLength / n
20                                    and if length of insertion > bestLength
21                                            store insertion;
22                                            m = m -1;
23                                            bestLength = length of insertion;
24                                    if number of new routes > n
25                                        stop
26                                    if there are points have not been selected
27                                        insert them into those formed new routes
28                                        go through all the routes in both parts and check
29                                        whether there are still pairs of routes exist.
30                                   if there is/are pair routes, go to step 2
31                                   else
32                                        Finished!
```

Note:

In the twelfth step, no matter the value of “int [(routeLength/speed) / bestTime]” is 0 or bigger than 0, we have to make sure n equals to 2 at least. Otherwise, there will be no change if n=1 again.

3.3 Experimental results

In this thesis, three important qualifications have been given. The first one is the number of point, the second one is the value of vehicle capacity and the third one is the bounded time period to vehicle. Those three instances are important because they have the crucial relationship to the final solutions.

Firstly, the number of point decides the complexity of calculation. In our problem, the number of points in each day is almost more than 200 and optimal results for VRP with only 100 points have been got so far. So in this experimental section, the first 100 customers will be selected from the original data in each day to test the algorithm.

Secondly, the length of route is formed under the constraint of vehicle capacity, which means if we change the value of capacity then all the routes will be changed also. So in this experimental section, we will assume that every vehicle will be added one trailer, which has the same capacity as the vehicle.

Third, the time period to vehicle is also very important. Since this value decides whether we should rearrange routes and how many routes need to be rearranged. This factor influences the efficiency of the algorithm. In this problem, 16 hours has been given. This value is reasonable in this current case, because 16 hours ensures that

- Time spent either in pickup part or delivery part is less than 16 hours. If there is overload route, we could just rearrange the routes in the worse part, but not both. So value of 16 simplifies the procedure of resolving overload routes.
- More number of overload routes will make the route length to be longer. 16 hours controls the number of overload.
- Less bounded time period may make us to rearrange the routes in both parts. And number of new routes will be increased, which brings bad solution.

Since on one hand the given bounded time period is reasonable, and on the other hand there is not enough time to make test of changing bounded time period. I display two groups of results and compare them under different constraints. There are four tables in the first group. The first two show some pertinent information of results coming from pickup and delivery parts with the original number of points and the capacity is 33. The second two tables show the results when capacity changes to be 66 without changing number of points.

After showing the results of route length, the time spent in computing are also calculated.

Note:

➢ maximum stepLength	We cannot get the best result if we set the value than it, but also cannot find better result if the stepLength value is set bigger than it
➢ overload route	The relative pair of routes whose total spending time exceeds the bound time from the given problem

Capacity = 33

Pickup part:

workday	number of customer	maximum stepLength	pairs of overload routes	result got before dealing with overload $\times 10^7$	final results $\times 10^7$
Day1	247	35	1	2.6040	2.6540
Day2	214	14	0	2.3023	2.3023
Day3	199	7	1	2.0881	2.1256
Day4	229	29	2	2.5359	2.6474
Day5	212	31	4	2.2232	2.2931

Table 3-3-1

Delivery part:

workday	number of customer	maximum stepLength	pairs of overload routes	result got before dealing with overload	final results
Day1	247	15	1	9085027	9085027
Day2	214	29	0	8379432	8379432
Day3	199	26	1	7914716	7914716
Day4	229	29	2	8483032	8483032
Day5	212	17	4	7789271	7789271

Table 3-3-2

Capacity = 66

Pickup part:

workday	number of customer	maximum stepLength	pairs of overload routes	result got before dealing with overload$\times 10^7$	final results $\times 10^7$
Day1	247	6	18	1.3950	1.7508
Day2	214	8	22	1.2200	1.6728
Day3	199	13	24	1.1335	1.6310
Day4	229	3	24	1.3085	1.8300
Day5	212	6	32	1.1632	1.6949

Table 3-3-3

Delivery part:

workday	number of customer	maximum stepLength	pairs of overload routes	result got before dealing with overload	final results
Day1	247	7	18	4816073	4816073
Day2	214	15	22	4263168	4263168
Day3	199	13	24	4212978	4230103
Day4	229	4	24	4834634	4840231
Day5	212	8	32	4393987	4393987

Table 3-3-4

Beside the display of results with different vehicle capacity, I also make tow tables show the value of consumed time in two different cases. The time is spent both in pickup part and delivery part.

➢ data: the given data at the beginning which includes the relevant information of every point and order

Capacity = 33, results : Computing time without reading data
Unitage : millisecond

workday	number of customer	pairs of overload routes	consuming time
Day1	247	1	9875
Day2	214	0	6213
Day3	199	1	5531
Day4	229	2	6375
Day5	212	4	5343

Table 3-3-5

Capacity = 66, results : Computing time without reading data
Unitage : millisecond

workday	number of customer	pairs of overload routes	consuming time
Day1	247	18	4344
Day2	214	22	4594
Day3	199	24	880
Day4	229	24	2031
Day5	212	32	3234

Table 3-3-6

Conclusion of the first group of results:

From the results displayed above, we can see that

- All the results in the delivery part before and after dealing with the overload problem are the same. So in most of the overload pairs of relative routes, the longer one between two routes is in the pickup part.
- When the value of capacity is bigger, then the routes will be also longer.
- The numbers of overload routes are more when route length increases
- The total route length with more capacity is less than the total route length with less capacity
- The time spending in computing with less capacity is more than the case with more capacity

Since optimal results for VRP with only 100 points has been explored by some guy so far, in the second group of results, the number of customers are set only to be 100. The way of fixing number of customers to be 100 is to select the first 100 customers from the data in every weekday.

Note:

- First part = pickup part
- Second part = delivery part

Customers = 100, capacity = 33,

weekday	pairs of overload routes	first part of results got before dealing with overload $\times 10^7$	final results of the first part $\times 10^7$	second part of results got before dealing with overload	final results of the second part
Day1	1	1.0832	1.1287	3485485	3485485
Day2	2	1.1587	1.2293	4172756	4172756
Day3	2	1.1020	1.1649	4229192	4229192
Day4	0	1.1149	1.1149	3883744	3883744
Day5	0	1.1156	1.1156	4274584	4274584

Table 3-3-7

Customers = 100, capacity = 66

weekday	pairs of overload routes	first part of results got before dealing with overload	final results of the first part	second part of results got before dealing with overload	final results of the second part
Day1	16	5694007	1.0214×10^7	1982503	1982053
Day2	16	6351351	9806012	2105782	2105782
Day3	9	5940909	9306530	2323729	2323729
Day4	11	6019874	1.0292×10^7	2364268	2364268
Day5	22	6299087	1.0069×10^7	2286409	2286409

Table 3-3-8

Capacity = 33, results : Computing time without reading data
Unitage : millisecond

workday	number of customer	pairs of overload routes	consuming time
Day1	100	1	2016
Day2	100	2	578
Day3	100	2	2360
Day4	100	0	1968
Day5	100	0	2265

Table3-3-9

Capacity = 66, results : Computing time without reading data
Unitage : millisecond

workday	number of customer	pairs of overload routes	Consuming time
Day1	100	16	2187
Day2	100	16	953
Day3	100	9	2235
Day4	100	23	1847
Day5	100	22	1562

Table3-3-10

Conclusion of the second group of results

- ➢ All the results in the delivery part before and after dealing with the overload problem are the same. So in most of the overload pairs of relative routes, the longer one between two routes is in the pickup part.
- ➢ The numbers of overload routes are more when route length increases
- ➢ The total route length with more capacity is less than the total route length with less capacity

Comparing with the conclusion got from the first group of results, there are only three left. When the data has been changed, some characters of results also change like the value of maxStepLength.

The value of maxStepLength can help us to get better results, but we should not fix value as the maxStepLength for any different cases. Because the maxStepLength is limited by two factors:

- ● The content of given data
- ● The number of routes

As we know that, if we have been given a new data. The value of the old maxStepLength will have no meaning. Once we are given new data, we have to test out new maxStepLength.

I have mentioned that the value of stepLength should not be increased up to the number of routes, because routes will stop being optimized after the stepLength reach to some value. Even though the stepLength can be increased bigger and bigger, but those bigger values of stepLength will not help routes to be optimized any further.If we make a test to test the first 100 points in the first weekday, we will realize different route length will be obtained by setting different stepLength and also understand the reason why maxStepLength need to be set to help us get the smallest value

Capacity = 33

Value of stepLength	**Value of route length** $\times 10^7$
3	1.0868743906
5	1.0842559938
6	1.0837876156
7	**1.0832486313**
10	**1.0832486313**
13	**1.0832486313**
15	**1.0832486313**

Table 3-3-11

It is clear to see that the maxStepLength is 7. Because when stepLength is smaller than 7, the result got by using such stepLength is bigger than the result got by setting 7 as the stepLength. But when the value of stepLength is bigger than 7, result will not change any more.

Moreover, if we do not care about the importance of setting maxStepLength, and choose a very big value as the stepLength, some dead circulation will happen when we run the algorithm.

So far, the algorithm has been composed. And we get the some important ideas after completing the algorithm, such as

- Only allow inserting point but not replacing
- Setting the maximum stepLength for iteration

By implementing the algorithm, the results are good but we do not know whether those results are optimal or not. In order to test that, I made a figure to display the distribution of the routes got from the second weekday.

From the table above, we know there are two pairs of overload routes in the second weekday when focusing on 100 points, then

Before handling the overload routes

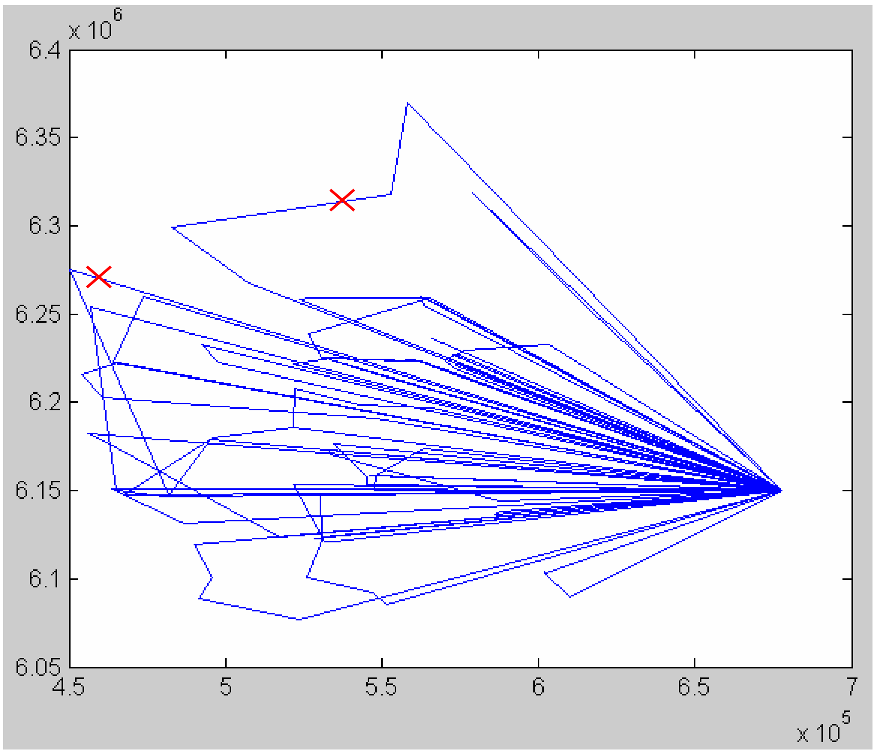


Figure 3-3-1

After handling the overload routes

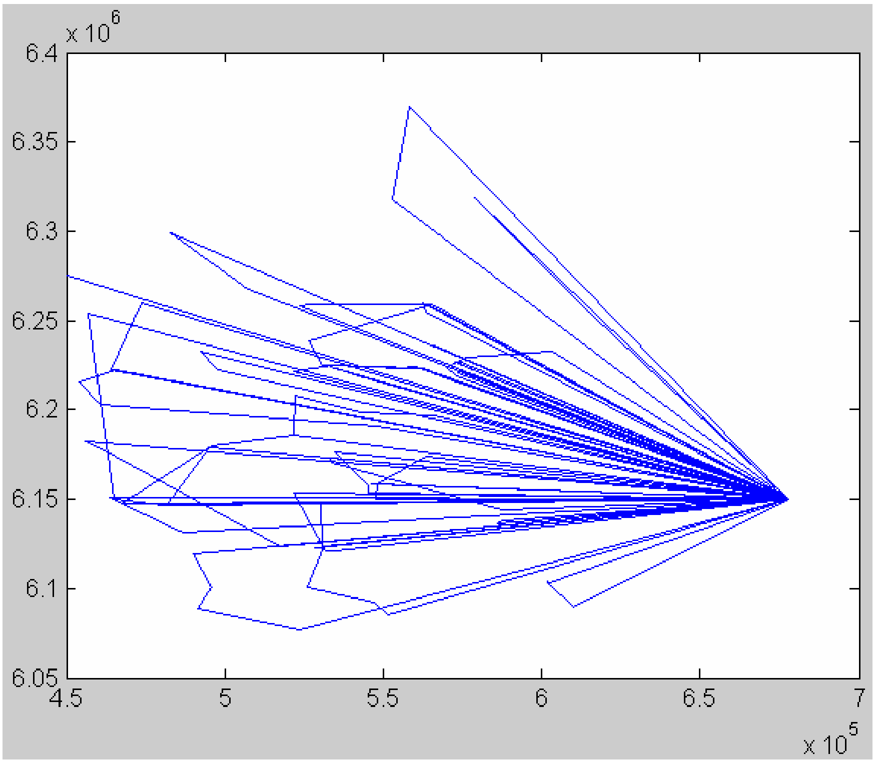


Figure 3-3-2

By handling the overload routes, it is obvious to find that longer routes in the figure ? are rearranged. Some new shorter routes are formed and displayed in the figure ?. But there are some routes overlap each other. No matter how, they do not influence the bounded the time period though there are two pair of overload routes in the whole problem. The action of transporting can be solved successfully even the algorithm has some drawback. The algorithm is definitely feasible.

4. Conclusions

This is the conclusion of the thesis, but research of this pickup and delivery with hub reloading problem is far away from the final ending. There is a long and hard route need to track. By doing this master project, I benefit well from this work. First, it lets me clearly realize the comprehensive application of VRP. Second, the attempt and exploration of approaches make me take the reins of basic methods of solving VRP systematically and consecutively, moreover the way of holistic thought how to modify the methods and get better ones. After finishing the project, I start to apprehend the VRP or PDP a little bit more and deeper.

In this thesis, I have displayed the mathematical model for the problem and adopted some algorithms. In the part of forming routes, the methods used mainly based on two aspects: Sweep_By_Angle and Insert_By_Distance. Those two aspects are discriminative and also correlative. Now, I will give some illuminations of two aspects both in disadvantages and advantages.

4.1 Based on Sweep_By_Angle and its pertinent approaches

1. Disadvantage

Because of emphasizing the idea of orderly distributing and attaching importance to the overlap of routes, the results got under those constraints are not perfect.

2. Advantage

It is a great gain of using tangent value of points to make a sequence for all the points around the same central depot. The application directly makes an effect to the part of method modification. There are also some inspirations from the disadvantage of the methods. We cannot seek the perfect solution blindly but also consider the actual instances. For instance, when the customers are plenty and they are clustered, we should not emphasize too much that overlap of routes need to avoid.

The use of the Sweep_By_Angle supports us a lot even it is failed. If there is no use of such method, I cannot easily agree that Inser_By_Distance is more reasonable and effective.

4.2 Based on Insert_By_Distance and its pertinent approaches

1. Disadvantage

At the beginning, because of too much limitation from the vehicle capacity, I insisted trying my best to compose a route. The total demand of points in such route should close to the capacity without considering how far the points between each other. At the end, there was a phenomenon that most of points in some routes were close to the depot and several points were far way from them, which made the route length very long.

2. Advantage

When I was in the procedure of searching points by using the way of Insert_By_Distance, though there were some bad results came out at the beginning, finally I found the method of Insert_By_Distance was effective from the solution And the results got from the modified Insert_By_Distance at the end had a big and good change.

4.3 Comparison and analysis of results

So far, the results got from the modification of Insert_By_Distance are the best excluding the new idea of methods. And I find I mostly rearrange the routes from the pickup part in the section of connecting correlative pair routes from different parts. The main reason for that is the pickup points are far from the central depot comparing with their corresponding terminal points. Further points produce longer routes, so I did the rearrangement mainly focusing on the pickup part.

But from the solution after rearrangement, there are some routes have been changed to be longer than before, I think people should pay some attention to this instance. There is probably more appropriate method will help us to get better solution.

4.4 Ideas of improving results

The VRP is very popular in research field. People have explored lots of algorithms for solving the VRP not only the heuristics but also metaheuristics. In this thesis, a feasible algorithm has been explored. In such algorithm, only inserting can be allowed, no point replacing exist and stepLength has been set to help us repeat the algorithm in feasible way.

Results got by applying the algorithm are good. There are not so many overload routes after forming and connecting relative routes in pickup and delivery part. The time spent in running the algorithm are short. Anyway, I am satisfied with this new algorithm. But there is still space and chance to improve the algorithm to be better and got better results.

In recent years, several metaheuristics have been explored out for VRP. There are six main types of metaheuristics used for VRP [12]: 1) Simulated Annealing; 2) Deterministic Annealing; 3) Tabu Search; 4) Genetic Algorithm; 5) Ant Systems; 6) Neural Networks. As the main character of metaheuristics allow deteriorating and infeasible intermediary solutions in the procedure of search process. Some mateheuristics could be tried to improve the existing results.

And in the algorithm, the acceptance of moving points only depends on shortening routes. If routes can be shorten, points will be considered to move in another route. Otherwise, no point moving can be accepted. Even there is a chance to get better solution in next section by acrossing the bad slution at current section. As

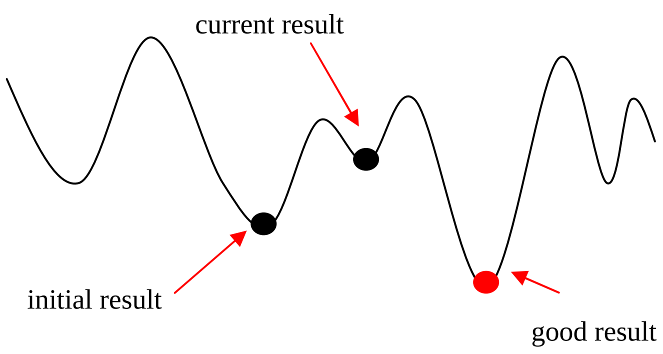


Figure 4-4-1

Because of the character of metaheuristics, if we accept to move points with some probability even bad results will be got, better results will be obtained later after passing by the bad solution first. Some of those metaheuristics displayed above can be adopted to improve the existed solution. Since there is no time for me implement any of them, people could try to use the metaheuristic to get better results if they are interested in problem also. More details can refer to the article made by G. Michel, L. Gilbert and P. Jean-Yves [12].

4.5 Comparison with the General Pickup and Delivery Problem

At the beginning of this thesis, I have mentioned that the pickup and delivery problem with hub reloading is a new transportation problem. Because the pickup and delivery parts in this problem will be handled by different vehicles. In General Pickup and Delivery Problem, the pickup and delivery parts are bounded together, which means one vehicle should be used to both picking up and delivering goods.

General Pickup and Delivery Problem

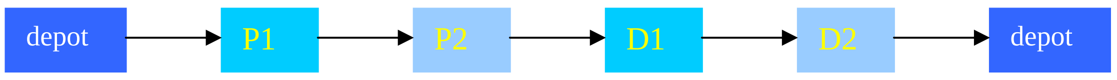


Figure 4-5-1

From the figure for the general PDP, one vehicle staring from the depot is used to pickup goods from different source points and delivery them to their corresponding destinations. This style of transportation has some limitations: if one vehicle visits some source points, it must also visit those source points' corresponding terminations. A restriction exists in the general PDP that vehicle cannot stop in an optimal way. If vehicle bounded with a fixed time has visited some pairs of points, source and terminal points, and if there is still redundant time for it to visit other source point but cannot visit its corresponding terminal one, a problem comes out: vehicles cannot use their bounded time period effectively.

But taking a look at this new PDP, the new PDP is much closer to our real life. Although, we consider that all the goods are the same without classifying in this thesis, it is more reasonable to transport different kinds of goods from their sources to their corresponding terminations. If we assume the process of settling in depot as a procedure of producing new goods, this new PDP is worthy and suitable for being considered in future.

5. References

[1] **M. L. Fisher,** ***Vehicle Routing*****. Operations and Information Management Department. The Wharton School, University of Pennsylvania, Philadelphia, PA 19104, U.S.A. page: 1-15**

[2] **N. Kohl, J. Desrosiers, O.B.G. Madsen, M. M. Solomon, F. Soumis,** ***2-Path Cuts for the Vehicle Routing Problem with Time Windows.*** **Transportation Science, 33(1), 101-116 (1999).**

[3] **G. Laporte and F. Semet,** ***Classical heuristics for the vehicle routing problem*****. In P. Toth, D. Vigo (Ed.),** ***The vehicle routing problem*****, SIAM, Philadelphia, page: 1-7, ISBN 0-89871-498-2.**

[4] **G. Clarke and J. W. Wright,** ***Scheduling if vehicles form a central depot to a number of delivery points*****. Operations Research, page: 568-581, 1964**

[5] **R.H. Mole and S.R. Jameson,** ***A sequential route-building algorithm employing a generalized saving criterion*****. Operational Research Quarterly page:503-511,1976**

[6] **N.Christofides, A. Mingozzi, and P.Toth,** ***The vehicle routing problem*****. In N.Christofides, A Mingozzi, p.Toch, and C. Sandi, editors, Combinatorial Optimization, page: 315-338. Wiley, Chrichester, 1979**

[7] **M. Junger, G. Reinelt, and G. Rinaldi,** ***The traveling salesman problem*****. Operations Research and Management Sciences: Networks (M. Ball, T. Magnanti, C.L. Monma, and G. Nemhauser, eds.), North-Holland, 1995, page. 225—330**

[8] **Hoong Chuin Lau, Zhe LiangT,** ***Pickup and Delivery with Time Windows: Algorithms and Test Case Generation*****. Page: 3-6, ICTAI 2001: 333-340**

[9] **M. L. Fisher and R. Jaikumar,** ***A Generalized Assignment Heuristic for Vehicle Routing*****. Networks, 11:109-124, 1981.**

[10] **R. Stefan,** ***Heuristics for the Multi-Vehicle Pickup and Delivery Problem with Time Windows*****. Master thesis, page: 45-79, 2002**

[11] **B. Olli,** ***Local Search and Variable Neighborhood Search Algorithms for the Vehicle Routing Problem with Time Windows*****. Ph.D thesis, page: 34-84, 2001**

[12] **G. Michel, L. Gilbert and P. Jean-Yves,** ***Metaheuristics for the Vehicle Routing Problem.*** **September, 1998 Revised: August, 1999. Les Cahiers du GERAD. G-98-52**

6. Appendix

6.1 Program of constructive methods:

****

Sweep_By_Angle:

```
import java.io.*;
import java.util.Vector;
import java.util.StringTokenizer;
import java.util.ArrayList;
import java.lang.String;
import java.util.Date;

/**
  *Demonstrate the way of getting the feasible routes in every weekday
  * @version (29_05_2005)
  */
public class Algorithm_Tan
{
        public static void main( int selectionDay)
        {
                System.out.println();
                System.out.println();
                System.out.println("TEST");


                Point depot = new Point( 0, 677908, 6150220, 33 );
                PointsUnion unionM = new PointsUnion();

                Date currentTime = new Date();
                long milSecond = currentTime.getTime();
                System.out.println( "Time Start to Read Data:" + milSecond );
                System.out.println();

                unionM.clearAllPoints();
                readFile( unionM, selectionDay );

                currentTime = new Date();
                milSecond = currentTime.getTime();
                System.out.println( "Time End:" + milSecond );
                System.out.println();

                //unionM.reducePointsUnionSize( 100 );


                currentTime = new Date();
                milSecond = currentTime.getTime();
                System.out.println( "Time Start of Algorithm Calculation:" + milSecond );
                System.out.println();


                /**
                  * Algorithm begins and some instances are initialized
                  */
                PointsUnion leftPoints = new PointsUnion();
```

```
            PointsUnion rightPoints = new PointsUnion();

            // Calculate the tan value for all of the points
            // and divides them into Left and Right parts
            for ( int i=0; i<unionM.getSize(); i++ )
            {
                  unionM.getPoint(i).calculateTan( depot );
                  if ( unionM.getPoint(i).getX() >= depot.getX() )
                        rightPoints.appendPoint( unionM.getPoint(i) );
                  else
                        leftPoints.appendPoint( unionM.getPoint(i) );
            }

            // Sort the points in left and right part
            rightPoints.sortPointsByTan();
            leftPoints.sortPointsByTan();

            // display rightPoints and leftPoints
            System.out.println( "All points: " + unionM.getSize() );
            System.out.println( "Right part points: " + rightPoints.getSize() );
            System.out.println( "Left part points: " + leftPoints.getSize() );
            System.out.println();
            System.out.println( "Right part points: " );
            for ( int i=0; i<rightPoints.getSize(); i++ )
                  System.out.println( "ID: " + rightPoints.getPoint(i).getID() + "         Tan(): " +
rightPoints.getPoint(i).getTan() );
            System.out.println();
            System.out.println( "Left part points: " );
            for ( int i=0; i<leftPoints.getSize(); i++ )
                  System.out.println( "ID: " + leftPoints.getPoint(i).getID() + "         Tan(): " +
leftPoints.getPoint(i).getTan() );

            // Calculate the routes
            ArrayList rightRoutes = new ArrayList();
            while ( rightPoints.getSize() > 0 )
            {
                  Route newRoute = new Route( depot );
                  for ( int i=0; i<rightPoints.getSize(); i++ )
                  {
                        // add a point if possible
                        if ( !newRoute.overLoad( rightPoints.getPoint(i) ) )
                        {
                              // add point
                              newRoute.appendPoint( rightPoints.getPoint(i) );
                              // delete point
                              rightPoints.removePoint( i );
                              // reset i to select the next point in next round
                              i--;
                        }
                  }
                  // save new route
                  rightRoutes.add( newRoute );
            }

            ArrayList leftRoutes = new ArrayList();
```

```
		while ( leftPoints.getSize() > 0 )
		{
			Route newRoute = new Route( depot );
			for ( int i=0; i<leftPoints.getSize(); i++ )
			{
				// add a point if possible
				if ( !newRoute.overLoad( leftPoints.getPoint(i) ) )
				{
					// add point
					newRoute.appendPoint( leftPoints.getPoint(i) );
					// delete point
					leftPoints.removePoint( i );
					// reset i to select the next point in next round
					i--;
				}
			}
			// save new route
			leftRoutes.add( newRoute );
		}

		// Show Result
		System.out.println();
		System.out.println();
		System.out.println( "The Result:" );
		System.out.println( "The right part:");
		double totalLength = 0.0d;
		Route newRoute = new Route( depot );
		for ( int i=0; i<rightRoutes.size(); i++ )
		{
			float rightRouteLength = 0.0f;
			System.out.print( "Right Route " + i + ":");
			newRoute = (Route)rightRoutes.get(i);
			for ( int j=0; j<newRoute.getSize(); j++ )
			{
				System.out.print(	"	"	+	newRoute.getPoint(j).getX()	+	","	+
newRoute.getPoint(j).getY() + "," + newRoute.getPoint(j).getGoods() );
				if ( j==0 )	// The first point connected with depot
					rightRouteLength += depot.getDistance( newRoute.getPoint(0) );
				else
					rightRouteLength += newRoute.getDistance( j, j-1 );
			}
			rightRouteLength += depot.getDistance( newRoute.getPoint(newRoute.getSize()-1) );
// finish the total length of the rightRoute
			System.out.println( "	Length:" + rightRouteLength );
			totalLength += rightRouteLength;
		}
		System.out.println();
		System.out.println( "The Total Length of " + rightRoutes.size() + " line(s): " + totalLength );

		System.out.println();
		System.out.println();
		System.out.println( "The left part:");
		totalLength = 0.0d;
		for ( int i=0; i<leftRoutes.size(); i++ )
		{
```

```
                float leftRouteLength = 0.0f;
                System.out.print( "Left Route " + i + ":");
                newRoute = (Route)leftRoutes.get(i);
                for ( int j=0; j<newRoute.getSize(); j++ )
                {
                    System.out.print(    "              "    +    newRoute.getPoint(j).getX()    +    ","    +
newRoute.getPoint(j).getY() + "," + newRoute.getPoint(j).getGoods() );
                    if ( j==0 )    // The first point connected with depot
                        leftRouteLength += depot.getDistance( newRoute.getPoint(0) );
                    else
                        leftRouteLength += newRoute.getDistance( j, j-1 );
                }
                leftRouteLength += depot.getDistance( newRoute.getPoint(newRoute.getSize()-1) ); //
finish the total length of the leftRoute
                System.out.println( "        Length:" + leftRouteLength );
                totalLength += leftRouteLength;
            }
            System.out.println();
            System.out.println( "The Total Length of " + leftRoutes.size() + " line(s): " + totalLength );


            currentTime = new Date();
            milSecond = currentTime.getTime();
            System.out.println( "Time End of Algorithm Calculation:" + milSecond );
            System.out.println();

        }


        private static void readFile( PointsUnion points, int selectedday )
        {
            try
            {
                //FileReader myFile = new FileReader( "D:\\BlueJ\\MyProject\\data.csv" );
                FileReader myFile = new FileReader( "data.csv" );

                if( myFile.ready() )
                {
                    System.out.println( "File Opened." );

                    BufferedReader bfr = new BufferedReader( myFile );
                    ArrayList dataAll = new ArrayList();
                    String line = bfr.readLine();
                    while ( line != null )
                    {
                        //System.out.println("READ>" + line + "<");
                        StringTokenizer tokenizer=new StringTokenizer(line,",");
                        Vector dataLine = new Vector();
                        for ( int i=0; i<7; i++ )
                        {
                            if ( tokenizer.hasMoreTokens() )
                            {
                                dataLine.addElement( tokenizer.nextToken() );
                            }
                            else
                            {
                                myFile.close();
                                System.out.println( "Data in the file error at line <ERROR>" + line );
                                return;
                            }
                        }
                        if ( tokenizer.hasMoreTokens() )
                        {
                            myFile.close();
                            System.out.println( "Data in the file error at line <ERROR>" + line );
                            return;
                        }
                        else
                        {
                            dataAll.add(dataLine);
                        }
                        line = bfr.readLine();
                    }

                    // Display dataAll
                    System.out.println( "dataAll:" );
                    for ( int i=0; i<dataAll.size(); i++ )
                    {
                        Vector oneline = (Vector)dataAll.get(i);
                        if ( string2int( (String)oneline.get(5) ) == selectedday )
                        {
                            for ( int j=0; j<oneline.size(); j++ )
                            {
                                line = (String)oneline.get(j);
                                if ( isAllNum(line) )
                                {
                                    //System.out.print( string2int(line) + " " );
                                }
                                else
                                {
                                    System.out.println( "Data in the file error at line <ERROR>" + i );
                                    return;
                                }
                            }
                            //System.out.println();
                            line = (String)oneline.get(1);
                            Point p = new Point( string2int( (String)oneline.get(0) ),
string2int( (String)oneline.get(1) ), string2int( (String)oneline.get(2) ),
string2int( (String)oneline.get(6) ) );
                            points.appendPoint( p );
                    }
                }
            }
            else
            {
                System.out.println( "Directory or File dose not exist." );
                }
            }
            catch ( Exception e )
            {
                System.out.println( "Cannot Find File" );
            }
        }

        private static boolean isAllNum( String s )
        {
            for ( int i=0; i<s.length(); i++ )
            {
                if ( ( s.charAt(i)<'0' ) || ( s.charAt(i)>'9') )
                return false;
            }
            return true;
        }

        private static int string2int( String s )
        {
            int out = 0;
            for ( int i=0; i<s.length(); i++ )
            {
                out *= 10;
                out += s.charAt(i) - 48;
            }
            return out;
        }
}
```

****

Insert_By_Distance

```
import java.io.*;
import java.util.Vector;
import java.util.StringTokenizer;
import java.util.ArrayList;
import java.lang.String;
import java.util.Date;

/**
  *Demonstrate the way of getting the feasible routes in every weekday
  * @version (29_05_2005)
  */
public class Algorithm
{
    private ArrayList pointsM = new ArrayList();

    public static void main(String[] args)
    {
        int selectionDay = 0;

        if ( args.length != 1 )
        {
            System.out.println("Please Input 1 Parameter.");
            return;
        }


        /**
  * The method of "isAllNum" has been given in the coming part, which
  * is used to remind user whether the imput of day is right or wrong.
  */
if ( !isAllNum( args[0] ) )
{
      System.out.println("Wrong Parameter!");
      return;
}
else

/**
  * Transfer the data of "args[]" from string to int since "selectionDay" is int
  */
      selectionDay = string2int( args[0] );

Point depot = new Point( 0, 677908, 6150220, 33 );
PointsUnion unionM = new PointsUnion();

Date currentTime = new Date();
long milSecond = currentTime.getTime();
System.out.println( "Time Start to Read Data:" + milSecond );
System.out.println();

unionM.clearAllPoints();
readFile( unionM, selectionDay );

currentTime = new Date();
milSecond = currentTime.getTime();
System.out.println( "Time End:" + milSecond );
System.out.println();

//unionM.reducePointsUnionSize( 100 );


currentTime = new Date();
milSecond = currentTime.getTime();
System.out.println( "Time Start of Algorithm Calculation:" + milSecond );
System.out.println();


/**
  * Algorithm begins and some instances are initialized
  */
Point startPoint = depot;
ArrayList routes = new ArrayList();
Route newRoute = new Route( depot ); //Every new route will start from the depot
do
{
      int selectedNode = -1; // no selected point at first
      float shortestDistance = 0.0f;


      /**
        * Loop for all of the points in unionM to find out the nearest points
        * to depot.
```

```
 */
for ( int j=0; j<unionM.getSize(); j++ )
{
    if ( !newRoute.overLoad( unionM.getPoint(j) ) )
    {
        if ( selectedNode == -1 )
        {
            selectedNode = j;
            shortestDistance = startPoint.getDistance( unionM.getPoint(j) );
        }
        else
        {
            float tempDistance = startPoint.getDistance( unionM.getPoint(j) );
            if ( tempDistance == shortestDistance )    // The 2 points have same
distance. Select one which contains more goods.
            {
                if ( unionM.getPoint(j).getGoods() >
unionM.getPoint(selectedNode).getGoods() )
                {
                    selectedNode = j;
                    // As 2 distances are same, need not change   shortestDistance.
                }
            }

            if ( tempDistance < shortestDistance )
            {
                selectedNode = j;
                shortestDistance = tempDistance;
            }
        }
    }
}

if ( selectedNode != -1 ) //some satisfied point has been searched
{
    if ( newRoute.appendPoint( unionM.getPoint(selectedNode) ) )
    {
        startPoint = unionM.getPoint(selectedNode); // reset startpoint
        unionM.removePoint( selectedNode ); // delete the selected point from
unionM
    }
    else
    {
        System.out.println("Adding method of ArrayList failed!");
        return;
    }
}
else   // no new point, this current route is finished
{
    // there are no points in the newroute
    if ( newRoute.getSize() == 0 )
    {
        if ( unionM.getSize() != 0 )
        {
```

```
                        System.out.println("There are/is point(s) left unselected. Algorithm
Failed!");
                    }
                    break;
                }

                // if there are points in the newroute, save this route
                if ( routes.add( newRoute ) )
                {
                    startPoint = depot; // reset startpoint
                    newRoute = new Route( depot );
                }
                else
                {
                    System.out.println("add method of ArrayList failed!!!");
                    return;
                }
            }
        }while ( true );

        // Show Result
        System.out.println();
        System.out.println();
        System.out.println( "The Result:" );
        double totalLength = 0.0d;
        for ( int i=0; i<routes.size(); i++ )
        {
            float routeLength = 0.0f;
            System.out.print( "Route " + i + ":");
            newRoute = (Route)routes.get(i);
            for ( int j=0; j<newRoute.getSize(); j++ )
            {
                System.out.print( "        " + newRoute.getPoint(j).getX() + "," +
newRoute.getPoint(j).getY() + "," + newRoute.getPoint(j).getGoods() );
                if ( j==0 )   // The first point connected with depot
                    routeLength += depot.getDistance( newRoute.getPoint(0) );
                else
                    routeLength += newRoute.getDistance( j, j-1 );
            }
            routeLength += depot.getDistance( newRoute.getPoint(newRoute.getSize()-1) ); //
finish the total length of the route
            System.out.println( "        Length:" + routeLength );
            totalLength += routeLength;
        }
        System.out.println();
        System.out.println( "The Total Length of " + routes.size() + " line(s): " + totalLength );

        currentTime = new Date();
        milSecond = currentTime.getTime();
        System.out.println( "Time End of Algorithm Calculation:" + milSecond );
        System.out.println();

    }
```

```
private static void readFile( PointsUnion points, int selectedday )
{
	try
	{
		//FileReader myFile = new FileReader( "D:\\BlueJ\\MyProject\\data.csv" );
		FileReader myFile = new FileReader( "data.csv" );

		if( myFile.ready() )
		{
			System.out.println( "File Opened." );

			BufferedReader bfr = new BufferedReader( myFile );
			ArrayList dataAll = new ArrayList();
			String line = bfr.readLine();
			while ( line != null )
			{
				System.out.println("READ>" + line + "<");
				StringTokenizer tokenizer=new StringTokenizer(line,",");
				Vector dataLine = new Vector();
				for ( int i=0; i<7; i++ )
				{
					if ( tokenizer.hasMoreTokens() )
					{
						dataLine.addElement( tokenizer.nextToken() );
					}
					else
					{
						myFile.close();
						System.out.println( "Data in the file error at line <ERROR>" + line );
						return;
					}
				}
				if ( tokenizer.hasMoreTokens() )
				{
					myFile.close();
					System.out.println( "Data in the file error at line <ERROR>" + line );
					return;
				}
				else
				{
					dataAll.add(dataLine);
				}
				line = bfr.readLine();
			}

			// Display dataAll
			System.out.println( "dataAll:" );
			for ( int i=0; i<dataAll.size(); i++ )
			{
				Vector oneline = (Vector)dataAll.get(i);
				if ( string2int( (String)oneline.get(5) ) == selectedday )
				{
					for ( int j=0; j<oneline.size(); j++ )
					{
						line = (String)oneline.get(j);
```

```
                        if ( isAllNum(line) )
                        {
                            System.out.print( string2int(line) + " " );
                        }
                        else
                        {
                            System.out.println( "Data in the file error at line <ERROR>" + i );
                            return;
                        }
                    }
                    System.out.println();
                    line = (String)oneline.get(1);
                    Point p = new Point( string2int( (String)oneline.get(0) ),
string2int( (String)oneline.get(1) ), string2int( (String)oneline.get(2) ),
string2int( (String)oneline.get(6) ) );
                    points.appendPoint( p );
                }
            }
        }
        else
        {
            System.out.println( "Directory or File dose not exist." );
        }
    }
    catch ( Exception e )
    {
        System.out.println( "Cannot Find File" );
    }
  }

  private static boolean isAllNum( String s )
  {
    for ( int i=0; i<s.length(); i++ )
    {
        if ( ( s.charAt(i)<'0' ) || ( s.charAt(i)>'9') )
            return false;
    }
    return true;
  }

  private static int string2int( String s )
  {
    int out = 0;
    for ( int i=0; i<s.length(); i++ )
    {
        out *= 10;
        out += s.charAt(i) - 48;
    }
    return out;
  }
}
```

****

Point

```
/**
```

```
 * Demonstrate the charactor of the points and the method of getting distance
 * bewteen two points.
 * @version (29_05_2005)
 */
public class Point
{
	// instance variables
	private int id;
	private int x;
	private int y;
	private int goods;
	private double tan;


	/**
	 * Constructor for objects of class Point
	 */
	public Point( int inputID, int inputX, int inputY, int inputGoods )
	{
		// initialise instance variables
		id = inputID;
		x = inputX;
		y = inputY;
		goods = inputGoods;
		tan = 0;
	}

	/**
	 * return ID
	 */
	public int getID()
	{
		return id;
	}


	/**
	 * return x
	 */
	public int getX()
	{
		return x;
	}


	/**
	 * return y
	 */
	public int getY()
	{
		return y;
	}


	/**
```

```
	 * return goods
	 */
	public int getGoods()
	{
		return goods;
	}


	/**
	 * return tan
	 */
	public double getTan()
	{
		return tan;
	}


	/**
	 * Calculate the distance between two Points
	 *
	 * @Parameter1: Another Point p
	 * return: distance between these 2 points
	 */
	public float getDistance( Point p )
	{
		double squareDis = ( (double)( x - p.getX() ) ) * ( (double)( x - p.getX() ) ) + ( (double)( y -
p.getY() ) ) * ( (double)( y - p.getY() ) );
		return (float) Math.sqrt( squareDis );
	}


	/**
	 * Calculate the tan() value between current Point and Depot
	 *
	 * @Parameter1: Depot point
	 * return: tan() value between these 2 points
	 */
	public void calculateTan( Point depot )
	{
		double disX = (double)( x - depot.getX() );
		if ( disX == 0.0d )
		{
			if ( y >= depot.getY() )
				tan = 9.999999999E9d;
			else
				tan = -9.999999999E9d;
		}
		double disY = (double)( y - depot.getY() );
		tan = (double)( disY/disX );
	}


	/**
	 * Clone a point
	 *
```

```
        * @Parameter1: point
        * return:
        */
       public void clonePoint( Point p )
       {
             id = p.getID();
             x = p.getX();
             y = p.getY();
             goods = p.getGoods();
             tan = p.getTan();
       }
```

****

PointUnion

```
import java.util.ArrayList;

/**
  * Demonstrate several actions of points in the algorithm process.
  * @version (29_05_2005)
  */
public class PointsUnion
{
 // instance variables
 private ArrayList union;


 /**
   * Constructor for objects of class PointsUnion
   */
 public PointsUnion()
 {
       // initialise instance variables
       union = new ArrayList();
 }


 /**
   * Appends a point to the end of this union.
   */
 public boolean appendPoint( Point p )
 {
       return( union.add( p ) );
 }

 /**
   * Deletes a point at specified position.
   */
 public void removePoint( int index )
 {
       union.remove(index);
 }


 /**
```

```
  * Clear all points
  */
public void clearAllPoints( )
{
      union.clear();
}


/**
  * Search for a specified point
  * Return: index of that point if found, -1 if not found
  */
public int searchPoint( Point p )
{
      return( union.indexOf( p ) );
}

/**
  * return the number of pointUnion
  */
public int getSize( )
{
      return( union.size() );
}


/**
  * Return the point at the specified index number
  */
public Point getPoint( int index )
{
      return( (Point)(union.get(index) ) );
}


/**
  * Return the distance between two points at the specified index numbers
  */
public float getDistance( int point1, int point2 )
{
      return( getPoint(point1).getDistance( getPoint(point2) ) );
}


/**
  * Return the distance between two points at the specified index numbers
  */
public void sortPointsByTan( )
{
      Point tmpPoint = new Point( 0, 0, 0, 0 );
      // sorting
      for ( int i=1; i<getSize(); i++ )
      {
            for ( int j=0; j<getSize()-i; j++ )
            {
                if ( getPoint(j).getTan() > getPoint(j+1).getTan() )
                {
                    tmpPoint.clonePoint( getPoint(j) );
                    getPoint(j).clonePoint(getPoint(j+1));
                    getPoint(j+1).clonePoint(tmpPoint);
                }
               }
        }
 }
}
```

6.2 program of modified methods

****

Based on Sweep_By_Angle

```
import java.io.*;
import java.util.Vector;
import java.util.StringTokenizer;
import java.util.ArrayList;
import java.lang.String;
import java.util.Date;

/**
 *Demonstrate the way of getting the feasible routes in every weekday
 * @version (30_06_2005)
 */
public class Algorithm_Tan
{
    public static void main( int selectionDay)
    {
        System.out.println();
        System.out.println();
        System.out.println("TEST");


        /**
         * At the beginning we can set some situation and make some examples to help us
         * test the algorithm
         */
        int[] datax = { 9, 7, 6, 9, 10 };
        int[] datay = { 7, 6, 3, 3, 5 };
        int[] datagoods = { 12, 8, 7, 13, 16 };

        /**
         * Add depot and all the points afterwards
         */
        Point depot = new Point( 0, 677908, 6150220, 33 );
        PointsUnion unionM = new PointsUnion();
      /*   for ( int i=0; i<datax.length; i++ )
        {
            Point p = new Point( datax[i], datay[i], datagoods[i] );
            if ( unionM.appendPoint(p) )
            {
                System.out.println( unionM.getPoint(i).getX() + " " + unionM.getPoint(i).getY() + " "
+ unionM.getPoint(i).getGoods() );
```

```
		}
		else
		{
			System.out.println( "Add Point Failed !");
			break;
		}
	}*/

	Date currentTime = new Date();
	long milSecond = currentTime.getTime();
	System.out.println( "Time Start to Read Data:" + milSecond );
	System.out.println();

	unionM.clearAllPoints();
	readFile( unionM, selectionDay );

		//unionM.reducePointsUnionSize( 100 );


	/**
	 * Algorithm begins and some instances are initialized
	 */
	PointsUnion leftPoints = new PointsUnion();
	PointsUnion rightPoints = new PointsUnion();

	// Calculate the tan value for all of the points
	// and divides them into Left and Right parts
	for ( int i=0; i<unionM.getSize(); i++ )
	{
		unionM.getPoint(i).calculateTan( depot );
		if ( unionM.getPoint(i).getX() >= depot.getX() )
			rightPoints.appendPoint( unionM.getPoint(i) );
		else
			leftPoints.appendPoint( unionM.getPoint(i) );
	}

	// Sort the points in left and right part
	rightPoints.sortPointsByTan();
	leftPoints.sortPointsByTan();

	// display rightPoints and leftPoints
	System.out.println( "All points: " + unionM.getSize() );
	System.out.println( "Right part points: " + rightPoints.getSize() );
	System.out.println( "Left part points: " + leftPoints.getSize() );
	System.out.println();
	System.out.println( "Right part points: " );
	for ( int i=0; i<rightPoints.getSize(); i++ )
		System.out.println(  "ID:  "  +  rightPoints.getPoint(i).getID()  +  "		Tan():  "  +
rightPoints.getPoint(i).getTan() );
	System.out.println();
	System.out.println( "Left part points: " );
	for ( int i=0; i<leftPoints.getSize(); i++ )
		System.out.println(  "ID:  "  +  leftPoints.getPoint(i).getID()  +  "		Tan():  "  +
leftPoints.getPoint(i).getTan() );
```

```
// Calculate the routes
ArrayList rightRoutes = new ArrayList();
while ( rightPoints.getSize() > 0 )
{
      Route newRoute = new Route( depot );
      for ( int i=0; i<rightPoints.getSize(); i++ )
      {
            // add a point if possible
            if ( !newRoute.overLoad( rightPoints.getPoint(i) ) )
            {
                  // Check for all of the points which have the same tan() value
                  if ( i != rightPoints.getSize()-1 )    // Not the last point
                  {
                        double tanvalue = rightPoints.getPoint(i).getTan();
                        int nexti = i + 1;
                        while ( rightPoints.getPoint(nexti).getTan() == tanvalue )
                        {
                              if  (  (  !newRoute.overLoad(  rightPoints.getPoint(nexti)  )  )  &&      //
Check capacity for point nexti
                                    (            rightPoints.getPoint(nexti).getDistance(depot)            >
rightPoints.getPoint(i).getDistance(depot) ) )      // Compare the 2 distances with the depot
                              {
                                    i = nexti;    // Use the farest for the route
                              }
                              nexti++;
                              if ( nexti == rightPoints.getSize() )      // The last point
                                    break;
                        }
                  }


                  // add point
                  newRoute.appendPoint( rightPoints.getPoint(i) );
                  // delete point
                  rightPoints.removePoint( i );
                  // reset i to select the next point in next round
                  i--;
            }
      }
      // save new route
      rightRoutes.add( newRoute );
}

ArrayList leftRoutes = new ArrayList();
while ( leftPoints.getSize() > 0 )
{
      Route newRoute = new Route( depot );
      for ( int i=0; i<leftPoints.getSize(); i++ )
      {
            // add a point if possible
            if ( !newRoute.overLoad( leftPoints.getPoint(i) ) )
            {
                  // Check for all of the points which have the same tan() value
                  if ( i != leftPoints.getSize()-1 )    // Not the last point
                  {
```

```
                                        double tanvalue = leftPoints.getPoint(i).getTan();
                                        int nexti = i + 1;
                                        while ( leftPoints.getPoint(nexti).getTan() == tanvalue )
                                        {
                                            if  (  (  !newRoute.overLoad(  leftPoints.getPoint(nexti)  )  )  &&      //
Check capacity for point nexti
                                                (             leftPoints.getPoint(nexti).getDistance(depot)             >
leftPoints.getPoint(i).getDistance(depot) ) )     // Compare the 2 distances with the depot
                                            {
                                                i = nexti;    // Use the farest for the route
                                            }
                                            nexti++;
                                            if ( nexti == leftPoints.getSize() )      // The last point
                                                break;
                                        }
                                    }

                                    // add point
                                    newRoute.appendPoint( leftPoints.getPoint(i) );
                                    // delete point
                                    leftPoints.removePoint( i );
                                    // reset i to select the next point in next round
                                    i--;
                                }
                            }
                            // save new route
                            leftRoutes.add( newRoute );
                        }

                        // Show Result
                        System.out.println();
                        System.out.println();
                        System.out.println( "The Result:" );
                        System.out.println( "The right part:");
                        double totalLength = 0.0d;
                        Route newRoute = new Route( depot );
                        for ( int i=0; i<rightRoutes.size(); i++ )
                        {
                            float rightRouteLength = 0.0f;
                            System.out.print( "Right Route " + i + ":");
                            newRoute = (Route)rightRoutes.get(i);
                            for ( int j=0; j<newRoute.getSize(); j++ )
                            {
                                System.out.print(     "              "     +     newRoute.getPoint(j).getX()     +     ","     +
newRoute.getPoint(j).getY() + "," + newRoute.getPoint(j).getGoods() );
                                if ( j==0 )    // The first point connected with depot
                                    rightRouteLength += depot.getDistance( newRoute.getPoint(0) );
                                else
                                    rightRouteLength += newRoute.getDistance( j, j-1 );
                            }
                            rightRouteLength  +=  depot.getDistance(  newRoute.getPoint(newRoute.getSize()-1)  );
// finish the total length of the rightRoute
                            System.out.println( "         Length:" + rightRouteLength );
                            totalLength += rightRouteLength;
                        }
```

```
        System.out.println();
        System.out.println( "The Total Length of " + rightRoutes.size() + " line(s): " + totalLength );

        System.out.println();
        System.out.println();
        System.out.println( "The left part:");
        totalLength = 0.0d;
        for ( int i=0; i<leftRoutes.size(); i++ )
        {
            float leftRouteLength = 0.0f;
            System.out.print( "Left Route " + i + ":");
            newRoute = (Route)leftRoutes.get(i);
            for ( int j=0; j<newRoute.getSize(); j++ )
            {
                System.out.print(    "         "    +    newRoute.getPoint(j).getX()    +    ","    +
newRoute.getPoint(j).getY() + "," + newRoute.getPoint(j).getGoods() );
                if ( j==0 )   // The first point connected with depot
                    leftRouteLength += depot.getDistance( newRoute.getPoint(0) );
                else
                    leftRouteLength += newRoute.getDistance( j, j-1 );
            }
            leftRouteLength += depot.getDistance( newRoute.getPoint(newRoute.getSize()-1) ); //
finish the total length of the leftRoute
            System.out.println( "       Length:" + leftRouteLength );
            totalLength += leftRouteLength;
        }
        System.out.println();
        System.out.println( "The Total Length of " + leftRoutes.size() + " line(s): " + totalLength );

        // Show Result again using ID instead of X,Y coordinates
        System.out.println();
        System.out.println();
        System.out.println( "The Result shows in ID:" );
        System.out.println( "The right part:");
        totalLength = 0.0d;
        for ( int i=0; i<rightRoutes.size(); i++ )
        {
            float rightRouteLength = 0.0f;
            System.out.print( "Right Route " + i + ":");
            newRoute = (Route)rightRoutes.get(i);
            for ( int j=0; j<newRoute.getSize(); j++ )
            {
                System.out.print( " " + newRoute.getPoint(j).getID() + "," );
                if ( j==0 )   // The first point connected with depot
                    rightRouteLength += depot.getDistance( newRoute.getPoint(0) );
                else
                    rightRouteLength += newRoute.getDistance( j, j-1 );
            }
            rightRouteLength += depot.getDistance( newRoute.getPoint(newRoute.getSize()-1) );
// finish the total length of the rightRoute
            System.out.println( "       Length:" + rightRouteLength );
            totalLength += rightRouteLength;
        }
        System.out.println();
        System.out.println( "The Total Length of " + rightRoutes.size() + " line(s): " + totalLength );
```

```
		System.out.println();
		System.out.println();
		System.out.println( "The left part:");
		totalLength = 0.0d;
		for ( int i=0; i<leftRoutes.size(); i++ )
		{
			float leftRouteLength = 0.0f;
			System.out.print( "Left Route " + i + ":");
			newRoute = (Route)leftRoutes.get(i);
			for ( int j=0; j<newRoute.getSize(); j++ )
			{
				System.out.print( " " + newRoute.getPoint(j).getID() + "," );
				if ( j==0 )    // The first point connected with depot
					leftRouteLength += depot.getDistance( newRoute.getPoint(0) );
				else
					leftRouteLength += newRoute.getDistance( j, j-1 );
			}
			leftRouteLength += depot.getDistance( newRoute.getPoint(newRoute.getSize()-1) ); //
finish the total length of the leftRoute
			System.out.println( "	Length:" + leftRouteLength );
			totalLength += leftRouteLength;
		}
		System.out.println();
		System.out.println( "The Total Length of " + leftRoutes.size() + " line(s): " + totalLength );


		currentTime = new Date();
		milSecond = currentTime.getTime();
		System.out.println( "Time End of Algorithm Calculation:" + milSecond );
		System.out.println();

	}


	private static void readFile( PointsUnion points, int selectedday )
	{
		try
		{
			//FileReader myFile = new FileReader( "D:\\BlueJ\\MyProject\\data.csv" );
			FileReader myFile = new FileReader( "data.csv" );

			if( myFile.ready() )
			{
				System.out.println( "File Opened." );

				BufferedReader bfr = new BufferedReader( myFile );
				ArrayList dataAll = new ArrayList();
				String line = bfr.readLine();
				while ( line != null )
				{
					//System.out.println("READ>" + line + "<");
					StringTokenizer tokenizer=new StringTokenizer(line,",");
					Vector dataLine = new Vector();
					for ( int i=0; i<7; i++ )
```

```
            {
                if ( tokenizer.hasMoreTokens() )
                {
                    dataLine.addElement( tokenizer.nextToken() );
                }
                else
                {
                    myFile.close();
                    System.out.println( "Data in the file error at line <ERROR>" + line );
                    return;
                }
            }
            if ( tokenizer.hasMoreTokens() )
            {
                myFile.close();
                System.out.println( "Data in the file error at line <ERROR>" + line );
                return;
            }
            else
            {
                dataAll.add(dataLine);
            }
            line = bfr.readLine();
        }

        // Display dataAll
        System.out.println( "dataAll:" );
        for ( int i=0; i<dataAll.size(); i++ )
        {
            Vector oneline = (Vector)dataAll.get(i);
            if ( string2int( (String)oneline.get(5) ) == selectedday )
            {
                for ( int j=0; j<oneline.size(); j++ )
                {
                    line = (String)oneline.get(j);
                    if ( isAllNum(line) )
                    {
                        //System.out.print( string2int(line) + " " );
                    }
                    else
                    {
                        System.out.println( "Data in the file error at line <ERROR>" + i );
                        return;
                    }
                }
                //System.out.println();
                line = (String)oneline.get(1);
                Point p = new Point( string2int( (String)oneline.get(0) ), string2int( (String)oneline.get(1) ), string2int( (String)oneline.get(2) ), string2int( (String)oneline.get(6) ) );
                points.appendPoint( p );
            }
        }
    }
    else
```

```
                {
                        System.out.println( "Directory or File dose not exist." );
                }
        }
        catch ( Exception e )
        {
                System.out.println( "Cannot Find File" );
        }
    }

    private static boolean isAllNum( String s )
    {
        for ( int i=0; i<s.length(); i++ )
        {
            if ( ( s.charAt(i)<'0' ) || ( s.charAt(i)>'9') )
                return false;
        }
        return true;
    }

    private static int string2int( String s )
    {
        int out = 0;
        for ( int i=0; i<s.length(); i++ )
        {
            out *= 10;
            out += s.charAt(i) - 48;
        }
        return out;
    }
}
```

****

Based on Insert_By_Distance

```
import java.io.*;
import java.util.Vector;
import java.util.StringTokenizer;
import java.util.ArrayList;
import java.lang.String;
import java.util.Date;

/**
 *Demonstrate the way of getting the feasible routes in every weekday
 * @version (30_06_2005)
 */
public class Algorithm
{
    private ArrayList pointsM = new ArrayList();

    public static void main(String[] args)
    {
        int selectionDay = 0;

        if ( args.length != 1 )
        {
```

```
        System.out.println("Please Input 1 Parameter.");
        return;
}


/**
 * The method of "isAllNum" has been given in the coming part, which
 * is used to remind user whether the imput of day is right or wrong.
 */
if ( !isAllNum( args[0] ) )
{
        System.out.println("Wrong Parameter!");
        return;
}
else

/**
 * Transfer the data of "args[]" from string to int since "selectionDay" is int
 */
        selectionDay = string2int( args[0] );

/**
 * Add depot and all the points afterwards
 */
Point depot = new Point( 0, 677908, 6150220, 33 );
PointsUnion unionM = new PointsUnion();

Date currentTime = new Date();
long milSecond = currentTime.getTime();
System.out.println( "Time Start to Read Data:" + milSecond );
System.out.println();

unionM.clearAllPoints();
readFile( unionM, selectionDay );

         //unionM.reducePointsUnionSize( 100 );

// Calculate the tan value for all of the points
for ( int i=0; i<unionM.getSize(); i++ )
        unionM.getPoint(i).calculateTan( depot );


/**
 * Algorithm begins and some instances are initialized
 */
Point startPoint = depot;
ArrayList routes = new ArrayList();
Route newRoute = new Route( depot ); //Every new route will start from the depot
do
{
        int selectedNode = -1; // no selected point at first
        float shortestDistance = 0.0f;


        /**
```

```
             * Loop for all of the points in unionM to find out the nearest points
             * to depot.
             */
            for ( int j=0; j<unionM.getSize(); j++ )
            {
                if ( !newRoute.overLoad( unionM.getPoint(j) ) )
                {
                    if ( selectedNode == -1 )
                    {
                        selectedNode = j;
                        shortestDistance = startPoint.getDistance( unionM.getPoint(j) );
                    }
                    else
                    {
                        float tempDistance = startPoint.getDistance( unionM.getPoint(j) );
                        if ( tempDistance == shortestDistance )     // The 2 points have same
distance. Select one which contains more goods.
                        {
                            if ( unionM.getPoint(j).getGoods() >
unionM.getPoint(selectedNode).getGoods() )
                            {
                                selectedNode = j;
                                // As 2 distances are same, need not change   shortestDistance.
                            }
                        }

                        if ( tempDistance < shortestDistance )
                        {
                            selectedNode = j;
                            shortestDistance = tempDistance;
                        }
                    }
                }
            }

            if ( selectedNode != -1 ) //some satisfied point has been searched
            {
                if ( newRoute.appendPoint( unionM.getPoint(selectedNode) ) )
                {
                    startPoint = unionM.getPoint(selectedNode); // reset startpoint
                    unionM.removePoint( selectedNode ); // delete the selected point from
unionM
                }
                else
                {
                    System.out.println("Adding method of ArrayList failed!");
                    return;
                }
            }
            else   // no new point, this current route is finished
            {
                // there are no points in the newroute
                if ( newRoute.getSize() == 0 )
                {
                    if ( unionM.getSize() != 0 )
                    {
                         System.out.println("There are/is point(s) left unselected. Algorithm Failed!");
                    }
                    break;
               }

               // if there are points in the newroute, save this route
               if ( routes.add( newRoute ) )
               {
                    startPoint = depot; // reset startpoint
                    newRoute = new Route( depot );
               }
               else
               {
                    System.out.println("add method of ArrayList failed!!!");
                    return;
               }
          }
     }while ( true );

     // Show Result
     System.out.println();
     System.out.println();
     System.out.println( "The Result:" );
     double totalLength = 0.0d;
     for ( int i=0; i<routes.size(); i++ )
     {
          float routeLength = 0.0f;
          System.out.print( "Route " + i + ":");
          newRoute = (Route)routes.get(i);
          for ( int j=0; j<newRoute.getSize(); j++ )
          {
               System.out.print( "          " + newRoute.getPoint(j).getX() + "," + newRoute.getPoint(j).getY() + "," + newRoute.getPoint(j).getGoods() );
               if ( j==0 )   // The first point connected with depot
                    routeLength += depot.getDistance( newRoute.getPoint(0) );
               else
                    routeLength += newRoute.getDistance( j, j-1 );
          }
          routeLength += depot.getDistance( newRoute.getPoint(newRoute.getSize()-1) ); // finish the total length of the route
          System.out.println( "        Length:" + routeLength );
          totalLength += routeLength;
     }
     System.out.println();
     System.out.println( "The Total Length of " + routes.size() + " line(s): " + totalLength );


     System.out.println();
     System.out.println();
     Result result = new Result( depot, routes );

     System.out.println( "Total Length: " + result.totalLength() + "        Total Points: " + result.totalPoints() );
```

```
        for ( int round=0; round<100; round++ )
        {
            result.sortRoutesByTan();
            result.optimize_Method_1( false );
            result.removeNullRoutes();
            //result.showResult_ID();
            System.out.println( "Total Length: " + result.totalLength() + "        Total Points: " +
result.totalPoints() );
            result.resetStatus();
        }

        currentTime = new Date();
        milSecond = currentTime.getTime();
        System.out.println( "Time End of Algorithm Calculation:" + milSecond );
        System.out.println();

    }


    private static void readFile( PointsUnion points, int selectedday )
    {
        try
        {
            //FileReader myFile = new FileReader( "D:\\BlueJ\\MyProject\\data.csv" );
            FileReader myFile = new FileReader( "data.csv" );

            if( myFile.ready() )
            {
                System.out.println( "File Opened." );

                BufferedReader bfr = new BufferedReader( myFile );
                ArrayList dataAll = new ArrayList();
                String line = bfr.readLine();
                while ( line != null )
                {
                    System.out.println("READ>" + line + "<");
                    StringTokenizer tokenizer=new StringTokenizer(line,",");
                    Vector dataLine = new Vector();
                    for ( int i=0; i<7; i++ )
                    {
                        if ( tokenizer.hasMoreTokens() )
                        {
                            dataLine.addElement( tokenizer.nextToken() );
                        }
                        else
                        {
                            myFile.close();
                            System.out.println( "Data in the file error at line <ERROR>" + line );
                            return;
                        }
                    }
                    if ( tokenizer.hasMoreTokens() )
                    {
                        myFile.close();
                        System.out.println( "Data in the file error at line <ERROR>" + line );
                        return;
                    }
                    else
                    {
                        dataAll.add(dataLine);
                    }
                    line = bfr.readLine();
                }

                // Display dataAll
                System.out.println( "dataAll:" );
                for ( int i=0; i<dataAll.size(); i++ )
                {
                    Vector oneline = (Vector)dataAll.get(i);
                    if ( string2int( (String)oneline.get(5) ) == selectedday )
                    {
                        for ( int j=0; j<oneline.size(); j++ )
                        {
                            line = (String)oneline.get(j);
                            if ( isAllNum(line) )
                            {
                                System.out.print( string2int(line) + " " );
                            }
                            else
                            {
                                System.out.println( "Data in the file error at line <ERROR>" + i );
                                return;
                            }
                        }
                        System.out.println();
                        line = (String)oneline.get(1);
                        Point p = new Point( string2int( (String)oneline.get(0) ),
string2int( (String)oneline.get(1) ) , string2int( (String)oneline.get(2) ),
string2int( (String)oneline.get(6) ) );
                        points.appendPoint( p );
                    }
                }
            }
            else
            {
                System.out.println( "Directory or File dose not exist." );
            }
        }
        catch ( Exception e )
        {
            System.out.println( "Cannot Find File" );
        }
    }

    private static boolean isAllNum( String s )
    {
        for ( int i=0; i<s.length(); i++ )
        {
            if ( ( s.charAt(i)<'0' ) || ( s.charAt(i)>'9') )
                return false;
        }
        return true;
    }

    private static int string2int( String s )
    {
        int out = 0;
        for ( int i=0; i<s.length(); i++ )
        {
            out *= 10;
            out += s.charAt(i) - 48;
        }
        return out;
    }
}
```

****

Point

```
/**
 * Demonstrate the charactor of the points and the method of getting distance
 * bewteen two points.
 * @version (30_06_2005)
 */
public class Point
{
    // instance variables
    private int id;
    private int x;
    private int y;
    private int goods;
    private double tan;


    /**
     * Constructor for objects of class Point
     */
    public Point( int inputID, int inputX, int inputY, int inputGoods )
    {
        // initialise instance variables
        id = inputID;
        x = inputX;
        y = inputY;
        goods = inputGoods;
        tan = 0;
    }

    /**
     * return ID
     */
    public int getID()
    {
        return id;
    }
	/**
	 * return x
	 */
	public int getX()
	{
		return x;
	}


	/**
	 * return y
	 */
	public int getY()
	{
		return y;
	}


	/**
	 * return goods
	 */
	public int getGoods()
	{
		return goods;
	}


	/**
	 * return tan
	 */
	public double getTan()
	{
		return tan;
	}


	/**
	 * Calculate the distance between two Points
	 *
	 * @Parameter1: Another Point p
	 * return: distance between these 2 points
	 */
	public float getDistance( Point p )
	{
		double squareDis = ( (double)( x - p.getX() ) ) * ( (double)( x - p.getX() ) ) + ( (double)( y - 
p.getY() ) ) * ( (double)( y - p.getY() ) );
		return (float) Math.sqrt( squareDis );
	}


	/**
	 * Calculate the tan() value between current Point and Depot
	 *
	 * @Parameter1: Depot point
	 * return: tan() value between these 2 points
```

```
          */
        public void calculateTan( Point depot )
        {
                double disX = (double)( x - depot.getX() );
                if ( disX == 0.0d )
                {
                        if ( y >= depot.getY() )
                                tan = 9.999999999E9d;
                        else
                                tan = -9.999999999E9d;
                }
                double disY = (double)( y - depot.getY() );
                tan = (double)( disY/disX );
        }


        /**
          * Clone a point
          *
          * @Parameter1: point
          * return:
          */
        public void clonePoint( Point p )
        {
                id = p.getID();
                x = p.getX();
                y = p.getY();
                goods = p.getGoods();
                tan = p.getTan();
        }


        /**
          * Clone a point
          *
          * @Parameter1: point
          * return:
          */
        public boolean equals( Point p )
        {
                if ( id != p.getID() )
                        return false;
                if ( x != p.getX() )
                        return false;
                if ( y != p.getY() )
                        return false;
                if ( goods != p.getGoods() )
                        return false;

                return true;
        }
}
```

****
PointUnion

```
import java.util.ArrayList;

/**
  * Demonstrate several actions of points in the algorithm process.
  * @version (30_06_2005)
  */
public class PointsUnion
{

        private ArrayList union;


        /**
          * Constructor for objects of class PointsUnion
          */
        public PointsUnion()
        {
                // initialise instance variables
                union = new ArrayList();
        }


        /**
          * Appends a point to the end of this union.
          */
        public boolean appendPoint( Point p )
        {
                return( union.add( p ) );
        }

        /**
          * Deletes a point at specified position.
          */
        public void removePoint( int index )
        {
                union.remove(index);
        }


        /**
          * Clear all points
          */
        public void clearAllPoints( )
        {
                union.clear();
        }


        /**
          * Search for a specified point
          * Return: index of that point if found, -1 if not found
          */
        public int searchPoint( Point p )
        {
                return( union.indexOf( p ) );
        }

        /**
          * return the number of pointUnion
          */
        public int getSize( )
        {
                return( union.size() );
        }


        /**
          * Return the point at the specified index number
          */
        public Point getPoint( int index )
        {
                return( (Point)(union.get(index) ) );
        }


        /**
          * Return the distance between two points at the specified index numbers
          */
        public float getDistance( int point1, int point2 )
        {
                return( getPoint(point1).getDistance( getPoint(point2) ) );
        }


        /**
          * Return the distance between two points at the specified index numbers
          */
        public void sortPointsByTan( )
        {
                Point tmpPoint = new Point( 0, 0, 0, 0 );
                // sorting
                for ( int i=1; i<getSize(); i++ )
                {
                    for ( int j=0; j<getSize()-i; j++ )
                    {
                        if ( getPoint(j).getTan() > getPoint(j+1).getTan() )
                        {
                            tmpPoint.clonePoint( getPoint(j) );
                            getPoint(j).clonePoint(getPoint(j+1));
                            getPoint(j+1).clonePoint(tmpPoint);
                        }
                    }
                }
        }


        /**
          * Return the distance between two points at the specified index numbers
          */
        public double calculateAllGoods( )
```

```
	{
		double total = 0.0d;
		for ( int i=0; i<union.size(); i++ )
			total += getPoint(i).getGoods();
		return total;
	}


	/**
	 * Return the distance between two points at the specified index numbers
	 */
	public int findPoint( Point p )
	{
		for ( int i=0; i<union.size(); i++ )
		{
			if ( getPoint(i).equals(p) )
				return i;
		}

		return -1;
	}
}
```

****

Route

```
/**
 * Demonstrate that the total demand of the route is the only contraint that
 * every route need to consider when define such route is feasible or not
 * @version (28_06_2005)
 */
public class Route extends PointsUnion
{

	private Point depot;
	public int optimizedTimes;
	public int removedPoints;
	public int addedPoints;
	public boolean selectedAsRoute1;


	/**
	 * Constructor for objects of class Route
	 */
	public Route( Point p )
	{
		super();
		depot = p;
		optimizedTimes = 0;
		removedPoints = 0;
		addedPoints = 0;
		selectedAsRoute1 = true;
	}


	public void resetStatus( )
	{
		optimizedTimes = 0;
		removedPoints = 0;
		addedPoints = 0;
		selectedAsRoute1 = true;
	}

	/**
	 * method for checking whether the total quantity of the goods exceeds the
	 * demand of each route or not
	 */
	public boolean overLoad( Point p )
	{
		int totalweight = 0;
		for ( int i=0; i<getSize(); i++ )
			totalweight += getPoint(i).getGoods();
		totalweight += p.getGoods();
		if ( totalweight <= depot.getGoods() )
			return false;
		else
			return true;
	}


	/**
	 * method for checking whether the total quantity of the goods exceeds the
	 * demand of each route or not
	 */
	public void resortByDistance( )
	{
		Point startPoint = depot;
		Route newRoute = new Route( depot ); //Every new route will start from the depot
		do
		{
			int selectedNode = -1; // no selected point at first
			float shortestDistance = 0.0f;

			/**
			 * Loop for all of the points in unionM to find out the nearest points
			 * to depot.
			 */
			for ( int j=0; j<getSize(); j++ )
			{
				if ( selectedNode == -1 )
				{
					selectedNode = j;
					shortestDistance = startPoint.getDistance( getPoint(j) );
				}
				else
				{
					float tempDistance = startPoint.getDistance( getPoint(j) );
					if ( tempDistance == shortestDistance )	// The 2 points have same distance.
Select one which contains more goods.
					{
						if ( getPoint(j).getGoods() > getPoint(selectedNode).getGoods() )
```

```
                        {
                            selectedNode = j;
                            // As 2 distances are same, need not change   shortestDistance.
                        }
                    }

                    if ( tempDistance < shortestDistance )
                    {
                        selectedNode = j;
                        shortestDistance = tempDistance;
                    }
                }
            }

            if ( selectedNode != -1 ) //some satisfied point has been searched
            {
                if ( newRoute.appendPoint( getPoint(selectedNode) ) )
                {
                    startPoint = getPoint(selectedNode); // reset startpoint
                    removePoint( selectedNode ); // delete the selected point from unionM
                }
                else
                {
                    System.out.println("Adding method of ArrayList failed!");
                    return;
                }
            }
            else   // no new point, this current route is finished
            {
                // there are no points in the newroute
                if ( newRoute.getSize() == 0 )
                {
                    if ( getSize() != 0 )
                    {
                        System.out.println("There   are/is   point(s)   left   unselected.   Algorithm
Failed!");
                    }
                    break;
                }

                // if there are points in the newroute, save this route
                for ( int j=0; j<newRoute.getSize(); j++ )
                {
                    if ( ! appendPoint( newRoute.getPoint(j) ) )
                    {
                        System.out.println("Adding method of ArrayList failed!");
                        return;
                    }
                }
                break;
            }
        }while ( true );
    }
```

```
	/**
	 * exchange Points, add insertpoint, delete a point
	 * return : integer>0: as can exchage successful, return the ID of the point need to exchange in
the route
	 *		-1: cannot perform exchage as the insertpoint is too large for it's goods.
	 */
	public int findExchangePoint( Point insertpoint )
	{
		int totalweight = 0;
		for ( int i=0; i<getSize(); i++ )
			totalweight += getPoint(i).getGoods();

		int selectedpoint = -1;
		Point deletePoint = new Point( 0, 0, 0, 0 );
		for ( int i=0; i<getSize(); i++ )
		{
			if ( ( totalweight - getPoint(i).getGoods() + insertpoint.getGoods() ) <=
depot.getGoods() )
			{
				if ( selectedpoint == -1 )
				{
					selectedpoint = i;
					deletePoint = getPoint(i);
				}
				if ( getPoint(i).getGoods() < deletePoint.getGoods() )
				{
					selectedpoint = i;
					deletePoint = getPoint(i);
				}
			}
		}
		return selectedpoint;
	}

	public void cloneRoute( Route r )
	{
		optimizedTimes = r.optimizedTimes;
		removedPoints = r.removedPoints;
		addedPoints = r.addedPoints;
		selectedAsRoute1 = r.selectedAsRoute1;

		for ( int i=getSize(); i>0; i-- )
			removePoint(i-1);
		for ( int i=0; i<r.getSize(); i++ )
			appendPoint( r.getPoint(i) );
	}


	public double getLength()
	{
		if ( getSize() == 0 )
			return 0.0d;

		double routeLength = 0.0d;
		for ( int j=0; j<getSize(); j++ )
```

```
            {
                  if ( j==0 )    // The first point connected with depot
                        routeLength += depot.getDistance( getPoint(0) );
                  else
                        routeLength += getDistance( j, j-1 );
            }
            routeLength += depot.getDistance( getPoint(getSize()-1) ); // finish the total length of the
route
            return routeLength;
      }
}
```

****

Result

```
import java.util.ArrayList;

/**
  * @version (30_06_2005)
  */
public class Result
{

      private Point depot;
      private Point zeroPoint;
      private Point leftPoint;
      private PointsUnion cannotBeAddedPoints;
      private int manipulatingPoint;
      private ArrayList routes;

      /**
        * Constructor for objects of class Result
        */
      public Result( Point resultDepot, ArrayList resultRoutes)
      {

            depot = resultDepot;
            routes = resultRoutes;
            zeroPoint = new Point( 0, 0, 0, 0 );
            leftPoint = new Point( 0, 0, 0, 0 );
            cannotBeAddedPoints = new PointsUnion();
      }



      public void resetStatus( )
      {
            for ( int i=0; i<routes.size(); i++ )
                    ((Route)routes.get(i)).resetStatus();

            for ( int i=cannotBeAddedPoints.getSize()-1; i>=0; i-- )
                  cannotBeAddedPoints.removePoint(i);

            leftPoint = zeroPoint;
      }
```

```
    public void showResult_ID( )
    {
        // Show Result again using ID instead of X,Y coordinates
        System.out.println();
        System.out.println();
        System.out.println( "The Result shows in ID:" );
        double totalLength = 0.0d;
        int totalPoints = 0;
        for ( int i=0; i<routes.size(); i++ )
        {
            float routeLength = 0.0f;
            System.out.print( "Route " + i + ":");
            Route newRoute = (Route)routes.get(i);
            for ( int j=0; j<newRoute.getSize(); j++ )
            {
                System.out.print( " " + newRoute.getPoint(j).getID() + "," );
                totalPoints++;
                if ( j==0 )    // The first point connected with depot
                    routeLength += depot.getDistance( newRoute.getPoint(0) );
                else
                    routeLength += newRoute.getDistance( j, j-1 );
            }
            routeLength  +=  depot.getDistance(  newRoute.getPoint(newRoute.getSize()-1)  );  //
finish the total length of the route
            System.out.println( "        Length:" + routeLength );
            totalLength += routeLength;
        }
        System.out.println();
        System.out.println( "The Total Length of " + routes.size() + " line(s): " + totalLength );
        System.out.println( "The Total Number of the Points: " + totalPoints );
    }



    public void removeNullRoutes( )
    {
        for ( int i=0; i<routes.size(); i++ )
        {
            Route newRoute = (Route)routes.get(i);
            if ( newRoute.getSize() == 0 )
                routes.remove(i);
        }
    }


    public double totalLength ( )
    {
        double tl = 0.0d;
        for ( int i=0; i<routes.size(); i++ )
              tl += ((Route)routes.get(i)).getLength();
        return tl;
    }
```

```
	public int totalPoints ( )
	{
		int tp = 0;
		for ( int i=0; i<routes.size(); i++ )
			tp += ((Route)routes.get(i)).getSize();
		return tp;
	}



	public void optimize_Method_1( boolean display )
	{
		if ( routes.size() <= 1 )
		{
			return;
		}

		boolean result = true;
		int selectedRoute = 0;
		int selectedRoute2 = 0;
		Route optimizeRoute1 = new Route( depot );
		Route optimizeRoute2 = new Route( depot );
		do
		{
			if ( result && (!leftPoint.equals(zeroPoint)) )
			{
				selectedRoute = selectedRoute2;
				optimizeRoute1 = (Route)routes.get(selectedRoute);
			}

			if ( leftPoint.equals(zeroPoint) )
			{
				// select route1, route1 here selected is the start route.
				selectedRoute = selectMinimumGoods();
				if ( selectedRoute == -1 )   // All routes have been added or removed point
					break;
				else
					optimizeRoute1 = (Route)routes.get(selectedRoute);
			}

			selectedRoute2 = selectedRoute;
			selectedRoute2++;
			if ( selectedRoute2 == routes.size() )
				selectedRoute2 = 0;
			optimizeRoute2 = (Route)routes.get(selectedRoute2);

			if  (display)   System.out.print(  "Try  Optimizing:  "  +  selectedRoute  +  "        "  +
selectedRoute2 + ".    " );
			result = Optimize_Function_1( optimizeRoute1, optimizeRoute2 );
			optimizeRoute1.selectedAsRoute1 = false;
			if (display)   System.out.print( "Selected Point ID:" + manipulatingPoint + ".      " );
			if ( result )
			{
				// change the selectedAsRoute1 of the previous route of route1 if necessary
				int preRoute;
```

```
                if ( selectedRoute == 0 )
                    preRoute = routes.size() - 1;
                else
                    preRoute = selectedRoute - 1;
                if ( ( ( (Route)routes.get(preRoute) ).selectedAsRoute1 == false ) &&
( ( (Route)routes.get(preRoute) ).removedPoints == 0 ) )   // preRoute has been selected as route1
and failed optimization at that time.
                    ( (Route)routes.get(preRoute) ).selectedAsRoute1 = true;

                if (display)
                {
                    System.out.print( "Success.    " );
                    if ( leftPoint.equals(zeroPoint) )
                        System.out.println( "One point added.");
                    else
                        System.out.println( "One point replaced.");
                }
            }
            else
                if (display)    System.out.println( "Failed." );

        } while ( true );

        // Deal with remaining cannotBeAddedPoints
        if ( cannotBeAddedPoints.getSize() == 0 )
            return;
        if (display)
        {
            System.out.println();
            System.out.print( "Remaining Points:" );
            for ( int i=0; i<cannotBeAddedPoints.getSize(); i++ )
                System.out.print( "    " + cannotBeAddedPoints.getPoint(i).getID() );
            System.out.println();
            System.out.println( "Dealing with remaining cannotBeAddedPoints:" );
        }

        /*
        // test!!!!!!!
        for ( int i=0; i<cannotBeAddedPoints.getSize(); i++ )
            addOnePointInRoutes( cannotBeAddedPoints.getPoint(i), display );
        for ( int i=cannotBeAddedPoints.getSize()-1; i>=0; i-- )
            cannotBeAddedPoints.removePoint(i);
        */

        cannotBeAddedPoints.sortPointsByTan();    // sort the points
        while ( cannotBeAddedPoints.getSize() > 0 )          // there are points left in
cannotBeAddedPoints
        {
            // Create one empty route
            Route newRoute = new Route( depot );

            // add first point
            Point firstPoint = cannotBeAddedPoints.getPoint(0);
            newRoute.appendPoint( cannotBeAddedPoints.getPoint(0) );
```

```
			// delete this point from cannotBeAddedPoints
			cannotBeAddedPoints.removePoint(0);

			if ( cannotBeAddedPoints.getSize() == 0 )	// only first point left in
cannotBeAddedPoints
			{
				routes.add( newRoute ); // save this route

				// Show the added route
				if (display)
				{
					System.out.print( "Added in Route " + ( routes.size()-1 ) + " :");
					for ( int i=0; i<((Route)routes.get(routes.size()-1)).getSize(); i++ )
						System.out.print(		"		"		+
((Route)routes.get(routes.size()-1)).getPoint(i).getID() );
					System.out.println();
				}

				break;
			}

			// add remaining points if possible
			while ( cannotBeAddedPoints.getSize() > 0 )
			{
				Point checkPoint = cannotBeAddedPoints.getPoint(0);
				if	(	(	!newRoute.overLoad(checkPoint)	)	&&
( findRouteBetweenTwoPoints( firstPoint, checkPoint ) == -1 ) )
				//if ( !newRoute.overLoad(checkPoint) )
				{
					// add a new point
					newRoute.appendPoint( checkPoint ); // add check point
					cannotBeAddedPoints.removePoint(0); // delete it from cannotBeAddedPoints

					if ( cannotBeAddedPoints.getSize() == 0 )	// no point left in
cannotBeAddedPoints
					{
						routes.add( newRoute ); // save this route

						// Show the added route
						if (display)
						{
							System.out.print( "Added in Route " + ( routes.size()-1 ) + " :");
							for ( int i=0; i<((Route)routes.get(routes.size()-1)).getSize(); i++ )
								System.out.print(		"		"		+
((Route)routes.get(routes.size()-1)).getPoint(i).getID() );
							System.out.println();
						}

						break;
					}
				}
				else
				{
					// cannot add new point. Terminate this route
					newRoute.resortByDistance(); // connect the route in order
```

```
                    routes.add( newRoute ); // save this route

                    // Show the added route
                    if (display)
                    {
                         System.out.print( "Added in Route " + ( routes.size()-1 ) + " :");
                         for ( int i=0; i<((Route)routes.get(routes.size()-1)).getSize(); i++ )
                              System.out.print(          "                    "           +
((Route)routes.get(routes.size()-1)).getPoint(i).getID() );
                         System.out.println();
                    }

                    break;   // start a new route
               }
          }
     }
}


public void sortRoutesByTan( )
{
     PointsUnion leftOrder = new PointsUnion();
     PointsUnion rightOrder = new PointsUnion();

     for ( int i=0; i<routes.size(); i++ )
     {
          Route thisRoute = (Route)routes.get(i);
          int selectedpoint = thisRoute.getSize()-1; // Last point
          if          (          (thisRoute.getPoint(0).getX()-depot.getX())          *
(thisRoute.getPoint(selectedpoint).getX()-depot.getX()) >= 0 )
          {
               if ( thisRoute.getPoint(0).getTan() > thisRoute.getPoint(selectedpoint).getTan() )
                    selectedpoint = 0;
          }
          else
          {
               if ( thisRoute.getPoint(0).getTan() < thisRoute.getPoint(selectedpoint).getTan() )
                    selectedpoint = 0;
          }

          Point newPoint = new Point( 0, 0, 0, 0 );
          newPoint.clonePoint( thisRoute.getPoint(selectedpoint) );

          if ( newPoint.getX() > depot.getX() )   // right side
               rightOrder.appendPoint( newPoint );
          else      // left side
               leftOrder.appendPoint( newPoint );
     }

     leftOrder.sortPointsByTan();
     rightOrder.sortPointsByTan();

     // rightOrder + leftOrder = orderPoints;
     PointsUnion orderPoints = new PointsUnion();
     for ( int i=0; i<rightOrder.getSize(); i++ )
```

```
            orderPoints.appendPoint( rightOrder.getPoint(i) );
        for ( int i=0; i<leftOrder.getSize(); i++ )
            orderPoints.appendPoint( leftOrder.getPoint(i) );

        // create a copy for routes
        ArrayList temp = new ArrayList();
        for ( int i=0; i<routes.size(); i++ )
            temp.add( routes.get(i) );

        // clear routes to wait for routes inserting in order
        routes.clear();

        // move each route into routes in the order
        if ( orderPoints.getSize() != temp.size() )
        {
            System.out.println( "ERROR happens in sortRoutesByTan !!!" );
            System.out.println( "ERROR happens in sortRoutesByTan !!!" );
            System.out.println( "ERROR happens in sortRoutesByTan !!!" );
            return;
        }
        for ( int i=0; i<orderPoints.getSize(); i++ )
        {
            Point orderPoint = orderPoints.getPoint(i);
            int j;
            for ( j=0; j<temp.size(); j++ )
            {
                // check whether orderPoint in route temp[j]
                if ( ((Route)temp.get(j)).findPoint(orderPoint) != -1 )
                {
                    routes.add( (Route)temp.get(j) );
                    break;
                }
            }

            if ( j == temp.size() )
            {
                System.out.println( "ERROR happens in sortRoutesByTan !!!" );
                System.out.println( "ERROR happens in sortRoutesByTan !!!" );
                System.out.println( "ERROR happens in sortRoutesByTan !!!" );
                return;
            }
        }

        // check routes at last
        if ( orderPoints.getSize() != routes.size() )
        {
            System.out.println( "ERROR happens in sortRoutesByTan !!!" );
            System.out.println( "ERROR happens in sortRoutesByTan !!!" );
            System.out.println( "ERROR happens in sortRoutesByTan !!!" );
            return;
        }
    }
```

```
	private int selectMinimumGoods( )
	{
		// Select the minimum demand route which has not been optimized
		double minimumGoods = 0.0d;
		int selectedRoute = -1;
		for ( int i=0; i<routes.size(); i++ )
		{
			Route newRoute = (Route)routes.get(i);
			// Check whether has been optimized
			if ( ( newRoute.addedPoints > 0 ) || ( !newRoute.selectedAsRoute1 ) )
			{
				continue;
			}
			else
			{
				// Is it the first one which has not been optimized
				if ( selectedRoute == -1 )
				{
					selectedRoute = i;
					minimumGoods = newRoute.calculateAllGoods();
				}
				else	// Compare and select minimum
				{
					double temp = newRoute.calculateAllGoods();
					if ( temp < minimumGoods )
					{
						minimumGoods = temp;
						selectedRoute = i;
					}
				}
			}
		}

		return selectedRoute;
	}


	private boolean Optimize_Function_1( Route route1, Route route2 )
	{
		// Check if left point need to insert into route2
		if ( leftPoint.equals( zeroPoint ) )
		{
			// no point left. Select a proper point from route1
			int selectedpoint = route1.getSize()-1; // Last point
			if		(		(route1.getPoint(0).getX()-depot.getX())		*
(route1.getPoint(selectedpoint).getX()-depot.getX()) >= 0 )
			{
				if ( route1.getPoint(0).getTan() > route1.getPoint(selectedpoint).getTan() )
					selectedpoint = 0;
			}
			else
			{
				if ( route1.getPoint(0).getTan() < route1.getPoint(selectedpoint).getTan() )
					selectedpoint = 0;
			}
```

```
            // select that point
            leftPoint = route1.getPoint(selectedpoint); // save the point as leftPoint
        }

        // record selected point ID.
        manipulatingPoint = leftPoint.getID();

        // add leftPoint into route2
        if ( route2.overLoad( leftPoint ) )
        {
            // overload
            int deletepoint = route2.findExchangePoint( leftPoint );
            if ( deletepoint == -1 )
            {
                // exchange failed, cannot exchange leftPoint into route2
                if ( route1.addedPoints > 0 ) // route1 is not a start route
                {
                    // remove the leftPoint in route1
                    int selectedpoint = route1.findPoint( leftPoint );
                    if ( selectedpoint < 0 )
                    {
                        System.out.println( "ERROR   ERROR   ERROR   ERROR   ERROR !!!
" );
                        System.out.println( "ERROR   ERROR   ERROR   ERROR   ERROR !!!
" );
                        System.out.println( "ERROR   ERROR   ERROR   ERROR   ERROR !!!
" );
                        return false;
                    }
                    route1.removePoint( selectedpoint );
                    route1.removedPoints++;
                    route1.resortByDistance();

                    // and save it into cannotBeAddedPoints
                    cannotBeAddedPoints.appendPoint( leftPoint );

                    leftPoint = zeroPoint; // reset leftPoint
                }
                else      // route1 is a start route
                {
                    leftPoint = zeroPoint; // reset leftPoint
                }

                return false;
            }
            else
            {
                // exchange successful
                int selectedpoint = route1.findPoint( leftPoint );
                if ( selectedpoint < 0 )
                {
                    System.out.println( "ERROR   ERROR   ERROR   ERROR   ERROR !!! " );
                    System.out.println( "ERROR   ERROR   ERROR   ERROR   ERROR !!! " );
                    System.out.println( "ERROR   ERROR   ERROR   ERROR   ERROR !!! " );
                    return false;
                }
                route1.removePoint( selectedpoint );
                route1.removedPoints++;
                route1.resortByDistance();

                route2.appendPoint( leftPoint );      // add left point
                leftPoint = route2.getPoint(deletepoint); // set new left point
                route2.resortByDistance();
                route2.addedPoints++;
                return true;
            }
        }
        else       // not overload
        {
            int selectedpoint = route1.findPoint( leftPoint );
            if ( selectedpoint<0 )
            {
                System.out.println( "ERROR   ERROR   ERROR   ERROR   ERROR !!! " );
                System.out.println( "ERROR   ERROR   ERROR   ERROR   ERROR !!! " );
                System.out.println( "ERROR   ERROR   ERROR   ERROR   ERROR !!! " );
                return false;
            }
            route1.removePoint( selectedpoint );
            route1.removedPoints++;
            route1.resortByDistance();

            route2.appendPoint( leftPoint );
            route2.addedPoints++;
            route2.resortByDistance();

            leftPoint = zeroPoint; // reset leftPoint
            return true;
        }
    }


    /**
     * find the first route in which all of the points lie between two input points
     * return : integer>=0, the index of the corresponding route in routes
     *               -1: cannot find the corresponding route
     */
    private int findRouteBetweenTwoPoints( Point p1, Point p2 )
    {
        double smalltan = p1.getTan();
        double largetan = p2.getTan();
        if ( smalltan>largetan) // exchage them
        {
            double temp = smalltan;
            smalltan = largetan;
            largetan = temp;
        }

        for ( int i=0; i<routes.size(); i++ )
        {
            Route checkRoute = (Route)routes.get(i);
```

```
			boolean allRouteIn = true;
			for ( int j=0; j<checkRoute.getSize(); j++ )
			{
				double checkTan = checkRoute.getPoint(j).getTan();
				if ( ( checkTan < smalltan ) || (checkTan > largetan ) )
				{
					allRouteIn = false;
					break;
				}
			}
			if ( allRouteIn )
				return i;
		}

		return -1;
	}


	private void addOnePointInRoutes( Point p , boolean display)
	{
		double smallestDis = depot.getDistance(p) * 2 ;
		int selectedRoute = -1;

		for ( int i=0; i<routes.size(); i++ )
		{
			if ( ((Route)routes.get(i)).overLoad(p) )
				continue;
			else
			{
				Route testRoute = new Route( depot );
				testRoute.cloneRoute( (Route)routes.get(i) );
				testRoute.appendPoint(p);
				testRoute.resortByDistance();
				double		increasedLength		=		testRoute.getLength()		-
((Route)routes.get(i)).getLength();
				if ( increasedLength < 0.0d )
				{
					System.out.println( "ERROR ERROR ERROR" );
				}
				if ( increasedLength<smallestDis )
				{
					selectedRoute = i;
					smallestDis = increasedLength;
				}
			}
		}

		if ( selectedRoute == -1 )
		{
			Route newRoute = new Route( depot );
			newRoute.appendPoint(p);
			routes.add(newRoute);
			if (display)	System.out.println( "Add point " + p.getID() + " in a sing-point new Route "
+ (routes.size()-1) );
		}
```

```
            else
            {
                  ((Route)routes.get(selectedRoute)).appendPoint(p);
                  ((Route)routes.get(selectedRoute)).resortByDistance();
                  if (display)    System.out.println( "Add point " + p.getID() + " in Route " +
selectedRoute );
            }
      }
}
```

6.3 final program of algorithm

```
***************************************************************************************************************
```

Algorithm

```
import java.io.*;
import java.util.Vector;
import java.util.StringTokenizer;
import java.util.ArrayList;
import java.util.Date;
import java.lang.String;

/**
 *Demonstrate the way of getting the feasible routes in every weekday
 * @version (28_08_2005)
 */
public class Algorithm
{
      private ArrayList pointsM = new ArrayList();

      public static void main(String[] args)
      {
            int selectionDay = 0;

            if ( args.length != 1 )
            {
                  System.out.println("Please Input 1 Parameter.");
                  return;
            }


            /**
             * The method of "isAllNum" has been given in the coming part, which
             * is used to remind user whether the imput of day is right or wrong.
             */
            if ( !isAllNum( args[0] ) )
            {
                  System.out.println("Wrong Parameter!");
                  return;
            }
            else
            {
                   /**
                     * Transfer the data of "args[]" from string to int since "selectionDay" is int
                   **/
                  selectionDay = string2int( args[0] );
```

```
		}

		Date currentTime = new Date();
		long milSecond = currentTime.getTime();
		System.out.println( "Time Start to Read Data:" + milSecond );
		System.out.println();

		/**
		  * Add depot and all the points afterwards
		  */
		Point depot = new Point( 0, 677908, 6150220, 66 );
		PointsUnion unionM = new PointsUnion();
		PointsUnion unionM1 = new PointsUnion();
		PointsUnion unionM2 = new PointsUnion();
		readFile( unionM1, unionM2, selectionDay );
		unionM1.reducePointsUnionSize( 100 );
		unionM2.reducePointsUnionSize( 100 );

		currentTime = new Date();
		milSecond = currentTime.getTime();
		System.out.println( "Time End:" + milSecond );
		System.out.println();


		currentTime = new Date();
		milSecond = currentTime.getTime();
		System.out.println( "Time Start of Algorithm Calculation:" + milSecond );
		System.out.println();

		// Calculate the tan value for all of the points
		for ( int i=0; i<unionM1.getSize(); i++ )
			unionM1.getPoint(i).calculateTan( depot );
		for ( int i=0; i<unionM2.getSize(); i++ )
			unionM2.getPoint(i).calculateTan( depot );

		// First Part
		ArrayList routes1 = new ArrayList();
		unionM1.generateRoutesByDistance( depot, routes1 );
		Result result1 = new Result( depot, routes1 );
		result1.showResult_ID();
		System.out.println( "Total Length: " + result1.totalLength() + "		Total Points: " +
result1.totalPoints() );

		int stepLength = 1;
		boolean usingResetSteplength = true;
		boolean resetStepLength;
		resetStepLength = usingResetSteplength;
		do
		{
			result1.sortRoutesByTan();
			result1.optimize_Method_1( stepLength, true, false );
			result1.removeNullRoutes();
			result1.optimize_Method_1( stepLength, false, false );
			result1.removeNullRoutes();
```

```
                if ( result1.noChange() )
                {
                    stepLength++;

                    // reset stepLength
                    if ( resetStepLength )
                    {
                        stepLength = 1;
                        resetStepLength = false;
                    }

                    if ( stepLength > 6 )    // reach MAX stepLength
                        break;
                    else
                    {
                        System.out.println( "Change stepLength to: " + stepLength );
                        continue;
                    }
                }
                else
                {
                    resetStepLength = usingResetSteplength;
                }

                //result.showResult_ID();
                System.out.println( "Total Length: " + result1.totalLength() + "        Total Points: " +
result1.totalPoints() );
                result1.resetStatus();
            } while ( true );

            // Final Result
            System.out.println( "Final Result of First Part:");
            result1.showResult_ID();

            // Second Part
            ArrayList routes2 = new ArrayList();
            unionM2.generateRoutesByDistance( depot, routes2 );
            Result result2 = new Result( depot, routes2 );
            result2.showResult_ID();
            System.out.println( "Total Length: " + result2.totalLength() + "        Total Points: " +
result2.totalPoints() );

            stepLength = 1;
            usingResetSteplength = true;
            resetStepLength = false;
            resetStepLength = usingResetSteplength;
            do
            {
                result2.sortRoutesByTan();
                result2.optimize_Method_1( stepLength, true, false );
                result2.removeNullRoutes();
                result2.optimize_Method_1( stepLength, false, false );
                result2.removeNullRoutes();

                if ( result2.noChange() )
```

```
				{
					stepLength++;

					// reset stepLength
					if ( resetStepLength )
					{
						stepLength = 1;
						resetStepLength = false;
					}

					if ( stepLength > 8 )    // reach MAX stepLength
						break;
					else
					{
						System.out.println( "Change stepLength to: " + stepLength );
						continue;
					}
				}
				else
				{
					resetStepLength = usingResetSteplength;
				}

				//result.showResult_ID();
				System.out.println( "Total Length: " + result2.totalLength() + "        Total Points: " +
result2.totalPoints() );
				result2.resetStatus();
			} while ( true );

			// Final Result
			System.out.println( "Final Result of Second Part:");
			result2.showResult_ID();

			// combine first part and second part
			ArrayList overLoad2Routes = new ArrayList();
			result1.makeRelationShip( result2, overLoad2Routes );
			if ( overLoad2Routes.size() != 0 )
			{
				System.out.println( "There is(are) OverLoad pair(s) of routes! " );
				for ( int i=0; i<overLoad2Routes.size(); i+=2 )
					System.out.println(      overLoad2Routes.get(i)      +      "      and      "      +
overLoad2Routes.get(i+1) );

				// Result before dealing with overLoadRoutes
				System.out.println();
				System.out.println();
				System.out.println();
				System.out.println();
				System.out.println();
				System.out.println( "Result of First Part before dealing with overLoadRoutes:");
				result1.showResult_ID();
				System.out.println();
				System.out.println( "Result of Second Part before dealing with overLoadRoutes:");
				result2.showResult_ID();
			}
```

```
        while ( overLoad2Routes.size() > 0 )
        {
            modifyOverLoadRoutes( depot, result1, result2, overLoad2Routes );
            overLoad2Routes = new ArrayList();
            result1.makeRelationShip( result2, overLoad2Routes );
        }

        // Final Result
        System.out.println();
        System.out.println( "Final Result of First Part:");
        result1.showResult_ID();
        System.out.println();
        System.out.println( "Final Result of Second Part:");
        result2.showResult_ID();

        currentTime = new Date();
        milSecond = currentTime.getTime();
        System.out.println( "Time End of Algorithm Calculation:" + milSecond );
        System.out.println();
    }


    private static void readFile( PointsUnion firstPartPoints, PointsUnion secondPartPoints, int
selectedday )
    {
        try
        {
            //FileReader myFile = new FileReader( "D:\\BlueJ\\MyProject\\data.csv" );
            FileReader myFile = new FileReader( "data.csv" );

            if( myFile.ready() )
            {
                System.out.println( "File Opened." );

                BufferedReader bfr = new BufferedReader( myFile );
                ArrayList dataAll = new ArrayList();
                String line = bfr.readLine();
                while ( line != null )
                {
                    System.out.println("READ>" + line + "<");
                    StringTokenizer tokenizer=new StringTokenizer(line,",");
                    Vector dataLine = new Vector();
                    for ( int i=0; i<7; i++ )
                    {
                        if ( tokenizer.hasMoreTokens() )
                        {
                            dataLine.addElement( tokenizer.nextToken() );
                        }
                        else
                        {
                            myFile.close();
                            System.out.println( "Data in the file error at line <ERROR>" +
line );
                            return;
                        }
                    }
                    if ( tokenizer.hasMoreTokens() )
                    {
                        myFile.close();
                        System.out.println( "Data in the file error at line <ERROR>" + line );
                        return;
                    }
                    else
                    {
                        dataAll.add(dataLine);
                    }
                    line = bfr.readLine();
                }

                // Display dataAll
                System.out.println( "dataAll:" );
                for ( int i=0; i<dataAll.size(); i++ )
                {
                    Vector oneline = (Vector)dataAll.get(i);
                    if ( string2int( (String)oneline.get(5) ) == selectedday )
                    {
                        for ( int j=0; j<oneline.size(); j++ )
                        {
                            line = (String)oneline.get(j);
                            if ( isAllNum(line) )
                            {
                                System.out.print( string2int(line) + " " );
                            }
                            else
                            {
                                System.out.println( "Data in the file error at line <ERROR>"
+ i );

                                return;
                            }
                        }
                        System.out.println();
                        line = (String)oneline.get(1);
                        Point p1 = new Point( string2int( (String)oneline.get(0) ),
string2int( (String)oneline.get(1) ) , string2int( (String)oneline.get(2) ),
string2int( (String)oneline.get(6) ) );
                        Point p2 = new Point( string2int( (String)oneline.get(0) ),
string2int( (String)oneline.get(3) ) , string2int( (String)oneline.get(4) ),
string2int( (String)oneline.get(6) ) );
                        firstPartPoints.appendPoint( p1 );
                        secondPartPoints.appendPoint( p2 );
                    }
                }
            }
            else
            {
                System.out.println( "Directory or File dose not exist." );
            }
        }
        catch ( Exception e )
```

```
		{
			System.out.println( "Cannot Find File" );
		}
	}

	private static boolean isAllNum( String s )
	{
		for ( int i=0; i<s.length(); i++ )
		{
			if ( ( s.charAt(i)<'0' ) || ( s.charAt(i)>'9') )
				return false;
		}
		return true;
	}

	private static int string2int( String s )
	{
		int out = 0;
		for ( int i=0; i<s.length(); i++ )
		{
			out *= 10;
			out += s.charAt(i) - 48;
		}
		return out;
	}

	private static void modifyOverLoadRoutes( Point depot, Result result1, Result result2,
ArrayList overLoad2Routes )
	{
		int bestTime = ( 16 - ( 40 + 2*depot.getGoods() ) / 60 ) / 2;
		double speed = 60000;

		ArrayList hasBeenSplittedInR1 = new ArrayList();
		ArrayList hasBeenSplittedInR2 = new ArrayList();

		int selectPart = 0;
		for ( int i=0; i<overLoad2Routes.size(); i+=2 )
		{
			String tmp = overLoad2Routes.get(i).toString();
			int pos1 = string2int( tmp );
			tmp = overLoad2Routes.get(i+1).toString();
			int pos2 = string2int( tmp );

			if ( hasBeenSplittedInR1.contains(pos1) || hasBeenSplittedInR2.contains(pos2) )
				continue;

			double length1 = ( (Route)( result1.routes.get( ( pos1 ) ) ) ).getLength();
			double length2 = ( (Route)( result2.routes.get( ( pos2 ) ) ) ).getLength();
			if ( length1 >= length2 )
				selectPart = 1;
			else
				selectPart = 2;

			Route splittedRoute;
			int splittedNumber = 0;
```

```
double averageLength = 0.0d;
if ( selectPart == 1 )
{
    splittedRoute = (Route)( result1.routes.get( ( pos1 ) ) );
    splittedNumber = (int)((length1/speed)/(double)bestTime) + 1;
    if ( splittedNumber == 1 )
        splittedNumber = 2;
    averageLength = length1/splittedNumber;

    hasBeenSplittedInR1.add( pos1 );
}
else
{
    splittedRoute = (Route)( result2.routes.get( ( pos2 ) ) );
    splittedNumber = (int)((length2/speed)/(double)bestTime) + 1;
    if ( splittedNumber == 1 )
        splittedNumber = 2;
    averageLength = length2/splittedNumber;

    hasBeenSplittedInR2.add( pos2 );
}

ArrayList newRoutes = new ArrayList();
for ( int j=0; j<splittedNumber; j++ )
{
    // Find the farest point
    int farestPoint = -1;
    double farestDistance = 0.0d;
    for ( int k=0; k<splittedRoute.getSize(); k++ )
    {
        if ( farestPoint == -1 )
        {
            farestPoint = k;
            farestDistance = splittedRoute.getPoint(k).getDistance(depot);
        }
        else
        {
            double tmplength = splittedRoute.getPoint(k).getDistance(depot);
            if ( tmplength > farestDistance )
            {
                farestDistance = tmplength;
                farestPoint = k;
            }
        }
    }

    // Create a new route and add the farestPoint
    Route oneNewRoute = new Route( depot );
    oneNewRoute.appendPoint( splittedRoute.getPoint( farestPoint ) );

    // Save and Delete farset point
    Point lastPoint = splittedRoute.getPoint( farestPoint );
    splittedRoute.removePoint( farestPoint );

    while ( true )
```

```
                    {
                        // Add the remaining point which is nearest with lastPoint and routeLength
possible
                        int shortestPoint = -1;
                        double shortestDistance = 0.0d;
                        for ( int k=0; k<splittedRoute.getSize(); k++ )
                        {
                            Route testRoute = new Route( depot );;
                            testRoute.cloneRoute( oneNewRoute );
                            testRoute.appendPoint( splittedRoute.getPoint( k ) );
                            if ( testRoute.getLength() > averageLength )
                                continue;
                            else
                            {
                                if ( shortestPoint == -1 )
                                {
                                    shortestDistance                              =
splittedRoute.getPoint( k ).getDistance( lastPoint );
                                    shortestPoint = k;
                                }
                                else
                                {
                                    double              tmpDis                    =
splittedRoute.getPoint( k ).getDistance( lastPoint );
                                    if ( tmpDis < shortestDistance )
                                    {
                                        shortestDistance = tmpDis;
                                        shortestPoint = k;
                                    }
                                }
                            }
                        }
                        if ( shortestPoint == -1 )
                            break;
                        else
                        {
                            // add shortestPoint
                            oneNewRoute.appendPoint( splittedRoute.getPoint( shortestPoint ) );
                            // save and delete shortestPoint
                            lastPoint = splittedRoute.getPoint( shortestPoint );
                            splittedRoute.removePoint( shortestPoint );
                        }
                    }

                    // Add the new route into newRoutes
                    newRoutes.add( oneNewRoute );
                }

                // add remaining point(s)
                for ( int j=0; j<splittedRoute.getSize(); j++ )
                {
                    int selectRoute = -1;
                    double shortestInc = 0.0d;
                    for ( int k=0; k<newRoutes.size(); k++ )
                    {
                        if ( selectRoute == -1 )
                        {
                            selectRoute = 0;
                            Route testRoute = new Route( depot );;
                            testRoute.cloneRoute( (Route)newRoutes.get(0) );
                            testRoute.appendPoint( splittedRoute.getPoint( j ) );
                            shortestInc          =          testRoute.getLength()          -
((Route)newRoutes.get(0)).getLength();
                        }
                        else
                        {
                            Route testRoute = new Route( depot );;
                            testRoute.cloneRoute( (Route)newRoutes.get(k) );
                            testRoute.appendPoint( splittedRoute.getPoint( j ) );
                            double      tmpInc          =       testRoute.getLength()       -
((Route)newRoutes.get(k)).getLength();
                            if ( tmpInc < shortestInc )
                            {
                                shortestInc = tmpInc;
                                selectRoute = k;
                            }
                        }
                    }

((Route)newRoutes.get(selectRoute)).appendPoint( splittedRoute.getPoint( j ) );
                }

                // Save the newRoutes, attention we should replace the splittedRoute using the first
route in newRoutes
                if ( selectPart == 1 )
                {
                    result1.routes.set( pos1, newRoutes.get(0) );
                    for ( int j=1; j<newRoutes.size(); j++ )
                        result1.routes.add( newRoutes.get(j) );
                }
                else
                {
                    result2.routes.set( pos2, newRoutes.get(0) );
                    for ( int j=1; j<newRoutes.size(); j++ )
                        result2.routes.add( newRoutes.get(j) );
                }
            }
        }
    }
```

****

Point

```
/**
  * Demonstrate the charactor of the points and the method of getting distance
  * bewteen two points.
  * @version (28_08_2005)
  */
public class Point
{
    private int id;
private int x;
private int y;
private int goods;
private double tan;


/**
 * Constructor for objects of class Point
 */
public Point( int inputID, int inputX, int inputY, int inputGoods )
{
	id = inputID;
	x = inputX;
	y = inputY;
	goods = inputGoods;
	tan = 0;
}

/**
 * return ID
 */
public int getID()
{
	return id;
}


/**
 * return x
 */
public int getX()
{
	return x;
}


/**
 * return y
 */
public int getY()
{
	return y;
}


/**
 * return goods
 */
public int getGoods()
{
	return goods;
}


/**
```

```
         * return tan
         */
        public double getTan()
        {
            return tan;
        }


        /**
         * Calculate the distance between two Points
         *
         * @Parameter1: Another Point p
         * return: distance between these 2 points
         */
        public float getDistance( Point p )
        {
            double squareDis = ( (double)( x - p.getX() ) ) * ( (double)( x - p.getX() ) ) + ( (double)( y -
p.getY() ) ) * ( (double)( y - p.getY() ) );
            return (float) Math.sqrt( squareDis );
        }


        /**
         * Calculate the tan() value between current Point and Depot
         *
         * @Parameter1: Depot point
         * return: tan() value between these 2 points
         */
        public void calculateTan( Point depot )
        {
            double disX = (double)( x - depot.getX() );
            if ( disX == 0.0d )
            {
                if ( y >= depot.getY() )
                    tan = 9.999999999E9d;
                else
                    tan = -9.999999999E9d;
            }
            double disY = (double)( y - depot.getY() );
            tan = (double)( disY/disX );
        }


        /**
         * Clone a point
         *
         * @Parameter1: point
         * return:
         */
        public void clonePoint( Point p )
        {
            id = p.getID();
            x = p.getX();
            y = p.getY();
            goods = p.getGoods();
```

```
		tan = p.getTan();
	}


	/**
	  * Clone a point
	  *
	  * @Parameter1: point
	  * return:
	  */
	public boolean equals( Point p )
	{
		if ( id != p.getID() )
			return false;
		if ( x != p.getX() )
			return false;
		if ( y != p.getY() )
			return false;
		if ( goods != p.getGoods() )
			return false;

		return true;
	}
}
```

****

PointUnion

```
import java.util.ArrayList;
import java.util.Random;

/**
  * Demonstrate several actions of points in the algorithm process.
  * @version (28_08_2005)
  */
public class PointsUnion
{
	private ArrayList union;


	/**
	  * Constructor for objects of class PointsUnion
	  */
	public PointsUnion()
	{
		union = new ArrayList();
	}


	/**
	  * Appends a point to the end of this union.
	  */
	public boolean appendPoint( Point p )
	{
		return( union.add( p ) );
	}
```

```
/**
 * Deletes a point at specified position.
 */
public void removePoint( int index )
{
	union.remove(index);
}


/**
 * Clear all points
 */
public void clearAllPoints( )
{
	union.clear();
}


/**
 * Search for a specified point
 * Return: index of that point if found, -1 if not found
 */
public int searchPoint( Point p )
{
	return( union.indexOf( p ) );
}

/**
 * return the number of pointUnion
 */
public int getSize( )
{
	return( union.size() );
}


/**
 * Return the point at the specified index number
 */
public Point getPoint( int index )
{
	return( (Point)(union.get(index) ) );
}


/**
 * Return the distance between two points at the specified index numbers
 */
public float getDistance( int point1, int point2 )
{
	return( getPoint(point1).getDistance( getPoint(point2) ) );
}
```

```
/**
 * Return the distance between two points at the specified index numbers
 */
public void sortPointsByTan( )
{
	Point tmpPoint = new Point( 0, 0, 0, 0 );
	// sorting
	for ( int i=1; i<getSize(); i++ )
	{
		for ( int j=0; j<getSize()-i; j++ )
		{
			if ( getPoint(j).getTan() > getPoint(j+1).getTan() )
			{
				tmpPoint.clonePoint( getPoint(j) );
				getPoint(j).clonePoint(getPoint(j+1));
				getPoint(j+1).clonePoint(tmpPoint);
			}
		}
	}
}


/**
 * Return the distance between two points at the specified index numbers
 */
public double calculateAllGoods( )
{
	double total = 0.0d;
	for ( int i=0; i<union.size(); i++ )
		total += getPoint(i).getGoods();
	return total;
}


/**
 * Return the distance between two points at the specified index numbers
 */
public int findPoint( Point p )
{
	for ( int i=0; i<union.size(); i++ )
	{
		if ( getPoint(i).equals(p) )
			return i;
	}

	return -1;
}


/**
 * Using Distance Algorithm to get the routes
 */
public void generateRoutesByDistance( Point depot, ArrayList routes )
{
	Point startPoint = depot;
```

```
        Route newRoute = new Route( depot ); //Every new route will start from the depot
        do
        {
            int selectedNode = -1; // no selected point at first
            float shortestDistance = 0.0f;


            /**
             * Loop for all of the points in unionM to find out the nearest points
             * to depot.
             */
            for ( int j=0; j<getSize(); j++ )
            {
                if ( !newRoute.overLoad( getPoint(j) ) )
                {
                    if ( selectedNode == -1 )
                    {
                        selectedNode = j;
                        shortestDistance = startPoint.getDistance( getPoint(j) );
                    }
                    else
                    {
                        float tempDistance = startPoint.getDistance( getPoint(j) );
                        if ( tempDistance == shortestDistance )     // The 2 points have same
distance. Select one which contains more goods.
                        {
                            if ( getPoint(j).getGoods() > getPoint(selectedNode).getGoods() )
                            {
                                selectedNode = j;
                                // As 2 distances are same, need not change
shortestDistance.
                            }
                        }

                        if ( tempDistance < shortestDistance )
                        {
                            selectedNode = j;
                            shortestDistance = tempDistance;
                        }
                    }
                }
            }

            if ( selectedNode != -1 ) //some satisfied point has been searched
            {
                if ( newRoute.appendPoint( getPoint(selectedNode) ) )
                {
                    startPoint = getPoint(selectedNode); // reset startpoint
                    removePoint( selectedNode ); // delete the selected point from unionM
                }
                else
                {
                    System.out.println("Adding method of ArrayList failed!");
                    return;
                }
```

```
                        }
                        else    // no new point, this current route is finished
                        {
                                // there are no points in the newroute
                                if ( newRoute.getSize() == 0 )
                                {
                                        if ( getSize() != 0 )
                                        {
                                                System.out.println("There  are/is  point(s)  left  unselected.  Algorithm
Failed!");
                                        }
                                        break;
                                }

                                // if there are points in the newroute, save this route
                                if ( routes.add( newRoute ) )
                                {
                                        startPoint = depot; // reset startpoint
                                        newRoute = new Route( depot );
                                }
                                else
                                {
                                        System.out.println("add method of ArrayList failed!!!");
                                        return;
                                }
                        }
                }while ( true );
        }

        void reducePointsUnionSize( int targetSize )
        {
                double random;
                while ( getSize() > targetSize )
                {
                        //random    = Math.random();
                        //removePoint( (int)( getSize() * random ) );
                        removePoint(getSize() - 1);
                }
        }
}
```

****

Route

```
import java.util.ArrayList;

/**
 * Demonstrate that the total demand of the route is the only contraint that
 * every route need to consider when define such route is feasible or not
 * @version (28_08_2005)
 */
public class Route extends PointsUnion
{
        private Point depot;
        public int optimizedTimes;
        public int removedPoints;
        public int addedPoints;
        public boolean selectedAsRoute1;
        public ArrayList relatedRoutes;

        /**
         * Constructor for objects of class Route
         */
        public Route( Point p )
        {
            super();
            depot = p;
            optimizedTimes = 0;
            removedPoints = 0;
            addedPoints = 0;
            selectedAsRoute1 = true;
            relatedRoutes = new ArrayList();
        }

        public boolean noChange( )
        {
            if ( ( removedPoints == 0 ) && ( addedPoints == 0 ) )
                return true;
            else
                return false;
        }

        public void resetStatus( )
        {
            optimizedTimes = 0;
            removedPoints = 0;
            addedPoints = 0;
            selectedAsRoute1 = true;
        }

        /**
         * method for checking whether the total quantity of the goods exceeds the
         * demand of each route or not
         */
        public boolean overLoad( Point p )
        {
            int totalweight = 0;
            for ( int i=0; i<getSize(); i++ )
                totalweight += getPoint(i).getGoods();
            totalweight += p.getGoods();
            if ( totalweight <= depot.getGoods() )
                return false;
            else
                return true;
        }


        /**
         * method for checking whether the total quantity of the goods exceeds the
         * demand of each route or not
         */
        	public void resortByDistance( )
        	{
        		Point startPoint = depot;
        		Route newRoute = new Route( depot ); //Every new route will start from the depot
        		do
        		{
        			int selectedNode = -1; // no selected point at first
        			float shortestDistance = 0.0f;

        			/**
        			 * Loop for all of the points in unionM to find out the nearest points
        			 * to depot.
        			 */
        			for ( int j=0; j<getSize(); j++ )
        			{
        				if ( selectedNode == -1 )
        				{
        					selectedNode = j;
        					shortestDistance = startPoint.getDistance( getPoint(j) );
        				}
        				else
        				{
        					float tempDistance = startPoint.getDistance( getPoint(j) );
        					if ( tempDistance == shortestDistance )	// The 2 points have same
        distance. Select one which contains more goods.
        					{
        						if ( getPoint(j).getGoods() > getPoint(selectedNode).getGoods() )
        						{
        							selectedNode = j;
        							// As 2 distances are same, need not change	shortestDistance.
        						}
        					}

        					if ( tempDistance < shortestDistance )
        					{
        						selectedNode = j;
        						shortestDistance = tempDistance;
        					}
        				}
        			}

        			if ( selectedNode != -1 ) //some satisfied point has been searched
        			{
        				if ( newRoute.appendPoint( getPoint(selectedNode) ) )
        				{
        					startPoint = getPoint(selectedNode); // reset startpoint
        					removePoint( selectedNode ); // delete the selected point from unionM
        				}
        				else
        				{
        					System.out.println("Adding method of ArrayList failed!");
        					return;
        				}
        			}
        			else	// no new point, this current route is finished
        			{
        				// there are no points in the newroute
        				if ( newRoute.getSize() == 0 )
        				{
        					if ( getSize() != 0 )
        					{
        						System.out.println("There are/is point(s) left unselected. Algorithm
        Failed!");
        					}
        					break;
        				}

        				// if there are points in the newroute, save this route
        				for ( int j=0; j<newRoute.getSize(); j++ )
        				{
        					if ( ! appendPoint( newRoute.getPoint(j) ) )
        					{
        						System.out.println("Adding method of ArrayList failed!");
        						return;
        					}
        				}
        				break;
        			}
        		}while ( true );
        	}


        	/**
        	 * exchange Points, add insertpoint, delete a point
        	 * return : integer>0: as can exchage successful, return the ID of the point need to exchange
        in the route
        	 *		-1: cannot perform exchage as the insertpoint is too large for it's goods.
        	 */
        	public int findExchangePoint( Point insertpoint )
        	{
        		int totalweight = 0;
        		for ( int i=0; i<getSize(); i++ )
        			totalweight += getPoint(i).getGoods();

        		int selectedpoint = -1;
        		Point deletePoint = new Point( 0, 0, 0, 0 );
        		for ( int i=0; i<getSize(); i++ )
        		{
        			if ( ( totalweight - getPoint(i).getGoods() + insertpoint.getGoods() ) <=
        depot.getGoods() )
        			{
        				if ( selectedpoint == -1 )
        				{
        					selectedpoint = i;
        					deletePoint = getPoint(i);
        				}
        				if ( getPoint(i).getGoods() < deletePoint.getGoods() )
        				{
        					selectedpoint = i;
        					deletePoint = getPoint(i);
```

```
                    }
                }
            }
            return selectedpoint;
        }


        public void cloneRoute( Route r )
        {
            optimizedTimes = r.optimizedTimes;
            removedPoints = r.removedPoints;
            addedPoints = r.addedPoints;
            selectedAsRoute1 = r.selectedAsRoute1;

            for ( int i=getSize(); i>0; i-- )
                removePoint(i-1);
            for ( int i=0; i<r.getSize(); i++ )
                appendPoint( r.getPoint(i) );
        }


        public double getLength()
        {
            if ( getSize() == 0 )
                return 0.0d;

            double routeLength = 0.0d;
            for ( int j=0; j<getSize(); j++ )
            {
                if ( j==0 )    // The first point connected with depot
                    routeLength += depot.getDistance( getPoint(0) );
                else
                    routeLength += getDistance( j, j-1 );
            }
            routeLength += depot.getDistance( getPoint(getSize()-1) ); // finish the total length of the
route
            return routeLength;
        }


        public int getTotalGoods()
        {
            int totalweight = 0;
            for ( int i=0; i<getSize(); i++ )
                totalweight += getPoint(i).getGoods();
            return totalweight;
        }
    }
```

****

Result

```
import java.util.ArrayList;

/**
 * @version (28_08_2005)
```

```
  */
public class Result
{
	private Point depot;
	private Point zeroPoint;
	private Point movePoint;
	public ArrayList routes;

	public Result( Point resultDepot, ArrayList resultRoutes)
	{
		depot = resultDepot;
		routes = resultRoutes;
		zeroPoint = new Point( 0, 0, 0, 0 );
		movePoint = new Point( 0, 0, 0, 0 );
	}


	public boolean noChange( )
	{
		for ( int i=0; i<routes.size(); i++ )
		{
			if ( !((Route)routes.get(i)).noChange() )
				return false;
		}
		return true;
	}


	public void resetStatus( )
	{
		for ( int i=0; i<routes.size(); i++ )
		    ((Route)routes.get(i)).resetStatus();
		movePoint = zeroPoint;
	}


	public void showResult_ID( )
	{
		// Show Result again using ID instead of X,Y coordinates
		System.out.println();
		System.out.println();
		System.out.println( "The Result shows in ID:" );
		double totalLength = 0.0d;
		int totalPoints = 0;
		for ( int i=0; i<routes.size(); i++ )
		{
			float routeLength = 0.0f;
			System.out.print( "Route " + i + ":");
			Route newRoute = (Route)routes.get(i);
			for ( int j=0; j<newRoute.getSize(); j++ )
			{
				System.out.print( " " + newRoute.getPoint(j).getID() + "," );
				totalPoints++;
				if ( j==0 )    // The first point connected with depot
					routeLength += depot.getDistance( newRoute.getPoint(0) );
```

```
                        else
                              routeLength += newRoute.getDistance( j, j-1 );
                  }
                  routeLength += depot.getDistance( newRoute.getPoint(newRoute.getSize()-1) ); // finish
the total length of the route
                  System.out.println( "        Length:" + routeLength );
                  totalLength += routeLength;
            }
            System.out.println();
            System.out.println( "The Total Length of " + routes.size() + " line(s): " + totalLength );
            System.out.println( "The Total Number of the Points: " + totalPoints );
      }


      public void removeNullRoutes( )
      {
            for ( int i=0; i<routes.size(); i++ )
            {
                  Route newRoute = (Route)routes.get(i);
                  if ( newRoute.getSize() == 0 )
                        routes.remove(i);
            }
      }


      public double totalLength ( )
      {
            double tl = 0.0d;
            for ( int i=0; i<routes.size(); i++ )
                   tl += ((Route)routes.get(i)).getLength();
            return tl;
      }

      public int totalPoints ( )
      {
            int tp = 0;
            for ( int i=0; i<routes.size(); i++ )
                   tp += ((Route)routes.get(i)).getSize();
            return tp;
      }


      public void optimize_Method_1( int stepLength, boolean anticlockwise, boolean display )
      {
            if ( routes.size() <= 1 )
            {
                  return;
            }

            boolean result = true;
            int selectedRoute1, selectedRoute2;
            Route optimizeRoute1 = new Route( depot );
            Route optimizeRoute2 = new Route( depot );
            if ( anticlockwise )
            {
```

```
            for ( selectedRoute1=0; selectedRoute1<routes.size(); selectedRoute1++ )
            {
                optimizeRoute1 = (Route)routes.get(selectedRoute1);

                selectedRoute2 = selectedRoute1;
                selectedRoute2 += stepLength;
                if ( selectedRoute2 >= routes.size() )
                    selectedRoute2 -= routes.size();
                optimizeRoute2 = (Route)routes.get(selectedRoute2);

                if (display)    System.out.print( "Try Optimizing: " + selectedRoute1 + "      " +
selectedRoute2 + ".    " );
                result = Optimize_Function_1( optimizeRoute1, optimizeRoute2, display );
                if ( result )
                {
                    if (display)
                        System.out.println( "Success. Point " + movePoint.getID() + " moved." );
                }
                else
                    if (display)    System.out.println( "Failed." );
            }
        }
        else       // clockwise
        {
            for ( selectedRoute1=routes.size()-1; selectedRoute1>=0; selectedRoute1-- )
            {
                optimizeRoute1 = (Route)routes.get(selectedRoute1);

                selectedRoute2 = selectedRoute1;
                selectedRoute2 -= stepLength;
                if ( selectedRoute2 < 0 )
                    selectedRoute2 += routes.size();
                optimizeRoute2 = (Route)routes.get(selectedRoute2);

                if (display)    System.out.print( "Try Optimizing: " + selectedRoute1 + "      " +
selectedRoute2 + ".    " );
                result = Optimize_Function_1( optimizeRoute1, optimizeRoute2, display );
                if ( result )
                {
                    if (display)
                        System.out.println( "Success. Point " + movePoint.getID() + " moved." );
                }
                else
                    if (display)    System.out.println( "Failed." );
            }
        }
    }




    public void sortRoutesByTan( )
    {
        PointsUnion leftOrder = new PointsUnion();
        PointsUnion rightOrder = new PointsUnion();
```

```
		for ( int i=0; i<routes.size(); i++ )
		{
			Route thisRoute = (Route)routes.get(i);
			int selectedpoint = thisRoute.getSize()-1; // Last point
			if ( (thisRoute.getPoint(0).getX()-depot.getX()) *
(thisRoute.getPoint(selectedpoint).getX()-depot.getX()) >= 0 )
			{
				if ( thisRoute.getPoint(0).getTan() > thisRoute.getPoint(selectedpoint).getTan() )
					selectedpoint = 0;
			}
			else
			{
				if ( thisRoute.getPoint(0).getTan() < thisRoute.getPoint(selectedpoint).getTan() )
					selectedpoint = 0;
			}

			Point newPoint = new Point( 0, 0, 0, 0 );
			newPoint.clonePoint( thisRoute.getPoint(selectedpoint) );

			if ( newPoint.getX() > depot.getX() )	// right side
				rightOrder.appendPoint( newPoint );
			else	// left side
				leftOrder.appendPoint( newPoint );
		}

		leftOrder.sortPointsByTan();
		rightOrder.sortPointsByTan();

		// rightOrder + leftOrder = orderPoints;
		PointsUnion orderPoints = new PointsUnion();
		for ( int i=0; i<rightOrder.getSize(); i++ )
			orderPoints.appendPoint( rightOrder.getPoint(i) );
		for ( int i=0; i<leftOrder.getSize(); i++ )
			orderPoints.appendPoint( leftOrder.getPoint(i) );

		// create a copy for routes
		ArrayList temp = new ArrayList();
		for ( int i=0; i<routes.size(); i++ )
			temp.add( routes.get(i) );

		// clear routes to wait for routes inserting in order
		routes.clear();

		// move each route into routes in the order
		if ( orderPoints.getSize() != temp.size() )
		{
			System.out.println( "ERROR happens in sortRoutesByTan !!!" );
			System.out.println( "ERROR happens in sortRoutesByTan !!!" );
			System.out.println( "ERROR happens in sortRoutesByTan !!!" );
			return;
		}
		for ( int i=0; i<orderPoints.getSize(); i++ )
		{
			Point orderPoint = orderPoints.getPoint(i);
```

```
            int j;
            for ( j=0; j<temp.size(); j++ )
            {
                // check whether orderPoint in route temp[j]
                if ( ((Route)temp.get(j)).findPoint(orderPoint) != -1 )
                {
                    routes.add( (Route)temp.get(j) );
                    break;
                }
            }

            if ( j == temp.size() )
            {
                System.out.println( "ERROR happens in sortRoutesByTan !!!" );
                System.out.println( "ERROR happens in sortRoutesByTan !!!" );
                System.out.println( "ERROR happens in sortRoutesByTan !!!" );
                return;
            }
        }

        // check routes at last
        if ( orderPoints.getSize() != routes.size() )
        {
            System.out.println( "ERROR happens in sortRoutesByTan !!!" );
            System.out.println( "ERROR happens in sortRoutesByTan !!!" );
            System.out.println( "ERROR happens in sortRoutesByTan !!!" );
            return;
        }
    }



    public void makeRelationShip( Result target, ArrayList overLoadRelationRoutes )
    {
        for ( int i=0; i<routes.size(); i++ )
        {
            Route r1 = (Route)routes.get(i);
            for ( int j=0; j< target.routes.size(); j++ )
            {
                Route r2 = (Route)target.routes.get(j);
                for ( int id1=0; id1<r1.getSize(); id1++ )
                {
                    for ( int id2=0; id2<r2.getSize(); id2++ )
                    {
                        if ( r1.getPoint(id1).getID() == r2.getPoint(id2).getID() )
                        {
                            double routeTime = ( ( r1.getLength() + r2.getLength() ) ) / 60000.0d;

                            if ( ( routeTime + ( 40 + ( 2*r1.getTotalGoods() + 2*r2.getTotalGoods() ) )
/ 60.0d ) < 16.0d )
                            {
                                if ( !r1.relatedRoutes.contains(j) )
                                    r1.relatedRoutes.add( j );
                                if ( !r2.relatedRoutes.contains(i) )
                                    r2.relatedRoutes.add( i );
                            }
                            else
                            {
                                overLoadRelationRoutes.add( i );
                                overLoadRelationRoutes.add( j );
                            }
                        }
                    }
                }
            }
        }
    }



    private int selectMinimumGoods( )
    {
        // Select the minimum demand route which has not been optimized
        double minimumGoods = 0.0d;
        int selectedRoute = -1;
        for ( int i=0; i<routes.size(); i++ )
        {
            Route newRoute = (Route)routes.get(i);
            // Check whether has been optimized
            if ( ( newRoute.addedPoints > 0 ) || ( !newRoute.selectedAsRoute1 ) )
            {
                continue;
            }
            else
            {
                // Is it the first one which has not been optimized
                if ( selectedRoute == -1 )
                {
                    selectedRoute = i;
                    minimumGoods = newRoute.calculateAllGoods();
                }
                else       // Compare and select minimum
                {
                    double temp = newRoute.calculateAllGoods();
                    if ( temp < minimumGoods )
                    {
                        minimumGoods = temp;
                        selectedRoute = i;
                    }
                }
            }
        }

        return selectedRoute;
    }


    private boolean Optimize_Function_1( Route route1, Route route2, boolean display )
    {
        int selectedPoint = -1;
```

```
            double largestDecrease = 0.0d;
            // Select all of the point from route1
            for ( int i=0; i<route1.getSize(); i++ )
            {
                  if ( ! route2.overLoad( route1.getPoint( i ) ) )
                  {
                        double decrease = decreaseByMoveOnePointBetween2Routes( route1, route2, i,
display );
                        if ( decrease > 0.0d )
                        {
                              if ( selectedPoint == -1 )
                              {
                                    selectedPoint = i;
                                    largestDecrease = decrease;
                              }
                              else
                              {
                                    if ( decrease == largestDecrease )
                                    {
                                          if ( route1.getPoint(i).getGoods() >
route1.getPoint(selectedPoint).getGoods() )
                                                selectedPoint = i;
                                    }
                                    else if ( decrease > largestDecrease)
                                    {
                                          selectedPoint = i;
                                          largestDecrease = decrease;
                                    }
                              }
                        }
                  }
            }

            if ( selectedPoint == -1 )
            {
                  movePoint = zeroPoint;
                  return false;
            }
            else
            {
                  movePoint = route1.getPoint( selectedPoint );

                  route1.removePoint( selectedPoint );
                  route1.removedPoints++;
                  route1.resortByDistance();

                  route2.appendPoint( movePoint );
                  route2.addedPoints++;
                  route2.resortByDistance();

                  return true;
            }
      }
```

```
	private int findRouteBetweenTwoPoints( Point p1, Point p2 )
	{
		double smalltan = p1.getTan();
		double largetan = p2.getTan();
		if ( smalltan>largetan) // exchage them
		{
			double temp = smalltan;
			smalltan = largetan;
			largetan = temp;
		}

		for ( int i=0; i<routes.size(); i++ )
		{
			Route checkRoute = (Route)routes.get(i);
			boolean allRouteIn = true;
			for ( int j=0; j<checkRoute.getSize(); j++ )
			{
				double checkTan = checkRoute.getPoint(j).getTan();
				if ( ( checkTan < smalltan ) || (checkTan > largetan ) )
				{
					allRouteIn = false;
					break;
				}
			}
			if ( allRouteIn )
				return i;
		}

		return -1;
	}


	private double decreaseByMoveOnePointBetween2Routes( Route route1, Route route2, int pointID ,
boolean display )
	{
		if ( route2.overLoad( route1.getPoint( pointID ) ) )
			return -9.99999999999E9;

		Route testRoute = new Route( depot );
		testRoute.cloneRoute( route1 );
		testRoute.removePoint( pointID );
		testRoute.resortByDistance();
		double decreasedLength = route1.getLength() - testRoute.getLength();
		//if ( decreasedLength < 0.0d )
			//System.out.println( "ERROR ERROR ERROR" );

		testRoute.cloneRoute( route2 );
		testRoute.appendPoint( route1.getPoint( pointID ) );
		testRoute.resortByDistance();
		double increasedLength = testRoute.getLength() - route2.getLength();
		//if ( increasedLength < 0.0d )
			//System.out.println( "ERROR ERROR ERROR" );

		return ( decreasedLength - increasedLength );
	}
	private void addOnePointInRoutes( Point p , boolean display)
	{
		double smallestDis = depot.getDistance(p) * 2 ;
		int selectedRoute = -1;

		for ( int i=0; i<routes.size(); i++ )
		{
			if ( ((Route)routes.get(i)).overLoad(p) )
				continue;
			else
			{
				Route testRoute = new Route( depot );
				testRoute.cloneRoute( (Route)routes.get(i) );
				testRoute.appendPoint(p);
				testRoute.resortByDistance();
				double increasedLength = testRoute.getLength() - ((Route)routes.get(i)).getLength();
				if ( increasedLength < 0.0d )
				{
					//System.out.println( "ERROR ERROR ERROR" );
				}
				if ( increasedLength<smallestDis )
				{
					selectedRoute = i;
					smallestDis = increasedLength;
				}
			}
		}

		if ( selectedRoute == -1 )
		{
			Route newRoute = new Route( depot );
			newRoute.appendPoint(p);
			routes.add(newRoute);
			if (display)	System.out.println( "Add point " + p.getID() + " in a sing-point new Route " +
(routes.size()-1) );
		}
		else
		{
			((Route)routes.get(selectedRoute)).appendPoint(p);
			((Route)routes.get(selectedRoute)).resortByDistance();
			if (display)	System.out.println( "Add point " + p.getID() + " in Route " + selectedRoute );
		}
	}
}
```