รายการอ้างอิง

ภาษาไทย

ปารเมศ ชุติมา. 2546. **เทคนิคการจัดตารางการดำเนินงาน.** กรุงเทพมหานคร: สำนักพิมพ์แห่งจุฬาลงกรณ์มหาวิทยาลัย.

พงศ์พัฒน์ โตตระกูล. 2546. **วิธีค้นหาคำตอบแบบฮิวริสติกสำหรับปัญหาการจัดเส้นทางการขนส่งเวชภัณฑ์ในระบบการกระจายเวชภัณฑ์ของโรงพยาบาล.** วิทยานิพนธ์ปริญญามหาบัณฑิต, ภาควิชาวิศวกรรมอุตสาหการ บัณฑิตวิทยาลัย จุฬาลงกรณ์มหาวิทยาลัย.

วนิดา เหมะกุล. 2535. **คณิตศาสตร์ดิสครีต.** กรุงเทพมหานคร: สำนักพิมพ์ซีเอ็ดยูเคชั่น.

ภาษาอังกฤษ

Blum, C., and Roli, A. 2003. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. **ACM Computing Surveys** 35: 268-308.

Clarke, G., and Wright, J. W. 1964. Scheduling of vehicles from a central depot to a number of delivery points. **Operations Research** 12: 568-581.

Cordeau, J., and Laporte, G. 2003. A tabu search heuristic for the static multi-vehicle dial-a-ride problem. **Transportation Research Part B** 37: 579-594.

Fisher, M. L., Jornsten, K. O., and Madsen, O. B. G. 1997. Vehicle routing with time windows: Two optimization algorithms. **Operations Research** 45: 488-492.

Gendreau, M., Laporte, G., and Seguin, R. 1996. Invited review stochastic vehicle routing. **European Journal of Operational Research** 88: 3-12.

Gillett, B. E., and Miller, L. R. 1974. A heuristic algorithm for the vehicle dispatching problem. **Operations Research** 35: 254-265.

Healy, P., and Moll R. 1995. Theory and methodology: A new extension of local search applied to the dial-a-ride problem. **European Journal of Operational Research** 83: 83-104.

Jaillet, P. 1988. A priori solution of a traveling salesman problem in which a random subset of the customers are visited. **Operations Research** 36: 929-936.

Laporte, G., Gendreau, M., Potvin, J., and Semet, F. 2000. Classical and modern heuristics for the vehicle routing problem. **International Transactions in Operational Research** 7: 285-300.

Lau, H. C., and Liang, Z. 2002. Pickup and delivery with time windows: Algorithms and test case generation. **International Journal on Artificial Intelligence Tools** 11: 455-472.

Li, H., and Lim, A. 2003. A metaheuristic for the pickup and delivery problem with time windows. **International Journal on Artificial Intelligence Tools** 12: 173-186.

Lu, Q., and Dessouky, M. 2004. An exact algorithm for the multiple vehicle pickup and delivery problem. **Transportation Science** 38: 503-514.

Minic, S. M. 1998. **Pickup and delivery problem with time windows: A survey.** (Unpublished Manuscript)

Nagy, G., and Salhi, S. 2005. Heuristic algorithms for single and multiple depot vehicle routing problems with pickups and deliveries. **European Journal of Operational Research** 162: 126-141.

Nanry, W. P., and Barnes, J. W. 2000. Solving the pickup and delivery problem with time windows using reactive tabu search. **Transportation Research Part B** 34: 107-121.

Psaraftis, H. N. 1983. An exact algorithm for the single vehicle many-to-many dial-a-ride problem with time windows. **Transportation Science** 17: 351-357.

Savelsbergh, M. W. P., and Sol, M. 1995. The general pickup and delivery problem. **Transportation Science** 29: 17-29.

Solomon, M. M. 1987. Algorithms for the vehicle routing and scheduling problems with time window constraints. **Operations Research** 35: 254-265.

Xiang, Z., Chu, C., and Chen, H. 2005. A fast heuristic for solving a large-scale static dial-a-ride problem under complex constraints. **European Journal of Operational Research.**

Xu, H., Chen, Z., Rajagopal, S., and Arunapuram, S. 2003. Solving a practical pickup and delivery problem. **Transportation Science** 37: 347-364.

ภาคผนวก

ภาคผนวก ก
การกำหนดค่าพารามิเตอร์สำหรับสร้างข้อมูลนำเข้าและเส้นทางการขนส่ง

สำหรับค่าพารามิเตอร์ที่ใช้สร้างข้อมูลนำเข้าและเส้นทางการขนส่ง ผู้วิจัยได้ทำการกำหนดขึ้นให้เป็นดังตารางที่ ก.1 และ ตารางที่ ก.2 ตามลำดับ

ตารางที่ ก.1 ค่าพารามิเตอร์สำหรับสร้างข้อมูลนำเข้า

พารามิเตอร์	ค่าของพารามิเตอร์
1.) เวลาพร้อมได้รับการขนส่งของเวชระเบียน	ยูนิฟอร์ม(0,180)
2.) โหลดของเวชระเบียน	1 หน่วย
3.) ความจุของรถ	150 หน่วย
4.) จำนวนหน่วยตรวจโรค	3-5 หน่วย

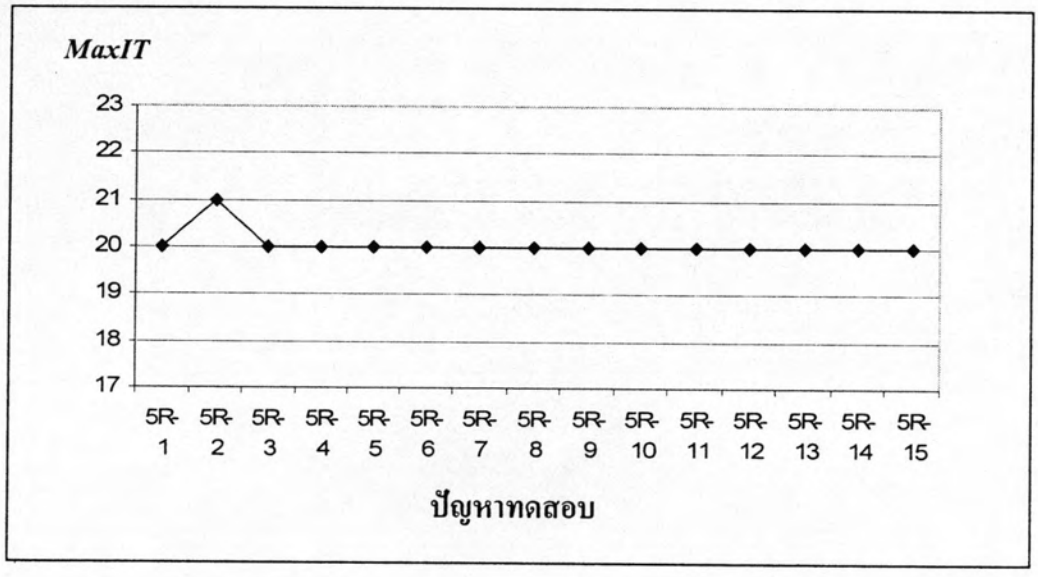
ตารางที่ ก.2 ค่าพารามิเตอร์สำหรับสร้างเส้นทางการขนส่ง

พารามิเตอร์	ค่าของพารามิเตอร์
1.) ระยะเวลารับประกัน	30 นาที
2.) ระยะเวลาในการให้บริการรับและส่งเวชระเบียน	0.05 นาที / ชิ้น
3.) ระยะเวลาในการเดินทาง	1 นาที / 1 หน่วยระยะทาง

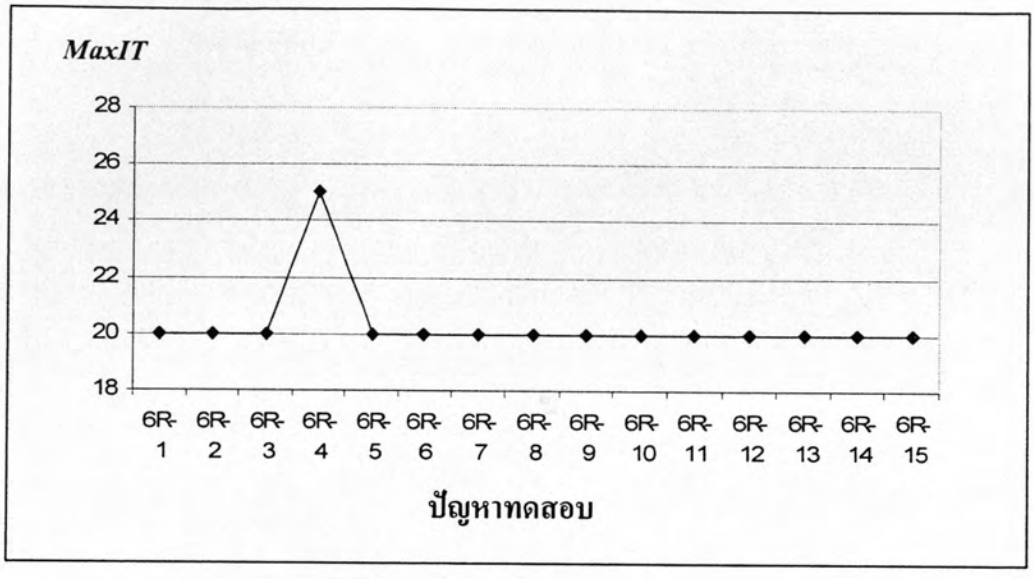
ภาคผนวก ข

การทดลองหาค่าพารามิเตอร์ *MaxIT* ที่เหมาะสม

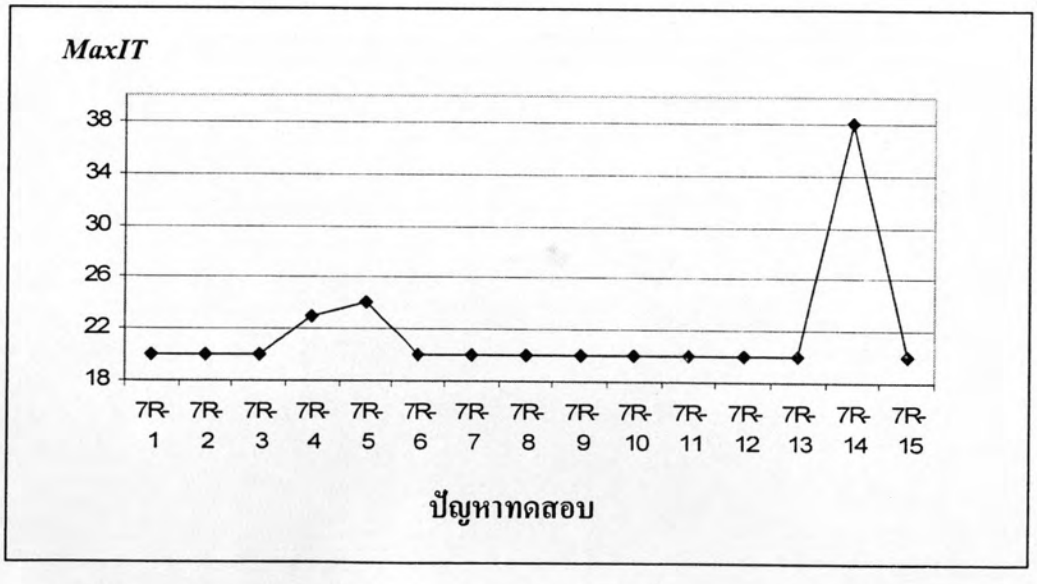
ผลการทดลองเพื่อหาค่า *MaxIT* ที่เหมาะสม แสดงดังรูปที่ ข.1 ถึง ข.13



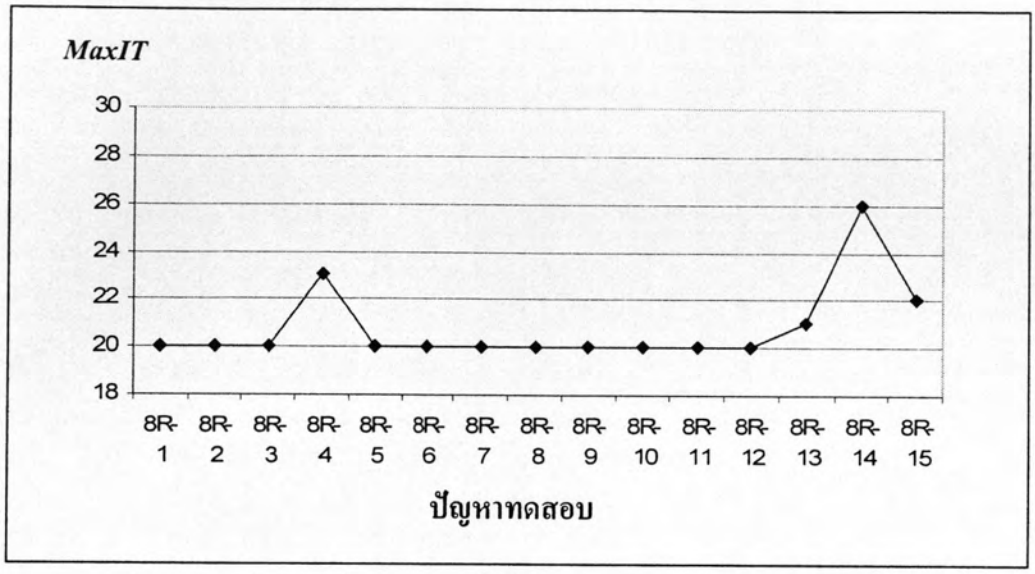
รูปที่ ข.1 การทดลองหาค่าของพารามิเตอร์ *MaxIT* ที่เหมาะสมสำหรับปัญหาทดสอบที่มีเวชระเบียนที่ต้องทำการขนส่ง 5 ชิ้น



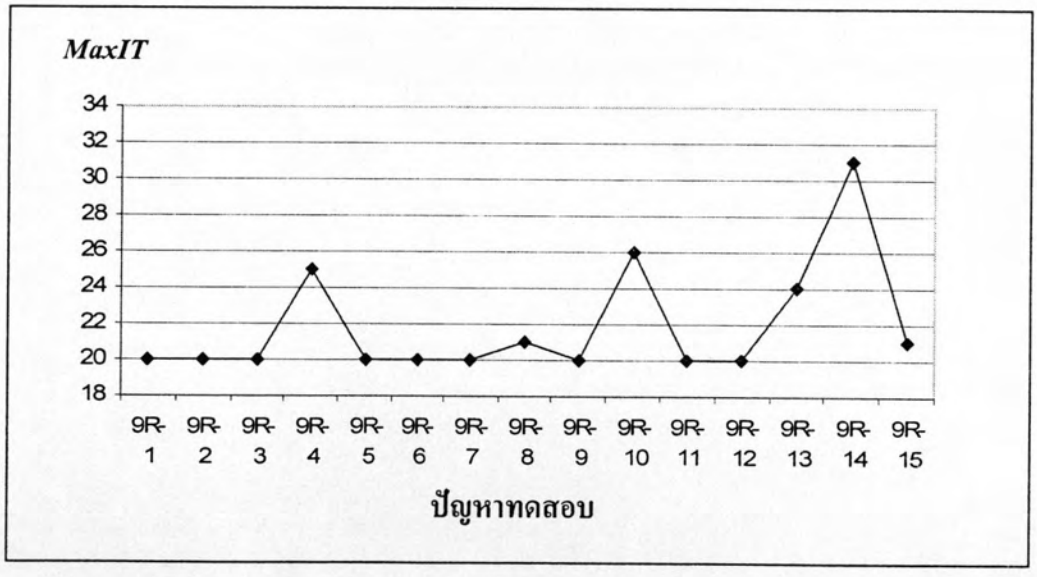
รูปที่ ข.2 การทดลองหาค่าของพารามิเตอร์ *MaxIT* ที่เหมาะสมสำหรับปัญหาทดสอบที่มีเวชระเบียนที่ต้องทำการขนส่ง 6 ชิ้น



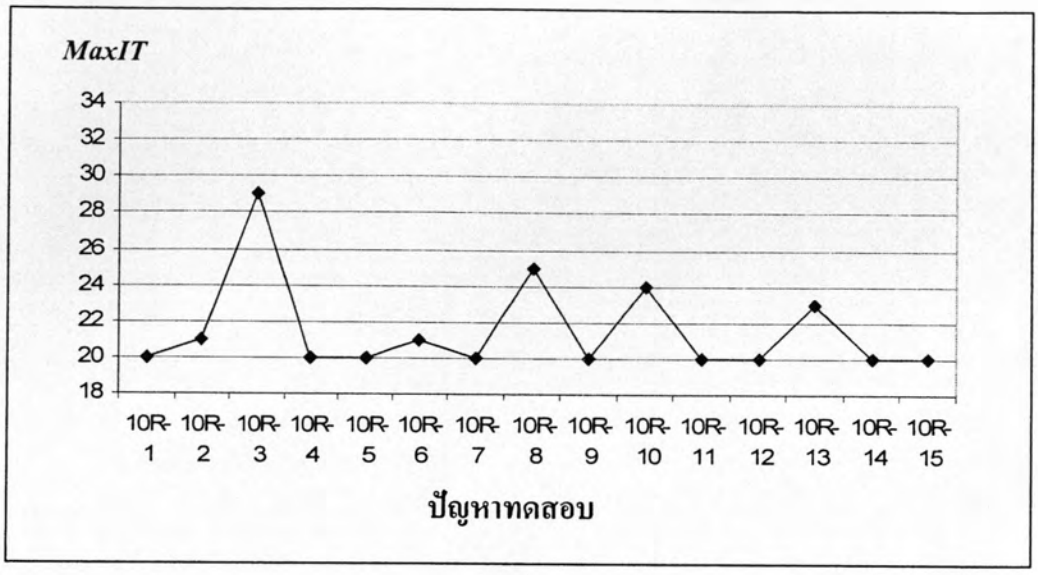
รูปที่ ข.3 การทดลองหาค่าของพารามิเตอร์ *MaxIT* ที่เหมาะสมสำหรับปัญหาทดสอบที่มีเวชระเบียนที่ต้องทำการขนส่ง 7 ชิ้น



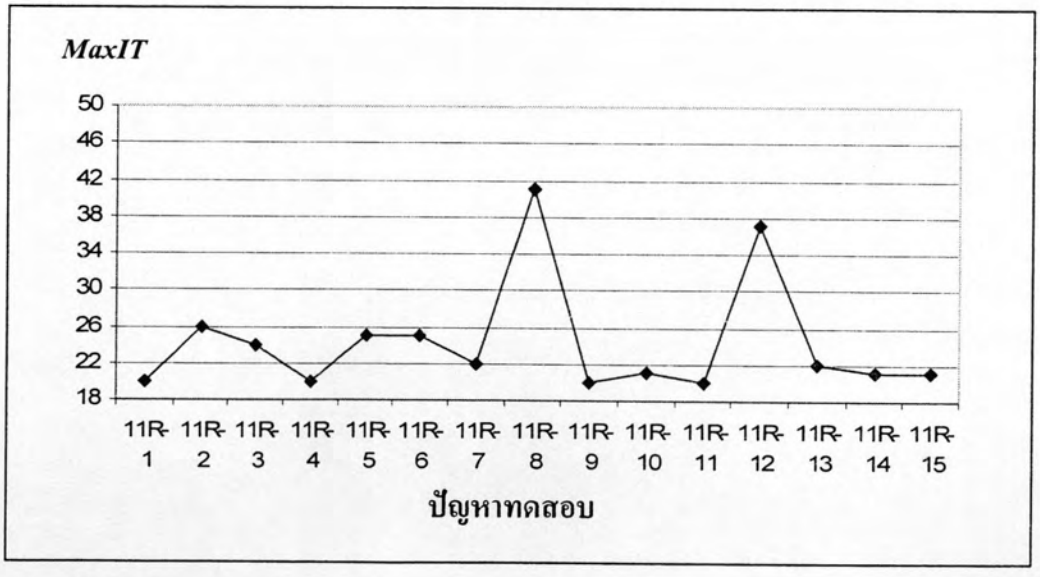
รูปที่ ข.4 การทดลองหาค่าของพารามิเตอร์ *MaxIT* ที่เหมาะสมสำหรับปัญหาทดสอบที่มีเวชระเบียนที่ต้องทำการขนส่ง 8 ชิ้น



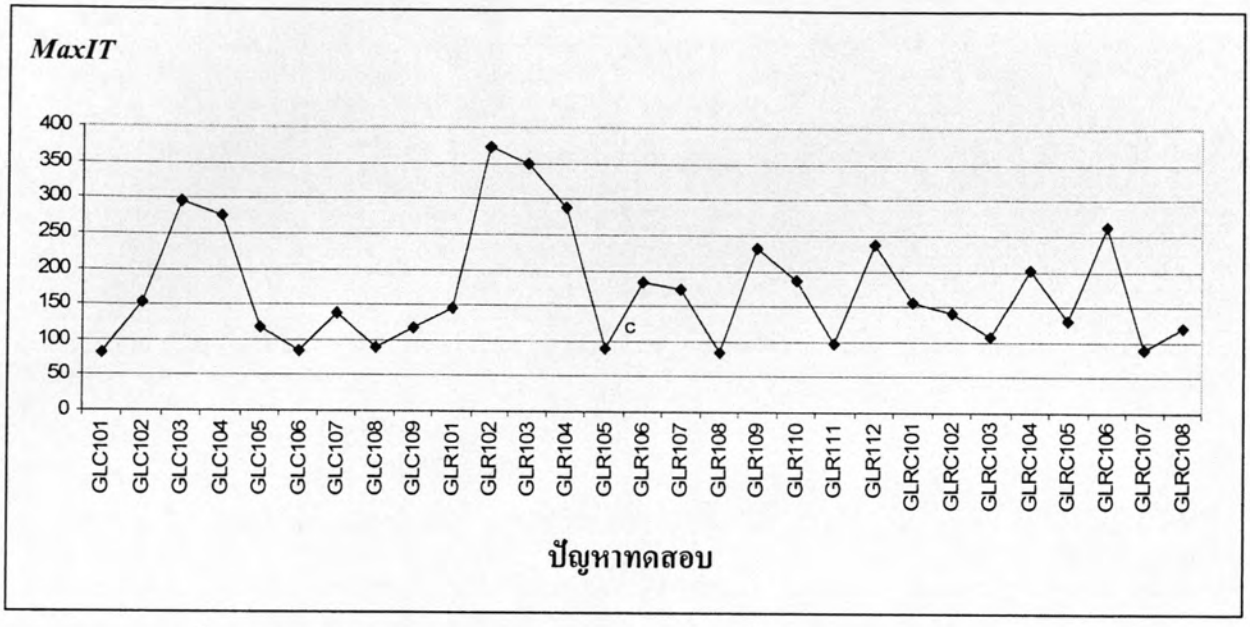
รูปที่ ข.5 การทดลองหาค่าของพารามิเตอร์ *MaxIT* ที่เหมาะสมสำหรับปัญหาทดสอบที่มีเวชระเบียนที่ต้องทำการขนส่ง 9 ชิ้น



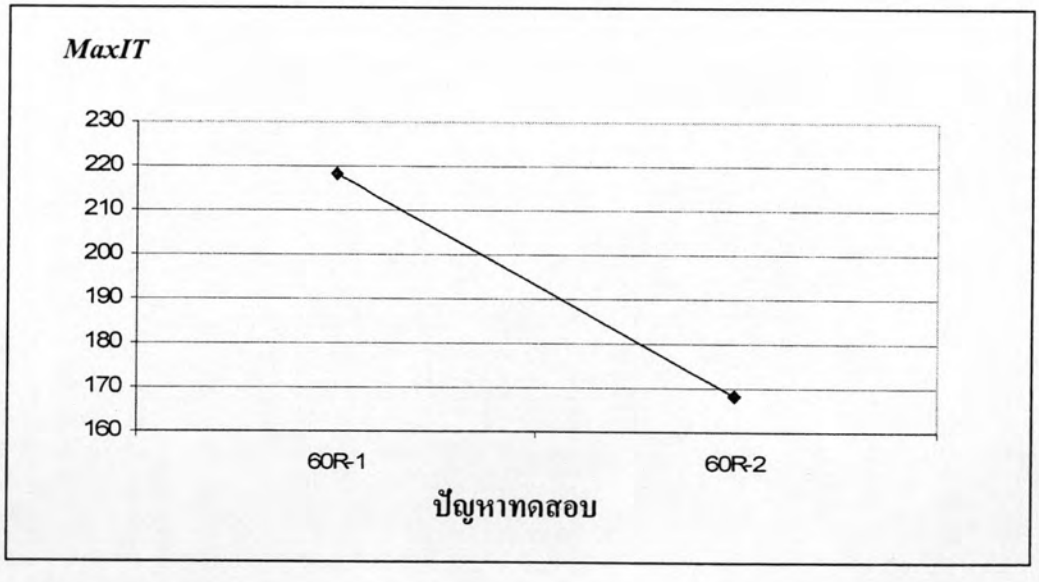
รูปที่ ข.6 การทดลองหาค่าของพารามิเตอร์ *MaxIT* ที่เหมาะสมสำหรับปัญหาทดสอบที่มีเวชระเบียนที่ต้องทำการขนส่ง 10 ชิ้น



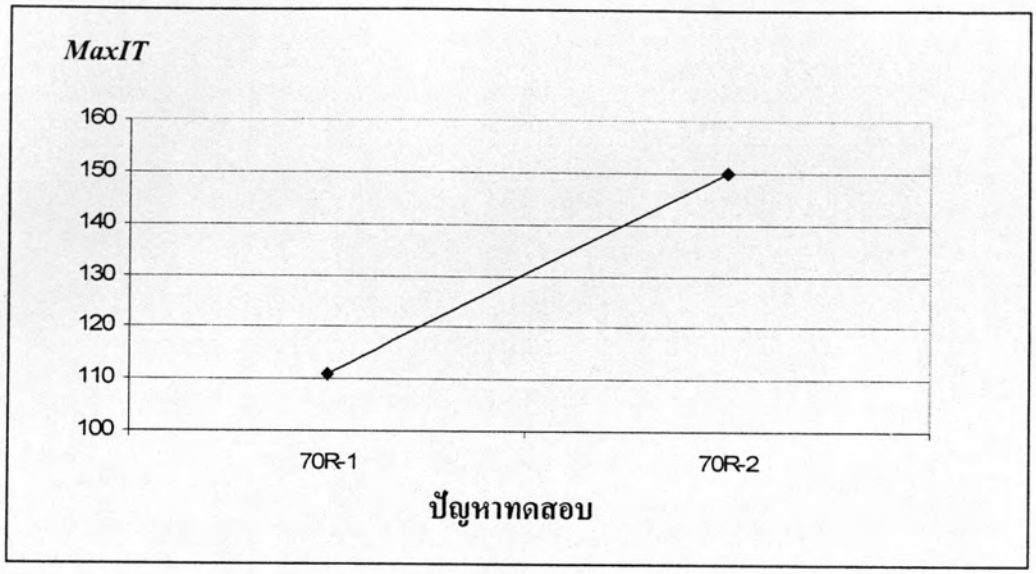
รูปที่ ข.7 การทดลองหาค่าของพารามิเตอร์ *MaxIT* ที่เหมาะสมสำหรับปัญหาทดสอบที่มีเวชระเบียนที่ต้องทำการขนส่ง 11 ชิ้น



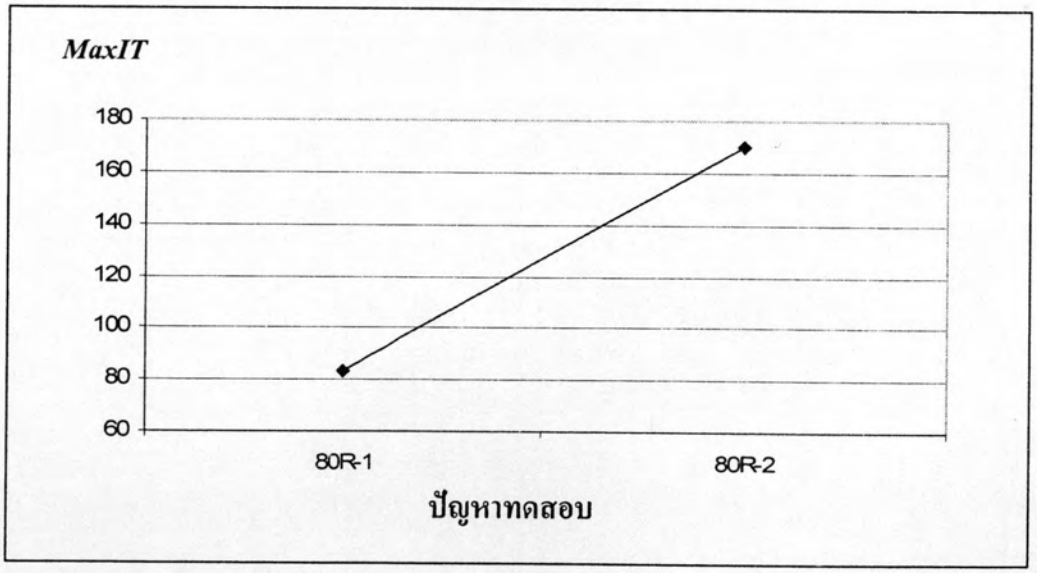
รูปที่ ข.8 การทดลองหาค่าของพารามิเตอร์ *MaxIT* ที่เหมาะสมสำหรับปัญหาทดสอบ PDPTW ที่ถูกดัดแปลงเป็นปัญหางานวิจัย



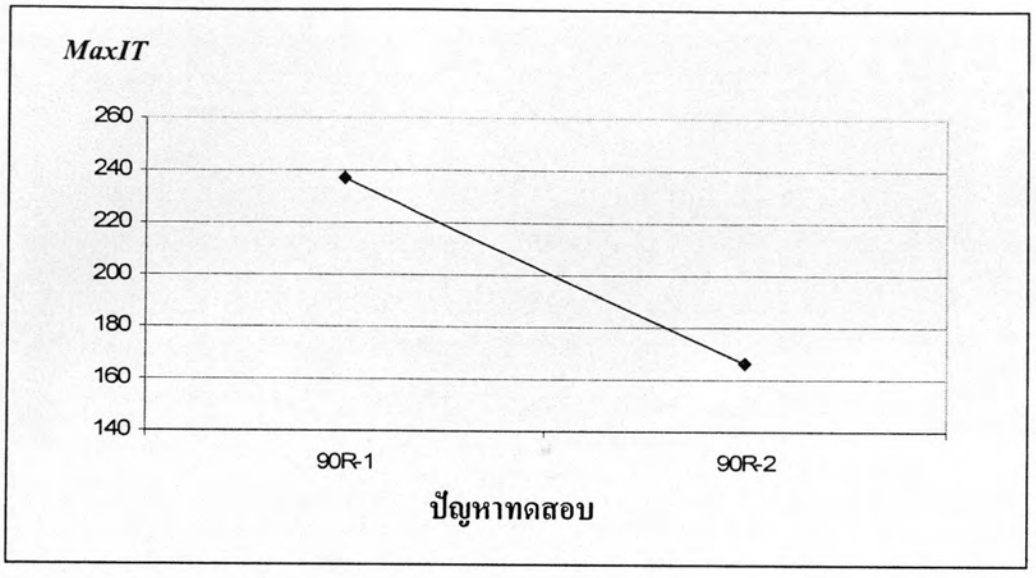
รูปที่ ข.9 การทดลองหาค่าของพารามิเตอร์ *MaxIT* ที่เหมาะสมสำหรับปัญหาทดสอบที่มีเวชระเบียนที่ต้องทำการขนส่ง 60 ชิ้น



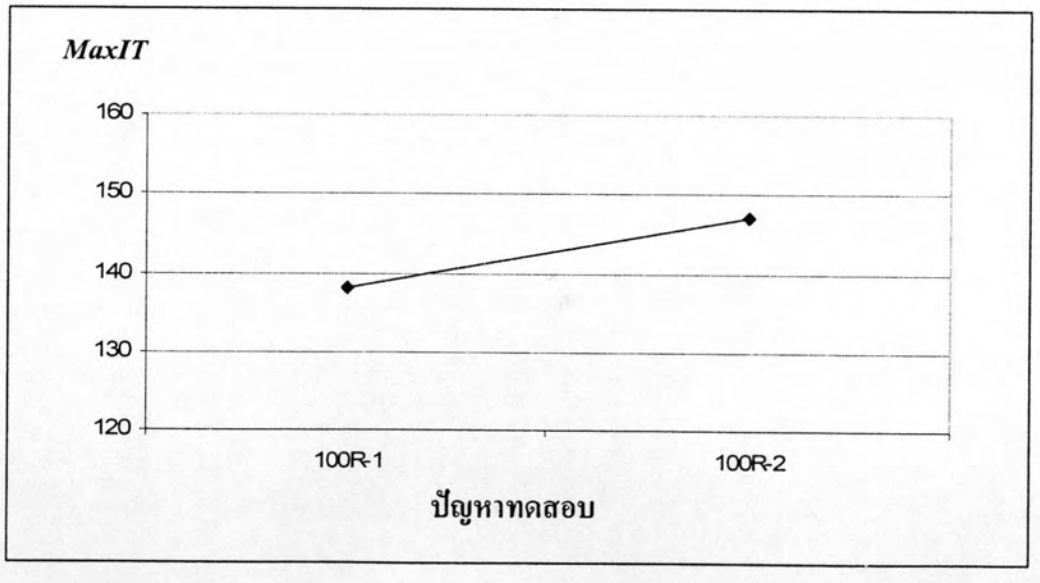
รูปที่ ข.10 การทดลองหาค่าของพารามิเตอร์ *MaxIT* ที่เหมาะสมสำหรับปัญหาทดสอบที่มีเวชระเบียนที่ต้องทำการขนส่ง 70 ชิ้น



รูปที่ ข.11 การทดลองหาค่าของพารามิเตอร์ *MaxIT* ที่เหมาะสมสำหรับปัญหาทดสอบที่มีเวชระเบียนที่ต้องทำการขนส่ง 80 ชิ้น



รูปที่ ข.12 การทดลองหาค่าของพารามิเตอร์ *MaxIT* ที่เหมาะสมสำหรับปัญหาทดสอบที่มีเวชระเบียนที่ต้องทำการขนส่ง 90 ชิ้น



รูปที่ ข.13 การทดลองหาค่าของพารามิเตอร์ *MaxIT* ที่เหมาะสมสำหรับปัญหาทดสอบที่มีเวชระเบียนที่ต้องทำการขนส่ง 100 ชิ้น

ภาคผนวก ค

แบบจำลองทางคณิตศาสตร์บน AMPL

```
### set Index ###
set PickupNode;
set DeliveryNode;
set PickupandDeliveryNode; # Pickup Node + Delivery Node
set AllNode;               # Hub + Pickup Node + Delivery Node
set PairsofPickupandDeliveryNode within {PickupNode,DeliveryNode};
set Cart;

### set Variables ###
var x{AllNode, AllNode, Cart}, binary;
var TimetoBeginService{PickupandDeliveryNode}, >=0;
var BeginningTime{Cart}, >=0;
var EndingTime{Cart}, >=0;
var LoadinCart{PickupandDeliveryNode}, >=0;
var y, integer >=0;

### set Parameters ###
param Distance{AllNode, AllNode};
param TravelTime{AllNode, AllNode};

param Demand{PickupandDeliveryNode};
param ReadyTime{PickupNode};

param ServiceTime;
param Capacity;
param GuaranteedTime;

param C;
param T;
param V;


### Model ###

# Obj Function #
minimize TotalCost: V*y + sum{i in AllNode, j in AllNode, k in Cart}
Distance[i,j]*x[i,j,k];

# Constraint #
s.t. Constraint_2{k in Cart}:
      sum{j in PickupNode} x[0,j,k] <= 1;

s.t. Constraint_3{j in DeliveryNode, k in Cart}:
      x[0,j,k] = 0;

s.t. Constraint_4{i in PickupNode, k in Cart}:
      x[i,0,k] = 0;

s.t. Constraint_5{j in PickupandDeliveryNode}:
      sum{i in AllNode, k in Cart : i<>j} x[i,j,k] = 1;

s.t. Constraint_6{i in PickupandDeliveryNode}:
      sum{j in AllNode, k in Cart : i<>j} x[i,j,k] = 1;

s.t. Constraint_7{j in PickupandDeliveryNode, k in Cart}:
      sum{h in AllNode: j<>h} x[j,h,k] = sum{g in AllNode : g<>j} x[g,j,k];
```

```
s.t. Constraint_8{(p,d) in PairsofPickupandDeliveryNode, k in Cart}:
       sum{q in AllNode: q <> p} x[q,p,k] = sum{s in PickupandDeliveryNode:
       s <> d} x[s,d,k];

s.t. Constraint_9{(p,d) in PairsofPickupandDeliveryNode}:
       TimetoBeginService[p] <= TimetoBeginService[d];

s.t. Constraint_10{p in PickupNode}:
       TimetoBeginService[p] >= ReadyTime[p];

s.t. Constraint_11{j in PickupNode, k in Cart}:
       BeginningTime[k] + TravelTime[0,j] - (1 - x[0,j,k])*T <=
       TimetoBeginService[j];

s.t. Constraint_12{i in PickupandDeliveryNode, j in PickupandDeliveryNode,
     k in Cart}:
       (TimetoBeginService[i] + ServiceTime + TravelTime[i,j]) -
       (1 - x[i,j,k])*T <= TimetoBeginService[j];

s.t. Constraint_13{i in PickupandDeliveryNode, k in Cart}:
       (TimetoBeginService[i] + ServiceTime + TravelTime[i,0]) -
       (1 - x[i,0,k])*T <= EndingTime[k];

s.t. Constraint_14{(p,d) in PairsofPickupandDeliveryNode}:
       (TimetoBeginService[d] - ReadyTime[p]) <= GuaranteedTime;

s.t. Constraint_15{i in PickupandDeliveryNode}:
       LoadinCart[i] <= Capacity;

s.t. Constraint_16{j in PickupNode, k in Cart}:
       Demand[j] - (1 - x[0,j,k])*C <= LoadinCart[j];

s.t. Constraint_17{i in PickupandDeliveryNode, j in PickupandDeliveryNode,
     k in Cart}:
       LoadinCart[i] + Demand[j] - (1 - x[i,j,k])*C <= LoadinCart[j];

s.t. Constraint_18:
       sum{j in PickupandDeliveryNode, k in Cart} x[0,j,k] = y;

s.t. Constraint_19{k1 in Cart, k2 in Cart : k2-k1=1}:
       sum{j in PickupNode} x[0,j,k1] >= sum{j in PickupNode} x[0,j,k2];
```

ภาคผนวก ง

โปรแกรมสร้างข้อมูลนำเข้าสำหรับทดสอบฮิวริสติก

```
/* "A HEURISTIC METHODOLOGY TO CREATE PICKUP AND DELIVERY ROUTES WITH
GUARANTEED TIME CONSTRAINT" */
/* Source Code - Generate - Input Data */
/* Written by Mr.Tripoom Punkiti */

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>

#define labeled 1
#define nolabeled 0

/* Global Variables */
FILE *fp;
int Clinics,Records;
int *Xloc,*Yloc;
int *PickupClinic,*DeliveryClinic;
int *XN,*YN;

int i,j;

int r,lr;
float rn,fract=100;

float dist,**Distance;
int dx,dy;

int *ReadyTime;

float ServiceTime = 0.05;
int GuaranteedTime = 30;

float ConstantTime = 1;

int *Demand;

int Capacity = 150;

float BigM = 10000;
float ProbofExistEdge = 0.6;

/* Declare Prototype */
float CalculateShortestPath(int i, int j);
float STP[50][50];
float DistanceinTable[50][50];
int NodeinTable[50][50];
int NodeinP[50];


/* Declare Prototype */
void AllocateMemory();
void FreeMemory();


main(int argc, char *argv[])
{
```

```
   char probname[100];

   if(argc != 2)
   {
      printf("Error in command line argument - Don't forget input filename of
data or output file!!!");
      exit(1);
   }

    clrscr();

   printf("************** Generate Problem for Heuristic ***********\n\n");

   printf("For Thesis - Input the number of Clinics : ");
   scanf("%d", &Clinics);
   printf("Clinics = %d\n", Clinics);

   printf("For Thesis - Input the number of Records : ");
   scanf("%d", &Records);
   printf("Records = %d\n", Records);

   printf("Please input NO. of Iteration to pickup 'Random Number Seed'\n");
   printf("for Generate Problem : ");
   scanf("%d", &lr);

   strcpy(probname, argv[1]);
   fp = fopen(probname,"w");

   for(i=1; i<lr; i++)
   {
      r = rand();
   }

   AllocateMemory();

   fprintf(fp, "****************** INPUT *****************\n");

   /* Write Title into File */
   fprintf(fp, "Problem        %dR-%d\n\n", Records, lr);

   fprintf(fp, "Clinics        %d\n", Clinics);
   fprintf(fp, "Records        %d\n\n", Records);

   /* Write Capacity */
    fprintf(fp, "Capacity       %d\n\n", Capacity);

   /* Write Capacity */
    fprintf(fp, "ServiceTime    %.2f\n", ServiceTime);
   /* Write Capacity */
    fprintf(fp, "GuaranteedTime %d\n\n", GuaranteedTime);

   /* Write Clinic NO. */
   fprintf(fp, "ClinicNO.         XCoor.      YCoor.\n");
   for(i=0; i<=Clinics; i++)
   {
      if(i==0)
      {
         fprintf(fp, "    0              0           0\n");
         Xloc[0] = 0;
         Yloc[0] = 0;
      }
      else if(i>0)
      {
         fprintf(fp, "%5d", i);
```

```
            r = rand();
            rn = (r % 100)/fract ; /*rn is random 0 - 1 */
            Xloc[i] = 20*rn;    /* x[i] is uniform(0,20) */
            fprintf(fp, "%16d", Xloc[i]);

            r = rand();
            rn = (r % 100)/fract ; /*rn is random 0 - 1 */
            Yloc[i] = 20*rn;    /* y[i] is uniform(0,20) */
            fprintf(fp, "%12d\n", Yloc[i]);
         }
      }
      fprintf(fp, "\n");


      /* Write avaliable edge */
      fprintf(fp, "# Avaliable Edge\n");
      for(i=0; i<=Clinics; i++)
      {
         for(j=0; j<=Clinics; j++)
         {
            if(i<=j)
            {
               dx = Xloc[i] - Xloc[j];
               dy = Yloc[i] - Yloc[j];

               dist = sqrt(pow(dx,2) + pow(dy,2));

               r = rand();
               rn = (r % 100)/fract ; /*rn is random 0 - 1 */
               if((i==j) || (rn <= ProbofExistEdge))
               {
                  Distance[i][j] = dist;
                  fprintf(fp, "  [C%d,C%d]  %10.4f\n", i, j, Distance[i][j]);
               }
               else if((i!=j) && (rn > ProbofExistEdge))
               {
                  Distance[i][j] = BigM;
                  fprintf(fp, "  [C%d,C%d]  %10.4f (inf.)\n", i, j,
Distance[i][j]);
               }
            }
            else if(i>j)
            {
               Distance[i][j] = Distance[j][i];
               if(Distance[j][i] < BigM)
               {

                  fprintf(fp, "  [C%d,C%d]  %10.4f\n", i, j, Distance[j][i]);
               }
               else if(Distance[j][i] == BigM)
               {
                  fprintf(fp, "  [C%d,C%d]  %10.4f (inf.)\n", i, j,
Distance[j][i]);
               }
            }
         }
         fprintf(fp, "\n");
      }
      fprintf(fp, "\n");


      /* Write Shortest Path */
      fprintf(fp, "#ShortestPath\n");
```

```
   for(i=0; i<=Clinics; i++)
   {
      for(j=0; j<=Clinics; j++)
      {
         if(i==j)
         {
            STP[i][j] = 0;
            fprintf(fp, "  STP[C%d,C%d]   %10.4f      -\n", i, j, STP[i][j]);
         }
         else
         {
            STP[i][j] = CalculateShortestPath(i,j);
         }


      }
      fprintf(fp, "\n");
   }
   fprintf(fp, "\n");


   /* Write Record NO. */
   fprintf(fp, "RecordNO.  PickupClinic   DeliveryClinic   ReadyTime
Demand\n");
   for(i=1; i<=Records; i++)
   {
      /* Write Record NO. */

      fprintf(fp, "%5d", i);
      /* Random Pickup Clinic */
      r = rand();
      rn = ((r % (Clinics+1)));
      PickupClinic[i] = rn;
      fprintf(fp, "%12d", PickupClinic[i]);

      /* Random Delivery Clinic */
      do
      {
         r = rand();
         rn = ((r % Clinics) + 1);
      }
      while (rn == PickupClinic[i]);

      DeliveryClinic[i] = rn;
      fprintf(fp, "%16d", DeliveryClinic[i]);

      /* Random Ready Time */
      r = rand();
      rn = (r % 100)/fract;

      ReadyTime[i] = 180*rn;  /* ReadyTime => uniform(0,180) */
      fprintf(fp, "%15d", ReadyTime[i]);

      /* Random Demand */
      r = rand();
      Demand[i] = 1;
      fprintf(fp, "%10d\n", Demand[i]);
   }


   printf("\n\nFinished");

   fclose(fp);
   FreeMemory();
```

```
}


/************************************************/
/* Function AllocateMemory()
   - Uses to allocate memory.
/************************************************/
void AllocateMemory()
{
   int i;

   Xloc = malloc((Clinics+1)*sizeof(int));
   Yloc = malloc((Clinics+1)*sizeof(int));

   PickupClinic = malloc((Records+1)*sizeof(int));
   DeliveryClinic = malloc((Records+1)*sizeof(int));

   XN = malloc((2*Records+1)*sizeof(int));
   YN = malloc((2*Records+1)*sizeof(int));

   ReadyTime = malloc((Records+1)*sizeof(int));

   Demand = malloc((2*Records+1)*sizeof(int));

   Distance = malloc((2*Records+1)*sizeof(float));
   for(i=0; i<=2*Records; i++)
   {
      Distance[i] = malloc((2*Records+1)*sizeof(float));
   }
}


/************************************************/
/* Function FreeMemory()
   - Uses to free memory.
/************************************************/
void FreeMemory()
{
   int i;

   free(Xloc);
   free(Yloc);

   free(PickupClinic);
   free(DeliveryClinic);

   free(XN);
   free(YN);

   free(ReadyTime);

   free(Demand);

   for(i=0; i<=2*Records; i++)
   {
      free(Distance[i]);
   }
}


/************************************************/
/* Function CalculateShortestPath();
   - Uses to find shortest path of any node (i,j)
     by using Dijkstra's Algorithm.
```

```
/************************************************/
float CalculateShortestPath(int nodei, int nodej)
{
   int n;
   int LabeledNode[50], LabeledNode_Iteration[50];
   float MinDist[50],temp;
   int Iteration;
   int x,y;
   float stp=0;


   /* Clear */
   for(x=0;x<=Clinics;x++)
   {
      NodeinP[x] = 0;
      LabeledNode[x] = nolabeled;
      for(y=0; y<=Clinics; y++)
      {
         DistanceinTable[x][y] = 0;
         NodeinTable[x][y] = 0;
      }
   }

   /* Step 1 */
   DistanceinTable[1][nodei] = 0;
   NodeinTable[1][nodei] = nodei;
   NodeinP[1] = nodei;
   LabeledNode[nodei] = labeled; /* labeled = 1 */
   LabeledNode_Iteration[1] = nodei;

   /* Step 2 */
   for(n=0; n<=Clinics; n++)
   {
      if(n != nodei)
      {
         if(Distance[nodei][n] < BigM)
         {
            DistanceinTable[2][n] = Distance[nodei][n];
            NodeinTable[2][n] = nodei;

         }
         else
         {
            DistanceinTable[2][n] = BigM;
            NodeinTable[2][n] = nodei;
         }
      }
   }

   /* Find Shortest */
   MinDist[2] = BigM; /* initial MinDist */
   for(n=0; n<=Clinics; n++)
   {
      if(LabeledNode[n] == nolabeled)
      {
         if(DistanceinTable[2][n] < MinDist[2])
         {
            MinDist[2] = DistanceinTable[2][n];
            NodeinP[2] = n;
         }
      }
   }
   LabeledNode[NodeinP[2]] = labeled;
```

```
   LabeledNode_Iteration[2] = NodeinP[2];


   /* Step 3,4, ... */
   Iteration = 3;
   while(NodeinP[Iteration-1] != nodej)
   {
      for(n=0; n<=Clinics; n++)
      {
         if(LabeledNode[n] != labeled)
         {
            temp = Distance[NodeinP[Iteration-1]][n] + MinDist[Iteration-1];

            if(temp < DistanceinTable[Iteration-1][n])
            {
               DistanceinTable[Iteration][n] = temp;
               NodeinTable[Iteration][n] = NodeinP[Iteration-1];
            }
            else
            {
               DistanceinTable[Iteration][n] = DistanceinTable[Iteration-
1][n];
               NodeinTable[Iteration][n] = NodeinTable[Iteration-1][n];
            }
         }
      }

      /* Find Shortest */
      MinDist[Iteration] = BigM; /* initial MinDist */
      for(n=0; n<=Clinics; n++)
      {
         if(LabeledNode[n] == nolabeled)
         {
            if(DistanceinTable[Iteration][n] < MinDist[Iteration])
            {
               MinDist[Iteration] = DistanceinTable[Iteration][n];
               NodeinP[Iteration] = n;
            }
         }
      }
      LabeledNode[NodeinP[Iteration]] = labeled;
      LabeledNode_Iteration[Iteration] = NodeinP[Iteration];

      Iteration = Iteration+1;
   }

   /* Print Table */
   for(x=1; x<Iteration; x++)
   {
      for(y=0; y<=Clinics; y++)
      {
         printf(" %f(%d)  ", DistanceinTable[x][y], NodeinTable[x][y]);
      }
      printf("%d  \n", NodeinP[x]);
   }

   /* Print Shortest Path */
   stp = DistanceinTable[Iteration-1][NodeinP[Iteration-1]];
   fprintf(fp, "  STP[C%d,C%d]   %10.4f      ", nodei, nodej, stp);

   fprintf(fp,"C%d", NodeinP[Iteration-1]);

   /* Print STP */
   x = Iteration-1;
```

```
   while(x > 1)
   {
      fprintf(fp," <- C%d", NodeinTable[x][LabeledNode_Iteration[x]]);

      n = 1;
      while(n < x)
      {
         if(NodeinP[n] == NodeinTable[x][LabeledNode_Iteration[x]])
         {
            x = n;
            break;
         }
         n = n + 1;
      }
   }

   fprintf(fp,"\n");

   return stp;
}
```

ภาคผนวก จ

โปรแกรมทดสอบฮิวริสติก

```
/* "A HEURISTIC METHODOLOGY TO CREATE PICKUP AND DELIVERY ROUTES WITH
GUARANTEED TIME CONSTRAINT" */
/* Source Code - Heuristic */
/* Written by Mr.Tripoom Punkiti */

#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<string.h>
#include<time.h>

#define TRUE 1
#define FALSE 0

#define FOUND 1
#define NOTFOUND 0

#define ACCEPT 1
#define REJECT 0


FILE *fpr, *fpo;
double EPS;

/* Clock */
clock_t start, finish;
double  duration;

/* Declare Variables for Read Data */
int Records, Clinics;
int *PickupClinic,*DeliveryClinic;
double **Distance, **STDistance;
double *ReadyTime;
int Capacity;
double ServiceTime;
double GuaranteedTime;
int *Demand;

/* Declare Variables for Heuristic */
double **SavingMatrix;
int **Route, **BestRoute;
int **BestNBRoute;
double *TimetoBeginService;
double **TravelTime;
int *LoadinVehicle;
double TotalCost, BestTotalCost;
double TotalCostofBestNB;
double BestTTT,BestTWT;
int Vehicles;
int NOInitialRoutes;
int Iteration, BestIteration;
int MaxNoImprove;
int IT;

int *PDList, *RecordList;
```

```
/* Variables for One Route V=1 */
int **NBRoute;

int CFS;

/* Declare General Variables */
int i,j,n,r,s,rt;
double dx,dy;
double dist;
double ConstantTime = 1.0;
char str[100];
char datafile[50];

/* TempRoute */
int *OneTR1, *OneTR2, **ManyTR, **ManyBTR;
int *OneTRF, *OneTRSolomon, *OneTRFSolomon, **ManyTRSF;

/* Parameters */
float alpha = 0.4;
float beta = 0.4;
int TabuIteration = 30;
int MaxIteration = 250;

 /* Tabu structure to represent a solution */
int *NV;     /* Number of Vehicles */
double *TD;  /* Total Distances */
double *TTT; /* Total Travel Time */
double *TWT; /* Total Waiting time */

/* Solomon Insertion */
double *MinimumInsertionCostofRecord;
int *BestOriginPositionofRecord, *BestDestinationPositionofRecord;
int *FeasibleInsertionPlace;


/********************** Declare Prototypes **********************/
void ReadData();

void SetInitialRoutes();
void SetInitialTabuList();

int ConstructInitialSolution_SolomonInsertion();

int FindUnroutedRecordwithEarliestReadyTime(int *PDList);

double CalculateCostofConstructionNewRoute(int RC);

int CheckGuaranteedTime(int *Route);
int CheckCapacity(int *Route);
int CheckPredecessorSuccessor(int *Route);
int CheckFeasibleRoute(int *Route);
int CheckFeasibleSolution(int **Route);


/* Prototype - Local Search */
void LocalSearch();
int FindSolution_HighestPositiveSavingCost_PDShiftOperator(int
*PDShift_BestRecord, int *PDShift_BestRoute1, int *PDShift_BestRoute2, int
*PDShift_BestOriginPosition, int *PDShift_BestDestinationPosition);
int FindSolution_HighestPositiveSavingCost_PDExchangeOperator(int
*PDExchange_Record1, int *PDExchange_Route1, int
*PDExchange_BestOriginPosition_P1intoR2, int
*PDExchange_BestDestinationPosition_P1intoR2, int *PDExchange_Record2, int
```

```
*PDExchange_Route2, int *PDExchange_BestOriginPosition_P2intoR1, int
*PDExchange_BestDestinationPosition_P2intoR1);
int
FindSolution_HighestPositiveSavingCost_RearrangeOperator_WithinRouteInsertio
n(int *WRI_Node, int *WRI_Route, int *WRI_BestPosition);
int
FindSolution_HighestPositiveSavingCost_RearrangeOperator_Swap3ConsecutivePos
itionsWithinRoute(int *OneRoute);
int ReduceRoute_HighestPositiveSavingCost();

/* Prototype - Tabu Search */
void TabuSearch();
int ConstructandPickupBestNeighborSolution(int *Route1, int *Route2);
int FindNeighbor_MinimumCost_PDShiftOperator(int *PDShift_BestRecord, int
*PDShift_BestRoute1, int *PDShift_BestRoute2, int
*PDShift_BestOriginPosition, int *PDShift_BestDestinationPosition, double
*PDShift_BestTotalCost);
int FindNeighbor_MinimumCost_PDExchangeOperator(int *PDExchange_BestRecord1,
int *PDExchange_BestRoute1, int *PDExchange_BestOriginPosition_P1intoR2, int
*PDExchange_BestDestinationPosition_P1intoR2, int *PDExchange_BestRecord2,
int *PDExchange_BestRoute2, int *PDExchange_BestOriginPosition_P2intoR1, int
*PDExchange_BestDestinationPosition_P2intoR1, double
*PDExchange_BestTotalCost);
int RearrangeRoute_MinimumWaitingTime(int *RT);

/* Prototype - Neighborhood move Operators */
void PDShiftOperator(int BestRecord, int *RT1, int *RT2, int
BestOriginPosition, int BestDestinationPosition);
void PDExchangeOperator( int P1, int *RT1, int P2, int *RT2, int
OriginPosition_P1intoR2, int DestinationPosition_P1intoR2, int
OriginPosition_P2intoR1, int DestinationPosition_P2intoR1);
void RearrangeOperator_WithinRouteInsertion(int Node, int *Route, int
BestPosition);

void RemoveNode(int RemoveNode, int *Route);
void InsertNode(int Node, int *Route, int Position);
void RemoveRecord(int RemoveRecord, int *Route);
void InsertRecord(int RandomRecord, int *Route, int OriginPosition, int
DestinationPosition);

/* Prototype - The Solution has only one Route */
void ConstructandPickupBestBeighborOneRoute(int *Route);

int CountNORecordsinRoute(int *Route);

void CopyOneRoute(int *R1, int *R2);
void CopyAllRoutes(int **R1, int **R2);
void RearrangeAllRoutes(int **Route);
int FindRoute(int PD);
double EvaluateCostOneRoute(int *Route);
double EvaluateCostAllRoutes(int **Route);
double EvaluateWaitingTimeOneRoute(int *Route);

void PrintOneRoute(int *Route, int r);
void PrintAllRoutes(int **Route);
void FPrintAllRoutes(FILE *f, int **Route);

int CountVehicles(int **Route);
double CalculateTotalDistance(int **Route);
double CalculateTotalTravelTime(int **Route);
double CalculateTotalWaitingTime(int **Route);

void UpdateTabuList(int **Route);
int CheckSolutioninTabuList(int **TempRoute);
```

```
void CreateRoutes(FILE *f, int **Route, int *PickupClinic, int
*DeliveryClinic);
void TransformSolution(FILE *f, int **Route, int *PickupClinic, int
*DeliveryClinic);

void PrintOutput();

void AllocateMemory();
void FreeMemory();

void ClearRoute(int *Route);
void ClearRoutes(int **Route);

/******************* End Declare Prototypes ***********************/


main(int argc, char *argv[])
{
   int InitialRoutes;

   if(argc != 3)
   {
      printf("Error in command line argument - Don't forget input filename of
data or output file!!!");
      exit(1);
   }

   strcpy(datafile, argv[1]);

   printf("Data File = %s\n\n", datafile);

   fpo = fopen(argv[2], "w");

   /* Write Output Title Files */
   fprintf(fpo, "***************** OUTPUT **********************\n");
   fprintf(fpo, "Problem        %s\n\n", argv[2]);

   ReadData();

   EPS = pow(10,-6);

   start = clock();

   SetInitialRoutes();

   /* Find Initial Solution */
   NOInitialRoutes = ConstructInitialSolution_SolomonInsertion();
   PrintAllRoutes(Route);
   TotalCost = EvaluateCostAllRoutes(Route);
   printf("\nTotal Initial Cost = %.2lf\n", TotalCost);

   /* Keep Best */
   CopyAllRoutes(Route, BestRoute);
   BestTotalCost = TotalCost;


   TabuSearch();


   /* Print Current Route */
   printf("\nPrint All Current Route\n");
   PrintAllRoutes(Route);
   TotalCost = EvaluateCostAllRoutes(Route);
```

```
   printf("\nCurrent Total Cost = %.2lf\n", TotalCost);

   /* Print Best Route */
   printf("\nPrint All Best Route\n");
   PrintAllRoutes(BestRoute);
   TotalCost = EvaluateCostAllRoutes(BestRoute);
   printf("\nBest Total Cost = %.2lf\n", TotalCost);
   BestTTT = CalculateTotalTravelTime(BestRoute);
   printf("Best Total Travel Time = %.2lf\n", BestTTT);
   BestTWT = CalculateTotalWaitingTime(BestRoute);
   printf("Best Total Waiting Time = %.2lf\n\n", BestTWT);

   finish = clock();

    duration = (double)(finish - start) / CLOCKS_PER_SEC;
   printf("Duration Time = %0.4lf sec\n\n", duration);
   printf("Iteration = %d\n", Iteration);
   printf("Best Iteration = %d\n\n", BestIteration);
   printf("Max NoImprove = %d\n\n", MaxNoImprove);

   CFS = CheckFeasibleSolution(BestRoute);
   printf("BestRoute CFS = %d\n", CFS);

   CFS = CheckFeasibleSolution(Route);
   printf("Route CFS = %d\n", CFS);

   /* Write Output to Output Files */
   PrintOutput();

   FreeMemory();

}


/*************************************************************/
/* Function : ReadData()
/* - Read Data from Data File.
/*
/*************************************************************/
void ReadData()
{
   fpr = fopen(datafile, "r");


   /* Read first line */
   fscanf(fpr, "%s", str);
   fscanf(fpr, "%s", str);
   fscanf(fpr, "%s", str);

   /* Read Title */
   fscanf(fpr, "%s", str); /* Read Problem */
   fscanf(fpr, "%s", str); /* Read Problem Name */

   /* Read NO.Clinics */
   fscanf(fpr, "%s", str); /* Read a Word 'Clinics' */
   fscanf(fpr, "%d", &Clinics);
   printf("Clinics = %d\n", Clinics);

   /* Read NO.Records */
   fscanf(fpr, "%s", str); /* Read a Word 'Records' */
   fscanf(fpr, "%d", &Records);
   printf("Records = %d\n", Records);
```

```
   AllocateMemory();

   /* Read Capacity of Vehicles */
   fscanf(fpr, "%s", str); /* Read a Word 'Capacity' */
   fscanf(fpr, "%d", &Capacity);
   printf("Capacity = %d\n", Capacity);

   /* Read Service Time */
   fscanf(fpr, "%s", str); /* Read a Word 'ServiceTime' */
   fscanf(fpr, "%lf", &ServiceTime);
   printf("Service Time = %.2lf\n", ServiceTime);

   /* Read Guaranteed Time */
   fscanf(fpr, "%s", str); /* Read a Word 'GuaranteedTime' */
   fscanf(fpr, "%lf", &GuaranteedTime);
   printf("Guaranteed Time = %.2lf\n", GuaranteedTime);


   /* Read Clinic */
   for(i=-1; i<=Clinics; i++)
   {
      if(i == -1) /* Title */
      {
         fscanf(fpr, "%s", str);
         fscanf(fpr, "%s", str);
         fscanf(fpr, "%s", str);
      }
   }

   /* Read through '#ShortestPath */
   do
   {
      fscanf(fpr, "%s", str);
   }
   while(!(str[0]=='#' && str[1]=='S' && str[2]=='h' && str[3]=='o' &&
str[4]=='r' && str[5]=='t'));

   /* Write Shortest Path in Output files */
   fprintf(fpo, "%s\n", str);

   for(i=0; i<=Clinics; i++)
   {
      for(j=0; j<=Clinics; j++)
      {
         fscanf(fpr, "%s", str);
         fprintf(fpo, "%s", str);

         fscanf(fpr, "%lf", &STDistance[i][j]);
         fprintf(fpo, "   %10.4lf", STDistance[i][j]);

         fgets(str, 100, fpr);
         fprintf(fpo, "%s", str);
      }
      fgets(str, 100, fpr);
      fprintf(fpo, "%s", str);
   }

   /* Read Record Data */
   for(i=0; i<=Records; i++)
   {
      if(i == 0) /* Title */
      {
         fscanf(fpr, "%s", str);
         fscanf(fpr, "%s", str);
```

```
            fscanf(fpr, "%s", str);
            fscanf(fpr, "%s", str);
            fscanf(fpr, "%s", str);
        }
        else if(i > 0)
        {
            fscanf(fpr, "%s", str); /* Read Record NO. */
            fscanf(fpr, "%d", &PickupClinic[i]);
            fscanf(fpr, "%d", &DeliveryClinic[i]);
            fscanf(fpr, "%lf", &ReadyTime[i]);
            fscanf(fpr, "%d", &Demand[i]);
        }
    }


    /* Assign Demand to Delivery Node */
    for(i=Records+1; i<=2*Records; i++)
    {
        Demand[i] = -Demand[i-Records];
    }

    /* Assign Distance */
    for(i=0; i<=2*Records; i++)
    {
        for(j=0; j<=2*Records; j++)
        {
            if(i==0 && j==0)
            {
                Distance[i][j] = 0;
            }
            else if(i!=0 && j==0)
            {
                if(i <= Records)
                {
                    Distance[i][j] = STDistance[PickupClinic[i]][0];
                }
                else if(i > Records)
                {
                    Distance[i][j] = STDistance[DeliveryClinic[i-Records]][0];
                }
            }
            else if(i==0 && j!=0)
            {
                if(j <= Records)
                {
                    Distance[i][j] = STDistance[0][PickupClinic[j]];
                }
                else if(j > Records)
                {
                    Distance[i][j] = STDistance[0][DeliveryClinic[j-Records]];
                }
            }
            else if(i!=0 && j!=0)
            {
                if(i <= Records && j <= Records)
                {
                    Distance[i][j] = STDistance[PickupClinic[i]][PickupClinic[j]];
                }
                else if(i <= Records && j > Records)
                {
                    Distance[i][j] = STDistance[PickupClinic[i]][DeliveryClinic[j-
Records]];
                }
```

```
            else if(i > Records && j <= Records)
            {
               Distance[i][j] = STDistance[DeliveryClinic[i-
Records]][PickupClinic[j]];
            }
            else if(i > Records && j > Records)
            {
               Distance[i][j] = STDistance[DeliveryClinic[i-
Records]][DeliveryClinic[j-Records]];
            }
         }
      }
   }


   /* Calculate Travel Time */
   for(i = 0; i <= 2*Records; i++)
   {
      for(j = 0; j <= 2*Records; j++)
      {
         TravelTime[i][j] = ConstantTime * Distance[i][j];
      }
   }

   fclose(fpr);
}


/*************************************************************/
/* Function : SetInitialRoutes()
/* - Set Initial Routes
/*
/*************************************************************/
void SetInitialRoutes()
{
   int r;

   for(r=1; r<=Records; r++)
   {
      Route[r][0] = 0;
      Route[r][1] = 0;
   }
}


/*************************************************************/
/* Function : ConstructInitialSolution_SolomonInsertion()
/* - Construct Initial Solution by Solomon Insertion Heuristic.
/*
/*************************************************************/
int ConstructInitialSolution_SolomonInsertion()
{
   int n,r,i;
   int IT,p,pd;

   int NoPositions,OriginPosition,DestinationPosition;

   int FeasibleRoute, FeasibleInsertforanyRecord;

   double CostOldRoute, CostNewRoute, InsertionCost;
   double TravelTimeOldRoute, TravelTimeNewRoute;
   double CostofConstructionNewRoute, SavingCost, MaximumSavingCost;
   int Prod, BestRecordtoInsert;
```

```
   /* Set Initial Record List for Construct Initial Solution by Solomon's
 Insertion Heuristic */
   for(i=1; i<=Records; i++)
   {
      PDList[i] = 1;
   }

   /* Clear Variables */
   for(i=1; i<=Records; i++)
   {
      MinimumInsertionCostofRecord[i] = 0;
      BestOriginPositionofRecord[i] = 0;
      BestDestinationPositionofRecord[i] = 0;
      FeasibleInsertionPlace[i] = 0;
   }


   /* InitializationCriteria is Earliest Ready Time */
   Prod = FindUnroutedRecordwithEarliestReadyTime(PDList);

   r = 1;
   /* Initial First Route with First Record */
   Route[r][0] = 0;
   Route[r][1] = Prod; /* Go to Pickup Point */
   Route[r][2] = Records+Prod; /* Go to Delivery Point */
   Route[r][3] = 0; /* Go back to Hub */

   /* Check Feasible Route */
   FeasibleRoute = CheckFeasibleRoute(Route[r]);
   if(FeasibleRoute == FALSE)
   {
      printf("Infeasible Route - Cannot construct Route[%d]!\n", r);
      PrintOneRoute(Route[r], r);
      exit(1);
   }

   /* Set First Record to 0 */
   PDList[Prod] = 0;


   /* Solomon Insertion Heuristic */
   for(IT=1; IT<=Records-1; IT++)
   {
      /* Calculate Insertion Cost */
      FeasibleInsertforanyRecord = NOTFOUND;

      for(p=1; p<=Records; p++)
      {
         n=0;
         FeasibleInsertionPlace[p] = NOTFOUND;
         if(PDList[p] == 1)
         {
            i =  CountNORecordsinRoute(Route[r]);
            NoPositions = 2*i+1;

            for(OriginPosition = 1; OriginPosition <= NoPositions;
OriginPosition++)
            {
               for(DestinationPosition = OriginPosition+1; DestinationPosition
<= NoPositions+1; DestinationPosition++)
               {
                  ClearRoute(OneTR1);
```

```
                CopyOneRoute(Route[r], OneTR1); /* Route for Before Insert
Record */

                InsertRecord(p, OneTR1, OriginPosition,
DestinationPosition);

                FeasibleRoute = CheckFeasibleRoute(OneTR1);


                if(FeasibleRoute == TRUE)
                {
                   FeasibleInsertforanyRecord = FOUND;

                   // distance
                   CostOldRoute = EvaluateCostOneRoute(Route[r]);
                   CostNewRoute = EvaluateCostOneRoute(OneTR1);
                   InsertionCost = CostNewRoute - CostOldRoute;

                   n = n+1;

                   if(n==1) /*first position of record p*/
                   {
                      MinimumInsertionCostofRecord[p] = InsertionCost;
                      BestOriginPositionofRecord[p] = OriginPosition;
                      BestDestinationPositionofRecord[p] =
DestinationPosition;
                      FeasibleInsertionPlace[p] = FOUND;
                   }
                   else if(n > 1)
                   {
                      if(InsertionCost < MinimumInsertionCostofRecord[p])
                      {
                         MinimumInsertionCostofRecord[p] = InsertionCost;
                         BestOriginPositionofRecord[p] = OriginPosition;
                         BestDestinationPositionofRecord[p] =
DestinationPosition;
                      }
                   }
                }
             }
          }
       }
    }


    /* Calculate Saving Cost when construct new Route */

    /* There is Feasible insert solution for any unroutes Records */
    if(FeasibleInsertforanyRecord == FOUND)
    {
       n=0;
       for(pd=1; pd<=Records; pd++)
       {
          if(PDList[pd] == 1 && FeasibleInsertionPlace[pd] == FOUND)
          {
             n = n+1;
             if(n==1)
             {
                CostofConstructionNewRoute =
CalculateCostofConstructionNewRoute(pd);
                SavingCost = CostofConstructionNewRoute -
MinimumInsertionCostofRecord[pd];
                MaximumSavingCost = SavingCost;
```

```
                BestRecordtoInsert = pd;
              }
              else if(n > 1)
              {
                CostofConstructionNewRoute =
CalculateCostofConstructionNewRoute(pd);
                SavingCost = CostofConstructionNewRoute -
MinimumInsertionCostofRecord[pd];


                if(SavingCost > MaximumSavingCost)
                {
                  MaximumSavingCost = SavingCost;
                  BestRecordtoInsert = pd;
                }
              }
            }
          }

          /* get Best Record to Insert in Route r */
          InsertRecord(BestRecordtoInsert, Route[r],
BestOriginPositionofRecord[BestRecordtoInsert],
BestDestinationPositionofRecord[BestRecordtoInsert]);
          PDList[BestRecordtoInsert] = 0;


        }
        else if(FeasibleInsertforanyRecord == NOTFOUND)
        {
          /* Initialization Criteria is Earliest Ready Time */
          Prod = FindUnroutedRecordwithEarliestReadyTime(PDList);

          r = r+1;
          /* Initial First Route with First Record */
          Route[r][0] = 0;
          Route[r][1] = Prod; /* Go to Pickup Point */
          Route[r][2] = Records+Prod; /* Go to Delivery Point */
          Route[r][3] = 0; /* Go back to Hub */

          /* Check Feasible Route */
          FeasibleRoute = CheckFeasibleRoute(Route[r]);
          if(FeasibleRoute == FALSE)
          {
            printf("Infeasible Route - Cannot construct Route[%d]!\n", r);
            PrintOneRoute(Route[r], r);
            exit(1);
          }

          /* Set First Record to 0 */
          PDList[Prod] = 0;
        }
      }

      return r;
}


/*********************************************************/
/* Function : FindUnroutedRecordwithEarliestReadyTime(int *PDList)
/* - Initialization Criteria is Earliest Ready Time.
/*
/*********************************************************/
int FindUnroutedRecordwithEarliestReadyTime(int *PDList)
{
```

```
   int i,n,Prod;
   double EarliestReadyTime;

   n=0;
   for(i=1; i<=Records; i++)
   {
      if(PDList[i] == 1)
      {
         n = n+1;
         if(n==1)
         {
            EarliestReadyTime = ReadyTime[i];
            Prod = i;
         }
         else if(n>1)
         {
            if(ReadyTime[i] < EarliestReadyTime)
            {
               EarliestReadyTime = ReadyTime[i];
               Prod = i;
            }
         }
      }
   }

   return Prod;
}


/*************************************************************/
/* Function : CalculateCostofConstructionNewRoute(int RC)
/* - Calculate cost of construction initial new route.
/*
/*************************************************************/
double CalculateCostofConstructionNewRoute(int RC)
{
   double Cost;

   Cost = Distance[0][RC] + Distance[RC][RC+Records] +
Distance[RC+Records][0];

   return Cost;
}


/*************************************************************/
/* Function : LocalSearch()
/* - Local Search for better solution.
/*
/*************************************************************/
void LocalSearch()
{
   int R,Vehicles;
   int RC;
   double TC;

   /* PDShift Variables */
   int PSC_Positive_PDShift;
   int PDShift_BestRecord, PDShift_BestRoute1, PDShift_BestRoute2,
PDShift_BestOriginPosition, PDShift_BestDestinationPosition;

   /* PDExchange Variables */
   int PSC_Positive_PDExchange;
```

```
   int PDExchange_Record1, PDExchange_Route1,
PDExchange_BestOriginPosition_P1intoR2,
PDExchange_BestDestinationPosition_P1intoR2;
   int PDExchange_Record2, PDExchange_Route2,
PDExchange_BestOriginPosition_P2intoR1,
PDExchange_BestDestinationPosition_P2intoR1;

   /* WRI Variables */
   int PSC_Positive_WRI, PSC_Positive_WRI_EachRoute;
   int WRI_Node, WRI_Route, WRI_BestPosition;


   /* Reduce Route Variables */
   int ReduceRoute;


   do
   {
      do
      {
         do
         {
            do
            {
               PSC_Positive_PDShift = NOTFOUND;
               RearrangeAllRoutes(Route);
               Vehicles = CountVehicles(Route);

               if(Vehicles > 1)
               {
                  PSC_Positive_PDShift =
FindSolution_HighestPositiveSavingCost_PDShiftOperator(&PDShift_BestRecord,
&PDShift_BestRoute1, &PDShift_BestRoute2, &PDShift_BestOriginPosition,
&PDShift_BestDestinationPosition);

                  if(PSC_Positive_PDShift == FOUND)
                  {
                     PDShiftOperator(PDShift_BestRecord,
Route[PDShift_BestRoute1], Route[PDShift_BestRoute2],
PDShift_BestOriginPosition, PDShift_BestDestinationPosition);
                     RearrangeAllRoutes(Route);
                  }
               }
            }
            while(PSC_Positive_PDShift == FOUND);

            PSC_Positive_PDExchange = NOTFOUND;
            RearrangeAllRoutes(Route);
            Vehicles = CountVehicles(Route);

            if(Vehicles > 1)
            {
               PSC_Positive_PDExchange =
FindSolution_HighestPositiveSavingCost_PDExchangeOperator(&PDExchange_Record
1, &PDExchange_Route1, &PDExchange_BestOriginPosition_P1intoR2,
&PDExchange_BestDestinationPosition_P1intoR2, &PDExchange_Record2,
&PDExchange_Route2, &PDExchange_BestOriginPosition_P2intoR1,
&PDExchange_BestDestinationPosition_P2intoR1);

               if(PSC_Positive_PDExchange == FOUND)
               {
                  PDExchangeOperator(PDExchange_Record1,
Route[PDExchange_Route1], PDExchange_Record2, Route[PDExchange_Route2],
PDExchange_BestOriginPosition_P1intoR2,
```

```
PDExchange_BestDestinationPosition_P1intoR2,
PDExchange_BestOriginPosition_P2intoR1,
PDExchange_BestDestinationPosition_P2intoR1);
                RearrangeAllRoutes(Route);
              }
            }
          }
          while(PSC_Positive_PDExchange == FOUND);

          PSC_Positive_WRI = NOTFOUND;
          RearrangeAllRoutes(Route);
          Vehicles = CountVehicles(Route);

          for(WRI_Route=1; WRI_Route<=Vehicles; WRI_Route++)
          {
             /*WRI - Type 1 */
             PSC_Positive_WRI_EachRoute =
FindSolution_HighestPositiveSavingCost_RearrangeOperator_WithinRouteInsertio
n(&WRI_Node, &WRI_Route, &WRI_BestPosition);

             if(PSC_Positive_WRI_EachRoute == FOUND)
             {
                PSC_Positive_WRI = FOUND;
                RearrangeOperator_WithinRouteInsertion(WRI_Node,
Route[WRI_Route], WRI_BestPosition);
             }
             else
             {
                /*WRI - Swap 3-consecutive positions */
                RC = CountNORecordsinRoute(Route[WRI_Route]);
                if(RC > 1)
                {
                   PSC_Positive_WRI_EachRoute =
FindSolution_HighestPositiveSavingCost_RearrangeOperator_Swap3ConsecutivePos
itionsWithinRoute(Route[WRI_Route]);
                   if(PSC_Positive_WRI_EachRoute == FOUND)
                   {
                      PSC_Positive_WRI = FOUND;
                   }
                }
             }
          }
       }
       while(PSC_Positive_WRI == FOUND);

       ReduceRoute = ReduceRoute_HighestPositiveSavingCost();
    }
    while(ReduceRoute == FOUND);
}


/*************************************************************/
/* Function : FindSolution_HighestPositiveSavingCost_PDShiftOperator(int
*PDShift_BestRecord, int *PDShift_BestRoute1, int *PDShift_BestRoute2, int
*PDShift_BestOriginPosition, int *PDShift_BestDestinationPosition)
/* - Find Pure Saving Cost.
/* - This function for Local Search.
/*
/*************************************************************/
int FindSolution_HighestPositiveSavingCost_PDShiftOperator(int
*PDShift_BestRecord, int *PDShift_BestRoute1, int *PDShift_BestRoute2, int
*PDShift_BestOriginPosition, int *PDShift_BestDestinationPosition)
{
```

```
    int i;
    int V;

    int RandomRoute1, RandomRoute2, RandomRecord1;
    int NPDShift;

    int FeasibleOneTR1,FeasibleOneTR2;

    double DistR1, DistR2, DistR1P, DistR2P;
    double PureSavingCost, BestPureSavingCost;

    int OriginPosition, DestinationPosition;
    int p, NoPositions;

    int PSC_Positive_PDShift;

    int n,N_Shift;


    V = CountVehicles(Route);

    PureSavingCost = 0;
    BestPureSavingCost = 0;

    NPDShift=0;

    PSC_Positive_PDShift = NOTFOUND;

    n=1;
    N_Shift = (int) floor(alpha*Records);

    while(n <= N_Shift)
    {
       /* Random Record1 */
       RandomRecord1 = rand();
       RandomRecord1 = RandomRecord1 % Records;
       RandomRecord1 = RandomRecord1 + 1;

       /* Random Route1 (find Route1 from Record1) */
       RandomRoute1 = FindRoute(RandomRecord1);

       /* Random Route2 */
       do
       {
          RandomRoute2 = rand();
          RandomRoute2 = RandomRoute2 % V;
          RandomRoute2 = RandomRoute2 + 1;
       }
       while(RandomRoute1 == RandomRoute2);


       /* Calculate Best PSC when insert RandomRecord from R1 to R2 */
       p =  CountNORecordsinRoute(Route[RandomRoute2]);
       NoPositions = 2*p+1;

       for(OriginPosition = 1; OriginPosition <= NoPositions;
OriginPosition++)
       {
          for(DestinationPosition = OriginPosition+1; DestinationPosition <=
NoPositions+1; DestinationPosition++)
          {
             ClearRoute(OneTR1);
             ClearRoute(OneTR2);
             ClearRoutes(ManyTR);
```

```
            CopyOneRoute(Route[RandomRoute1], OneTR1);
            CopyOneRoute(Route[RandomRoute2], OneTR2);

            RemoveRecord(RandomRecord1, OneTR1);
            InsertRecord(RandomRecord1, OneTR2, OriginPosition,
DestinationPosition);

            FeasibleOneTR1 = CheckFeasibleRoute(OneTR1);
            FeasibleOneTR2 = CheckFeasibleRoute(OneTR2);

            /* Calculate PSC for Feasible Route */
            if(FeasibleOneTR1 == TRUE && FeasibleOneTR2 == TRUE)
            {
               DistR1 = EvaluateCostOneRoute(Route[RandomRoute1]);
               DistR2 = EvaluateCostOneRoute(Route[RandomRoute2]);
               DistR1P = EvaluateCostOneRoute(OneTR1);
               DistR2P = EvaluateCostOneRoute(OneTR2);

               PureSavingCost = DistR1 + DistR2 - DistR1P - DistR2P;

               /* Keep Best*/
               if(PureSavingCost > 0)
               {
                  NPDShift = NPDShift+1;
                  PSC_Positive_PDShift = FOUND;

                  if(NPDShift == 1)
                  {
                     BestPureSavingCost = PureSavingCost;
                     *PDShift_BestOriginPosition = OriginPosition;
                     *PDShift_BestDestinationPosition = DestinationPosition;
                     *PDShift_BestRoute1 = RandomRoute1;
                     *PDShift_BestRoute2 = RandomRoute2;
                     *PDShift_BestRecord = RandomRecord1;
                  }
                  else if(NPDShift > 1)
                  {
                     if(PureSavingCost > BestPureSavingCost)
                     {
                        BestPureSavingCost = PureSavingCost;
                        *PDShift_BestOriginPosition = OriginPosition;
                        *PDShift_BestDestinationPosition = DestinationPosition;
                        *PDShift_BestRoute1 = RandomRoute1;
                        *PDShift_BestRoute2 = RandomRoute2;
                        *PDShift_BestRecord = RandomRecord1;
                     }
                  }
               }
            }
         }
      }
      n = n+1;
   }

   return PSC_Positive_PDShift;
}


/*************************************************************/
/* Function :
FindSolution_HighestPositiveSavingCost_PDExchangeOperator_AllPairs(int
*PDExchange_Record1, int *PDExchange_Route1, int
```

```
*PDExchange_BestOriginPosition_P1intoR2, int
*PDExchange_BestDestinationPosition_P1intoR2, int *PDExchange_Record2, int
*PDExchange_Route2, int *PDExchange_BestOriginPosition_P2intoR1, int
*PDExchange_BestDestinationPosition_P2intoR1)
/* - Find Pure Saving Cost.
/* - This function for Local Search.
/*
/*************************************************************/
int FindSolution_HighestPositiveSavingCost_PDExchangeOperator(int
*PDExchange_Record1, int *PDExchange_Route1, int
*PDExchange_BestOriginPosition_P1intoR2, int
*PDExchange_BestDestinationPosition_P1intoR2, int *PDExchange_Record2, int
*PDExchange_Route2, int *PDExchange_BestOriginPosition_P2intoR1, int
*PDExchange_BestDestinationPosition_P2intoR1)
{
   int i,j;

   int FeasibleOneTR1, FeasibleOneTR2;
   int V,p;

   double BestPureSavingCost, PureSavingCost;
   double DistR1, DistR2, DistR1P, DistR2P;

   int RandomRoute1, RandomRoute2, RandomRecord1, RandomRecord2;

   int OriginPosition_P1intoR2, DestinationPosition_P1intoR2,
NoPositions_P1intoR2;
   int OriginPosition_P2intoR1, DestinationPosition_P2intoR1,
NoPositions_P2intoR1;
   int PSC_Positive_PDExchange, NPDExchange;

   int n,N_Exchange;


   RearrangeAllRoutes(Route);
   V = CountVehicles(Route);

   PureSavingCost = 0;
   BestPureSavingCost = 0;

   NPDExchange = 0;

      PSC_Positive_PDExchange = NOTFOUND;

   n=1;
   N_Exchange = (int) floor(beta*Records);

   while(n <= N_Exchange)
   {
      /* Random Record1 */
      RandomRecord1 = rand();
      RandomRecord1 = RandomRecord1 % Records;
      RandomRecord1 = RandomRecord1 + 1;

      /* Random Route1 (find Route1 from Record1) */
      RandomRoute1 = FindRoute(RandomRecord1);

      /* Random Route2 */
      do
      {
         /* Random Record2 */
         RandomRecord2 = rand();
         RandomRecord2 = RandomRecord2 % Records;
         RandomRecord2 = RandomRecord2 + 1;
```

```
            /* Random Route2 (find Route2 from Record2) */
            RandomRoute2 = FindRoute(RandomRecord2);
         }
         while(RandomRoute1 == RandomRoute2);


         /* Calculate Best PSC when insert RandomRecord from R1 to R2 */
         p =  CountNORecordsinRoute(Route[RandomRoute2]);
         NoPositions_P1intoR2 = 2*(p-1)+1;

         p =  CountNORecordsinRoute(Route[RandomRoute1]);
         NoPositions_P2intoR1 = 2*(p-1)+1;

         /********** Insert Record1 into Route2 *****************/
         for(OriginPosition_P1intoR2 = 1; OriginPosition_P1intoR2 <=
NoPositions_P1intoR2; OriginPosition_P1intoR2++)
         {
            for(DestinationPosition_P1intoR2 = OriginPosition_P1intoR2+1;
DestinationPosition_P1intoR2 <= NoPositions_P1intoR2+1;
DestinationPosition_P1intoR2++)
            {
               ClearRoute(OneTR2);
               CopyOneRoute(Route[RandomRoute2], OneTR2);

               RemoveRecord(RandomRecord2, OneTR2);
               InsertRecord(RandomRecord1, OneTR2, OriginPosition_P1intoR2,
DestinationPosition_P1intoR2);

               FeasibleOneTR2 = CheckFeasibleRoute(OneTR2);

               if(FeasibleOneTR2 == TRUE)
               {
                  /****** Insert Record2 into Route1 *********/
                  for(OriginPosition_P2intoR1 = 1; OriginPosition_P2intoR1 <=
NoPositions_P2intoR1; OriginPosition_P2intoR1++)
                  {
                     for(DestinationPosition_P2intoR1 =
OriginPosition_P2intoR1+1; DestinationPosition_P2intoR1 <=
NoPositions_P2intoR1+1; DestinationPosition_P2intoR1++)
                     {
                        ClearRoute(OneTR1);

                        CopyOneRoute(Route[RandomRoute1], OneTR1);

                        RemoveRecord(RandomRecord1, OneTR1);
                        InsertRecord(RandomRecord2, OneTR1,
OriginPosition_P2intoR1, DestinationPosition_P2intoR1);

                        FeasibleOneTR1 = CheckFeasibleRoute(OneTR1);

                        /* Check Feasible Route */
                        if(FeasibleOneTR1 == TRUE)
                        {
                           DistR1 = EvaluateCostOneRoute(Route[RandomRoute1]);
                           DistR2 = EvaluateCostOneRoute(Route[RandomRoute2]);
                           DistR1P = EvaluateCostOneRoute(OneTR1);
                           DistR2P = EvaluateCostOneRoute(OneTR2);

                           PureSavingCost = DistR1 + DistR2 - DistR1P - DistR2P;

                           if(PureSavingCost > 0)
                           {
                              NPDExchange = NPDExchange+1;
```

```
                        PSC_Positive_PDExchange = FOUND;

                        if(NPDExchange == 1)
                        {
                           BestPureSavingCost = PureSavingCost;
                           *PDExchange_Route1 = RandomRoute1;
                           *PDExchange_Route2 = RandomRoute2;
                           *PDExchange_Record1 = RandomRecord1;
                           *PDExchange_Record2 = RandomRecord2;
                           *PDExchange_BestOriginPosition_P1intoR2      =
OriginPosition_P1intoR2;
                           *PDExchange_BestDestinationPosition_P1intoR2 =
DestinationPosition_P1intoR2;
                           *PDExchange_BestOriginPosition_P2intoR1      =
OriginPosition_P2intoR1;
                           *PDExchange_BestDestinationPosition_P2intoR1 =
DestinationPosition_P2intoR1;
                        }
                        else if(NPDExchange > 1)
                        {
                           if(PureSavingCost > BestPureSavingCost)
                           {
                              BestPureSavingCost = PureSavingCost;
                              *PDExchange_Route1 = RandomRoute1;
                              *PDExchange_Route2 = RandomRoute2;
                              *PDExchange_Record1 = RandomRecord1;
                              *PDExchange_Record2 = RandomRecord2;
                              *PDExchange_BestOriginPosition_P1intoR2      =
OriginPosition_P1intoR2;
                              *PDExchange_BestDestinationPosition_P1intoR2 =
DestinationPosition_P1intoR2;
                              *PDExchange_BestOriginPosition_P2intoR1      =
OriginPosition_P2intoR1;
                              *PDExchange_BestDestinationPosition_P2intoR1 =
DestinationPosition_P2intoR1;
                           }
                        }
                     }
                  }
               }
            }
         }
      }
      n = n+1;
   }

   return PSC_Positive_PDExchange;
}


/*********************************************************/
/* Function :
FindSolution_HighestPositiveSavingCost_RearrangeOperator_WithinRouteInsertio
n(int *WRI_Node, int *WRI_Route, int *WRI_BestPosition)
/* - Within Route Insertion.
/* - This function for Local Search.
/*
/*********************************************************/
int
FindSolution_HighestPositiveSavingCost_RearrangeOperator_WithinRouteInsertio
n(int *WRI_Node, int *WRI_Route, int *WRI_BestPosition)
{
   int r,rn;
```

```
int s,t,i,p,pr;

int FeasibleTRP;

double DistR, DistTRP;
double PureSavingCost, BestPureSavingCost;

int Position, FirstPositionToInsert, LastPositionToInsert;

int NoPositions;

int NWRI;
int PSC_Positive_WRI_EachRoute;


PureSavingCost = 0;
BestPureSavingCost = 0;


ClearRoute(OneTR1);
CopyOneRoute(Route[*WRI_Route], OneTR1);

p = CountNORecordsinRoute(Route[*WRI_Route]);


/* Random Position in Route */
r = 2*p;

rn = rand();
rn = rn % r;
Position = rn+1;
*WRI_Node = OneTR1[Position];

PSC_Positive_WRI_EachRoute = NOTFOUND;
NWRI = 0;


if(*WRI_Node <= Records)      /* Node is Predecessor */
{
   /* Remove Node */
   RemoveNode(*WRI_Node, OneTR1);

   FirstPositionToInsert = 1;

   /* Find LastPositionToInsert */
   s = 0;
   do
   {
      if(OneTR1[s] == *WRI_Node+Records)
      {
         LastPositionToInsert = s;
         break;
      }

      s = s+1;
   }
   while(OneTR1[s] != 0);


   /* Insert Predecessor until First to Last Position */
   for(i=FirstPositionToInsert; i<=LastPositionToInsert; i++)
   {
      ClearRoute(OneTR2);
      ClearRoutes(ManyTR);
```

```
            CopyOneRoute(OneTR1, OneTR2);
            CopyAllRoutes(Route, ManyTR);

            InsertNode(*WRI_Node, OneTR2, i);

            FeasibleTRP = CheckFeasibleRoute(OneTR2);

            CopyOneRoute(OneTR2, ManyTR[*WRI_Route]);

            /* Calculate PSC for Feasible TR */
            if(FeasibleTRP == TRUE)
            {
               DistR = EvaluateCostOneRoute(Route[*WRI_Route]);
               DistTRP = EvaluateCostOneRoute(OneTR2);

               PureSavingCost = DistR - DistTRP;

               if(PureSavingCost > 0)
               {
                  NWRI = NWRI+1;
                  PSC_Positive_WRI_EachRoute = FOUND;

                  if(NWRI == 1)
                  {
                     BestPureSavingCost = PureSavingCost;
                     *WRI_BestPosition = i;
                  }
                  else if(NWRI > 1)
                  {
                     if(PureSavingCost > BestPureSavingCost)
                     {
                        PSC_Positive_WRI_EachRoute = FOUND;
                        BestPureSavingCost = PureSavingCost;
                        *WRI_BestPosition = i;
                     }
                  }
               }
            }
         }
      }
      else if(*WRI_Node > Records) /* Node is Successor */
      {
         /* Remove Node */
         RemoveNode(*WRI_Node, OneTR1);

         /* Find FirstPositionToInsert */
         s = 0;
         do
         {
            if(OneTR1[s] == *WRI_Node-Records)
            {
               FirstPositionToInsert = s+1;
            }

            s = s+1;
         }
         while(OneTR1[s] != 0);
         LastPositionToInsert = s;

         /* Insert Predecessor until First to Last Position */
         for(i=FirstPositionToInsert; i<=LastPositionToInsert; i++)
         {
            ClearRoute(OneTR2);
```

```
            ClearRoutes(ManyTR);

            CopyOneRoute(OneTR1, OneTR2);
            CopyAllRoutes(Route, ManyTR);

            InsertNode(*WRI_Node, OneTR2, i);

            FeasibleTRP = CheckFeasibleRoute(OneTR2);

            CopyOneRoute(OneTR2, ManyTR[*WRI_Route]);

            /* Calculate PSC for Feasible TR */
            if(FeasibleTRP == TRUE)
            {
               DistR = EvaluateCostOneRoute(Route[*WRI_Route]);
               DistTRP = EvaluateCostOneRoute(OneTR2);

               PureSavingCost = DistR - DistTRP;

               if(PureSavingCost > 0)
               {
                  NWRI = NWRI+1;
                  PSC_Positive_WRI_EachRoute = FOUND;

                  if(NWRI == 1)
                  {
                     BestPureSavingCost = PureSavingCost;
                     *WRI_BestPosition = i;
                  }
                  else if(NWRI > 1)
                  {
                     if(PureSavingCost > BestPureSavingCost)
                     {
                        PSC_Positive_WRI_EachRoute = FOUND;
                        BestPureSavingCost = PureSavingCost;
                        *WRI_BestPosition = i;
                     }
                  }
               }
            }
         }
      }

      return PSC_Positive_WRI_EachRoute;
   }


   /*************************************************************/
   /* Function :
   FindSolution_HighestPositiveSavingCost_RearrangeOperator_Swap3ConsecutivePos
   itionsWithinRoute(int *OneRoute)
   /* - Swap 3-consecutive positions.
   /* - This function for Local Search.
   /*
   /*************************************************************/
   int
   FindSolution_HighestPositiveSavingCost_RearrangeOperator_Swap3ConsecutivePos
   itionsWithinRoute(int *OneRoute)
   {
      int i,nb=0;
      int RC,Position;
      int temp;
      int Feasible;
```

```
int PSC_Positive_WRI_EachRoute;

double TempCost, Cost;

PSC_Positive_WRI_EachRoute = NOTFOUND;

RC = CountNORecordsinRoute(OneRoute);
Cost = EvaluateCostOneRoute(OneRoute);

/* Position until swap 3 consecutive-RC */
Position = 2*RC - 2;

for(i=1; i<=Position; i++)
{
   /* Type 2 - 1 3 2 */
   CopyOneRoute(OneRoute, OneTR1);
   temp = OneTR1[i+2];
   OneTR1[i+2] = OneTR1[i+1];
   OneTR1[i+1] = temp;

   Feasible = CheckFeasibleRoute(OneTR1);

   if(Feasible == TRUE)
   {
      TempCost = EvaluateCostOneRoute(OneTR1);
      if(TempCost < Cost)
      {
         PSC_Positive_WRI_EachRoute = FOUND;
         Cost = TempCost;
         CopyOneRoute(OneTR1, OneTR2);
      }
   }

   /* Type 3 - 2 1 3 */
   CopyOneRoute(OneRoute, OneTR1);
   temp = OneTR1[i+1];
   OneTR1[i+1] = OneTR1[i];
   OneTR1[i] = temp;

   Feasible = CheckFeasibleRoute(OneTR1);

   if(Feasible == TRUE)
   {
      TempCost = EvaluateCostOneRoute(OneTR1);
      if(TempCost < Cost)
      {
         PSC_Positive_WRI_EachRoute = FOUND;
         Cost = TempCost;
         CopyOneRoute(OneTR1, OneTR2);
      }
   }

   /* Type 4 - 2 3 1 */
   CopyOneRoute(OneRoute, OneTR1);
   temp = OneTR1[i];
   OneTR1[i] = OneTR1[i+1];
   OneTR1[i+1] = OneTR1[i+2];
   OneTR1[i+2] = temp;

   Feasible = CheckFeasibleRoute(OneTR1);

   if(Feasible == TRUE)
   {
      TempCost = EvaluateCostOneRoute(OneTR1);
```

```
            if(TempCost < Cost)
            {
               PSC_Positive_WRI_EachRoute = FOUND;
               Cost = TempCost;
               CopyOneRoute(OneTR1, OneTR2);
            }
         }

         /* Type 5 - 3 1 2 */
         CopyOneRoute(OneRoute, OneTR1);
         temp = OneTR1[i+2];
         OneTR1[i+2] = OneTR1[i+1];
         OneTR1[i+1] = OneTR1[i];
         OneTR1[i] = temp;

         Feasible = CheckFeasibleRoute(OneTR1);

         if(Feasible == TRUE)
         {
            TempCost = EvaluateCostOneRoute(OneTR1);
            if(TempCost < Cost)
            {
               PSC_Positive_WRI_EachRoute = FOUND;
               Cost = TempCost;
               CopyOneRoute(OneTR1, OneTR2);
            }
         }

         /* Type 6 - 3 2 1 */
         CopyOneRoute(OneRoute, OneTR1);
         temp = OneTR1[i];
         OneTR1[i] = OneTR1[i+2];
         OneTR1[i+2] = temp;

         Feasible = CheckFeasibleRoute(OneTR1);

         if(Feasible == TRUE)
         {
            TempCost = EvaluateCostOneRoute(OneTR1);
            if(TempCost < Cost)
            {
               PSC_Positive_WRI_EachRoute = FOUND;
               Cost = TempCost;
               CopyOneRoute(OneTR1, OneTR2);
            }
         }
      }

      /* Copy Back */
      if(PSC_Positive_WRI_EachRoute == FOUND)
      {
         CopyOneRoute(OneTR2, OneRoute);
      }

      return PSC_Positive_WRI_EachRoute;
   }


   /*************************************************************/
   /* Function : ReduceRoute_HighestPositiveSavingCost()
   /* - Reduce Route.
   /* - This function for Local Search.
   /*
   /*************************************************************/
```

```
int ReduceRoute_HighestPositiveSavingCost()
{
   int n,s,i,j;
   int p,P;
   int r,r1,r2,rinsert;
   int VR,VTR;

   int NoPositions, OriginPosition, DestinationPosition;
   int BestOriginPosition, BestDestinationPosition;

   int FeasibleTR, FeasibleInsertPosition;
   double DistTR,TotalDistTR,LeastDistance;

   double BestTC, TC;

   int ReduceRoute;


   ReduceRoute = NOTFOUND;

   BestTC = 0;
   TC = 0;

   /* No. of Routes before reduce route */
   VR = CountVehicles(Route);

   /* Allocate TempRoute */
   ClearRoutes(ManyTR);
   ClearRoutes(ManyBTR);

   /* Clear RecordList */
   for(i=0; i<=Records; i++)
   {
      RecordList[i] = 0;
   }

   for(r=1; r<=VR; r++)
   {
      CopyAllRoutes(Route, ManyTR);

      P = CountNORecordsinRoute(ManyTR[r]);

      s = 1;
      p = 0;

      while(ManyTR[r][s] != 0)
      {
         if(ManyTR[r][s] <= Records)
         {
            p = p+1;
            RecordList[p] = ManyTR[r][s];
         }
         s = s+1;
      }


      /* set route */
      for(s=0; s<= 2*Records+1; s++)
      {
         ManyTR[r][s] = 0;
      }

      RearrangeAllRoutes(ManyTR);
```

```
      /* No. of Routes of ManyTR */
      VTR = CountVehicles(ManyTR);

      /* insert route */
      for(P=1; P<=p; P++)
      {
         FeasibleInsertPosition = NOTFOUND;
         n = 0;

         for(r1=1; r1<=VTR; r1++)
         {
            i =  CountNORecordsinRoute(ManyTR[r1]);
            NoPositions = 2*i+1;

            for(OriginPosition = 1; OriginPosition<=NoPositions;
OriginPosition++)
            {
               for(DestinationPosition = OriginPosition+1;
DestinationPosition<=NoPositions+1; DestinationPosition++)
               {
                  ClearRoute(OneTR1);

                  CopyOneRoute(ManyTR[r1], OneTR1);

                  InsertRecord(RecordList[P], OneTR1, OriginPosition,
DestinationPosition);

                  FeasibleTR = CheckFeasibleRoute(OneTR1);

                  if(FeasibleTR == TRUE)
                  {
                     FeasibleInsertPosition = FOUND;
                     n = n+1;

                     if(n==1)
                     {
                        TotalDistTR = 0;
                        for(r2=1; r2<=VTR; r2++)
                        {
                           if(r2 != r1)
                           {
                              DistTR = EvaluateCostOneRoute(ManyTR[r2]);
                           }
                           else if(r2 == r1)
                           {
                              DistTR = EvaluateCostOneRoute(OneTR1);
                           }

                           TotalDistTR = TotalDistTR + DistTR;
                        }

                        LeastDistance = TotalDistTR;
                        BestOriginPosition = OriginPosition;
                        BestDestinationPosition = DestinationPosition;
                        rinsert = r1;
                     }
                     else if(n>1)
                     {
                        TotalDistTR = 0;
                        for(r2=1; r2<=VTR; r2++)
                        {
                           if(r2 != r1)
                           {
                              DistTR = EvaluateCostOneRoute(ManyTR[r2]);
```

```
                        }
                        else if(r2 == r1)
                        {
                           DistTR = EvaluateCostOneRoute(OneTR1);
                        }

                        TotalDistTR = TotalDistTR + DistTR;
                     }

                     if(TotalDistTR < LeastDistance)
                     {
                        LeastDistance = TotalDistTR;
                        BestOriginPosition = OriginPosition;
                        BestDestinationPosition = DestinationPosition;
                        rinsert = r1;
                     }
                  }
               }
            }
         }

         /* Insert P in suitable Position */
         if(FeasibleInsertPosition == FOUND)
         {
            InsertRecord(RecordList[P], ManyTR[rinsert], BestOriginPosition,
BestDestinationPosition);
         }
         else if(FeasibleInsertPosition == NOTFOUND)
         {
            VTR = VTR+1;

            ManyTR[VTR][0] = 0;
            ManyTR[VTR][1] = RecordList[P]; /* Go to Pickup Point */
            ManyTR[VTR][2] = Records+RecordList[P]; /* Go to Delivery Point */
            ManyTR[VTR][3] = 0; /* Go back to Hub */

         }
      }

      /* Keep Best */
      if(r == 1)
      {
         TC = EvaluateCostAllRoutes(ManyTR);
         BestTC = TC;
         CopyAllRoutes(ManyTR, ManyBTR);
      }
      else if(r>1)
      {
         TC = EvaluateCostAllRoutes(ManyTR);
         if(TC < BestTC)
         {
            BestTC = TC;
            CopyAllRoutes(ManyTR, ManyBTR);
         }
      }
   }

   /* Compare between BTR and Route */
   TC = EvaluateCostAllRoutes(Route);
   if(BestTC < TC)
   {
      CopyAllRoutes(ManyBTR, Route);
      ReduceRoute = FOUND;
```

```
    }

    return ReduceRoute;
}


/************************************************************/
/* Function : TabuSearch()
/* - Tabu Search.
/*
/************************************************************/
void TabuSearch()
{
    int Route1, Route2;
    int P, V, Vehicles;

    int BestNeighbor;

    double TC, TotalCost;

    int CFS;

    /* WRI Variables */
    int PSC_Positive_WRI, PSC_Positive_WRI_EachRoute;
    int WRI_Node, WRI_Route, WRI_BestPosition;

    /* Reduce Route Variables */
    int ReduceRoute;


    TC = 0;
    Iteration = 1;



    /* Step 1 - Initial Local Search */
    LocalSearch();

    /* Update Best Solution */
    TC = EvaluateCostAllRoutes(Route);
    if(TC < BestTotalCost)
    {
       CopyAllRoutes(Route, BestRoute);
       BestTotalCost = TC;
    }

    printf("\n\nRoute after Initial Local Search = %.2lf\n", TC);
    PrintAllRoutes(Route);
    printf("\nTotal Cost = %.2lf\n", TC);

    /* Step 2 - Loop */
    BestIteration = 0;
    while(Iteration <= MaxIteration)
    {

       V = CountVehicles(Route);
       if(V > 1)
       {
          /* Step 2.1 */
          Route1 = 0;
          Route2 = 0;
          BestNeighbor = ConstructandPickupBestNeighborSolution(&Route1,
&Route2);
```

```
            if(BestNeighbor == FOUND)
            {
               /* Update Tabu List */
               UpdateTabuList(Route);

               /* Step 2.2 */
               P = CountNORecordsinRoute(Route[Route1]);
               if(P > 0)
               {
                  RearrangeRoute_MinimumWaitingTime(Route[Route1]);
               }

               P = CountNORecordsinRoute(Route[Route2]);
               if(P > 0)
               {
                  RearrangeRoute_MinimumWaitingTime(Route[Route2]);
               }

               /* Step 2.3 */
               RearrangeAllRoutes(Route);

               /* Step 2.4 */
               LocalSearch();
            }
         }
         else if(V==1)
         {
            ConstructandPickupBestBeighborOneRoute(Route[1]);
            UpdateTabuList(Route);
         }

         /* Step 2.5 */
         TC = EvaluateCostAllRoutes(Route);

         if(TC < BestTotalCost)
         {
            CopyAllRoutes(Route, BestRoute);
            BestTotalCost = TC;
         }

         printf("\n***** Iteration = %d *****\n", Iteration);

         TotalCost = EvaluateCostAllRoutes(Route);
         printf("\nTotal Cost = %.2lf\n", TotalCost);

         Iteration = Iteration+1;
      }
   }


/*************************************************************/
/* Function : ConstructandPickupBestNeighborSolution(int *Route1, int
*Route2)
/* - Choose Best Neighbor where is no in Tabu List.
/*
/*************************************************************/
int ConstructandPickupBestNeighborSolution(int *Route1, int *Route2)
{
   double TC,TTT,TWT;
   int BestNeighbor;
```

```
   /* PDShift Variables */
   int BestNeighborPDShift;
   int PDShift_BestRecord, PDShift_BestRoute1, PDShift_BestRoute2,
PDShift_BestOriginPosition, PDShift_BestDestinationPosition;
   double PDShift_BestTotalCost;

   /* PDExchange Variables */
   int BestNeighborPDExchange;
   int PDExchange_BestRecord1, PDExchange_BestRoute1,
PDExchange_BestOriginPosition_P1intoR2,
PDExchange_BestDestinationPosition_P1intoR2;
   int PDExchange_BestRecord2, PDExchange_BestRoute2,
PDExchange_BestOriginPosition_P2intoR1,
PDExchange_BestDestinationPosition_P2intoR1;
   double PDExchange_BestTotalCost;


   BestNeighbor = NOTFOUND;


   BestNeighborPDShift =
FindNeighbor_MinimumCost_PDShiftOperator(&PDShift_BestRecord,
&PDShift_BestRoute1, &PDShift_BestRoute2, &PDShift_BestOriginPosition,
&PDShift_BestDestinationPosition, &PDShift_BestTotalCost);
   BestNeighborPDExchange =
FindNeighbor_MinimumCost_PDExchangeOperator(&PDExchange_BestRecord1,
&PDExchange_BestRoute1, &PDExchange_BestOriginPosition_P1intoR2,
&PDExchange_BestDestinationPosition_P1intoR2, &PDExchange_BestRecord2,
&PDExchange_BestRoute2, &PDExchange_BestOriginPosition_P2intoR1,
&PDExchange_BestDestinationPosition_P2intoR1, &PDExchange_BestTotalCost);


   printf("BestNeighborPDShift = %d\n", BestNeighborPDShift);
   printf("BestNeighborPDExchange = %d\n", BestNeighborPDExchange);


   /* Find Best Neighbor between PDShift and PDExchange Neighbor */
   if(BestNeighborPDShift == FOUND && BestNeighborPDExchange == FOUND)
   {
      if(PDShift_BestTotalCost <= PDExchange_BestTotalCost) /* PDShift Move*/
      {
         PDShiftOperator(PDShift_BestRecord, Route[PDShift_BestRoute1],
Route[PDShift_BestRoute2], PDShift_BestOriginPosition,
PDShift_BestDestinationPosition);
         *Route1 = PDShift_BestRoute1;
         *Route2 = PDShift_BestRoute2;
         RearrangeAllRoutes(Route);
         BestNeighbor = FOUND;
      }
      else /* PDExchange Move */
      {
         PDExchangeOperator(PDExchange_BestRecord1,
Route[PDExchange_BestRoute1], PDExchange_BestRecord2,
Route[PDExchange_BestRoute2], PDExchange_BestOriginPosition_P1intoR2,
PDExchange_BestDestinationPosition_P1intoR2,
PDExchange_BestOriginPosition_P2intoR1,
PDExchange_BestDestinationPosition_P2intoR1);
         *Route1 = PDExchange_BestRoute1;
         *Route2 = PDExchange_BestRoute2;
         BestNeighbor = FOUND;
      }
   }
```

```
   else if(BestNeighborPDShift == FOUND && BestNeighborPDExchange ==
NOTFOUND) /* NB PDShift */
   {
      PDShiftOperator(PDShift_BestRecord, Route[PDShift_BestRoute1],
Route[PDShift_BestRoute2], PDShift_BestOriginPosition,
PDShift_BestDestinationPosition);
      *Route1 = PDShift_BestRoute1;
      *Route2 = PDShift_BestRoute2;
      RearrangeAllRoutes(Route);
      BestNeighbor = FOUND;
   }
   else if(BestNeighborPDShift == NOTFOUND && BestNeighborPDExchange ==
FOUND) /* NB PDExchange */
   {
      PDExchangeOperator(PDExchange_BestRecord1,
Route[PDExchange_BestRoute1], PDExchange_BestRecord2,
Route[PDExchange_BestRoute2], PDExchange_BestOriginPosition_P1intoR2,
PDExchange_BestDestinationPosition_P1intoR2,
PDExchange_BestOriginPosition_P2intoR1,
PDExchange_BestDestinationPosition_P2intoR1);
      *Route1 = PDExchange_BestRoute1;
      *Route2 = PDExchange_BestRoute2;
      BestNeighbor = FOUND;
   }


   printf("Neighbor = %d\n", BestNeighbor);


   /* after */
   printf("\n*** Choosen NB ***\n");
   TC = EvaluateCostAllRoutes(Route);
   //PrintAllRoutes(Route);
   printf("Total Cost of Route = %.2lf\n", TC);
   TTT = CalculateTotalTravelTime(Route);
   printf("Total Travel Time = %.2lf\n", TTT);
   TWT = CalculateTotalWaitingTime(Route);
   printf("Total Waiting Time = %.2lf\n\n", TWT);

   return BestNeighbor;
}


/*********************************************************/
/* Function : FindNeighbor_MinimumCost_PDShiftOperator(int **Route)
/* One function of 2 partial functions of
ConstructandPickupBestNeighborSolution.
/*
/*********************************************************/
int FindNeighbor_MinimumCost_PDShiftOperator(int *PDShift_BestRecord, int
*PDShift_BestRoute1, int *PDShift_BestRoute2, int
*PDShift_BestOriginPosition, int *PDShift_BestDestinationPosition, double
*PDShift_BestTotalCost)
{
   int i;
   int V;
   int CSTL;

   int RandomRoute1, RandomRoute2, RandomRecord1;
   int NBPDShift;

   int FeasibleOneTR1,FeasibleOneTR2;

   double TC;
```

```
   int OriginPosition, DestinationPosition;
   int p, NoPositions;
   int BestNeighborPDShift;

   int n,N_Shift;


   V = CountVehicles(Route);

   BestNeighborPDShift = NOTFOUND;
   NBPDShift = 0;

   n=1;
   N_Shift = (int) floor(alpha*Records);

   while(n <= N_Shift)
   {
      /* Random Record1 */
      RandomRecord1 = rand();
      RandomRecord1 = RandomRecord1 % Records;
      RandomRecord1 = RandomRecord1 + 1;

      /* Random Route1 (find Route1 from Record1) */
      RandomRoute1 = FindRoute(RandomRecord1);

      /* Random Route2 */
      do
      {
         RandomRoute2 = rand();
         RandomRoute2 = RandomRoute2 % V;
         RandomRoute2 = RandomRoute2 + 1;
      }
      while(RandomRoute1 == RandomRoute2);


      /* Calculate Best PSC when insert RandomRecord from R1 to R2 */
      p =  CountNORecordsinRoute(Route[RandomRoute2]);
      NoPositions = 2*p+1;

      for(OriginPosition = 1; OriginPosition <= NoPositions;
OriginPosition++)
      {
         for(DestinationPosition = OriginPosition+1; DestinationPosition <=
NoPositions+1; DestinationPosition++)
         {
            ClearRoute(OneTR1);
            ClearRoute(OneTR2);
            ClearRoutes(ManyTR);

            CopyOneRoute(Route[RandomRoute1], OneTR1);
            CopyOneRoute(Route[RandomRoute2], OneTR2);
            CopyAllRoutes(Route, ManyTR);

            RemoveRecord(RandomRecord1, OneTR1);
            InsertRecord(RandomRecord1, OneTR2, OriginPosition,
DestinationPosition);

            FeasibleOneTR1 = CheckFeasibleRoute(OneTR1);
            FeasibleOneTR2 = CheckFeasibleRoute(OneTR2);

            CopyOneRoute(OneTR1, ManyTR[RandomRoute1]);
            CopyOneRoute(OneTR2, ManyTR[RandomRoute2]);
```

```
            CSTL = CheckSolutioninTabuList(ManyTR);

            /* Check Feasible Route */
            if(FeasibleOneTR1 == TRUE && FeasibleOneTR2 == TRUE && CSTL ==
NOTFOUND)
            {
               NBPDShift = NBPDShift+1;
               BestNeighborPDShift = FOUND;

               if(NBPDShift == 1)
               {
                  *PDShift_BestRecord = RandomRecord1;
                  *PDShift_BestRoute1 = RandomRoute1;
                  *PDShift_BestRoute2 = RandomRoute2;
                  *PDShift_BestOriginPosition  = OriginPosition;
                  *PDShift_BestDestinationPosition = DestinationPosition;

                  TC = EvaluateCostAllRoutes(ManyTR);
                  *PDShift_BestTotalCost = TC;
               }
               else if(NBPDShift > 1)
               {
                  TC = EvaluateCostAllRoutes(ManyTR);

                  if(TC < *PDShift_BestTotalCost)
                  {
                     *PDShift_BestRecord = RandomRecord1;
                     *PDShift_BestRoute1 = RandomRoute1;
                     *PDShift_BestRoute2 = RandomRoute2;
                     *PDShift_BestOriginPosition  = OriginPosition;
                     *PDShift_BestDestinationPosition = DestinationPosition;
                     *PDShift_BestTotalCost = TC;
                  }
               }
            }
         }
      }
      n = n+1;
   }

   return BestNeighborPDShift;
}


/*************************************************************/
/* Function : FindNeighbor_MinimumCost_PDExchangeOperator(int
*PDExchange_BestRecord1, int *PDExchange_BestRoute1, int
*PDExchange_BestOriginPosition_P1intoR2, int
*PDExchange_BestDestinationPosition_P1intoR2, int *PDExchange_BestRecord2,
int *PDExchange_BestRoute2, int *PDExchange_BestOriginPosition_P2intoR1, int
*PDExchange_BestDestinationPosition_P2intoR1, double
*PDExchange_BestTotalCost)
/* -One function of 2 partial functions of
ConstructandPickupBestNeighborSolution.
/*
/*************************************************************/
int FindNeighbor_MinimumCost_PDExchangeOperator(int *PDExchange_BestRecord1,
int *PDExchange_BestRoute1, int *PDExchange_BestOriginPosition_P1intoR2, int
*PDExchange_BestDestinationPosition_P1intoR2, int *PDExchange_BestRecord2,
int *PDExchange_BestRoute2, int *PDExchange_BestOriginPosition_P2intoR1, int
*PDExchange_BestDestinationPosition_P2intoR1, double
*PDExchange_BestTotalCost)
{
   int i,j;
```

```
   int FeasibleOneTR1, FeasibleOneTR2;
   int CSTL,V,p;

   int RandomRoute1, RandomRoute2, RandomRecord1, RandomRecord2;

   double TC;
   int OriginPosition_P1intoR2, DestinationPosition_P1intoR2,
NoPositions_P1intoR2;
   int OriginPosition_P2intoR1, DestinationPosition_P2intoR1,
NoPositions_P2intoR1;
   int BestNeighborPDExchange;

   int NBPDExchange;

   int n,N_Exchange;


   RearrangeAllRoutes(Route);
   V = CountVehicles(Route);

   NBPDExchange = 0;
   BestNeighborPDExchange = NOTFOUND;

   n=1;
   N_Exchange = (int) floor(beta*Records);

   while(n <= N_Exchange)
   {
      /* Random Record1 */
      RandomRecord1 = rand();
      RandomRecord1 = RandomRecord1 % Records;
      RandomRecord1 = RandomRecord1 + 1;

      /* Random Route1 (find Route1 from Record1) */
      RandomRoute1 = FindRoute(RandomRecord1);

      /* Random Route2 */
      do
      {
         /* Random Record2 */
         RandomRecord2 = rand();
         RandomRecord2 = RandomRecord2 % Records;
         RandomRecord2 = RandomRecord2 + 1;

         /* Random Route2 (find Route2 from Record2) */
         RandomRoute2 = FindRoute(RandomRecord2);
      }
      while(RandomRoute1 == RandomRoute2);

      /* Calculate Best NB when insert RandomRecord from R1 to R2 */
      p =  CountNORecordsinRoute(Route[RandomRoute2]);
      NoPositions_P1intoR2 = 2*(p-1)+1;

      p =  CountNORecordsinRoute(Route[RandomRoute1]);
      NoPositions_P2intoR1 = 2*(p-1)+1;


      /********** Insert P1 into R2 ***************/
      for(OriginPosition_P1intoR2 = 1; OriginPosition_P1intoR2 <=
NoPositions_P1intoR2; OriginPosition_P1intoR2++)
      {
```

```
        for(DestinationPosition_P1intoR2 = OriginPosition_P1intoR2+1;
DestinationPosition_P1intoR2 <= NoPositions_P1intoR2+1;
DestinationPosition_P1intoR2++)
        {
           ClearRoute(OneTR2);
           ClearRoutes(ManyTR);

           CopyOneRoute(Route[RandomRoute2], OneTR2);
           CopyAllRoutes(Route, ManyTR);

           RemoveRecord(RandomRecord2, OneTR2);
           InsertRecord(RandomRecord1, OneTR2, OriginPosition_P1intoR2,
DestinationPosition_P1intoR2);

           FeasibleOneTR2 = CheckFeasibleRoute(OneTR2);

           CopyOneRoute(OneTR2, ManyTR[RandomRoute2]);

           if(FeasibleOneTR2 == TRUE)
           {
              /*********** Insert P2 to R1 ************/
              for(OriginPosition_P2intoR1 = 1; OriginPosition_P2intoR1 <=
NoPositions_P2intoR1; OriginPosition_P2intoR1++)
              {
                 for(DestinationPosition_P2intoR1 =
OriginPosition_P2intoR1+1; DestinationPosition_P2intoR1 <=
NoPositions_P2intoR1+1; DestinationPosition_P2intoR1++)
                 {
                    ClearRoute(OneTR1);
                    CopyOneRoute(Route[RandomRoute1], OneTR1);

                    RemoveRecord(RandomRecord1, OneTR1);
                    InsertRecord(RandomRecord2, OneTR1,
OriginPosition_P2intoR1, DestinationPosition_P2intoR1);

                    FeasibleOneTR1 = CheckFeasibleRoute(OneTR1);

                    CopyOneRoute(OneTR1, ManyTR[RandomRoute1]);

                    CSTL = CheckSolutioninTabuList(ManyTR);

                    /* Check Feasible Route */
                    if(FeasibleOneTR1 == TRUE && CSTL == NOTFOUND)
                    {
                       NBPDExchange = NBPDExchange+1;
                       BestNeighborPDExchange = FOUND;

                       if(NBPDExchange == 1)
                       {

                          *PDExchange_BestRecord1 = RandomRecord1;
                          *PDExchange_BestRoute1   = RandomRoute1;
                          *PDExchange_BestRecord2 = RandomRecord2;
                          *PDExchange_BestRoute2   = RandomRoute2;
                          *PDExchange_BestOriginPosition_P1intoR2      =
OriginPosition_P1intoR2;
                          *PDExchange_BestDestinationPosition_P1intoR2 =
DestinationPosition_P1intoR2;
                          *PDExchange_BestOriginPosition_P2intoR1      =
OriginPosition_P2intoR1;
                          *PDExchange_BestDestinationPosition_P2intoR1 =
DestinationPosition_P2intoR1;

                          TC = EvaluateCostAllRoutes(ManyTR);
```

```
                        *PDExchange_BestTotalCost = TC;
                     }
                     else if(NBPDExchange > 1)
                     {
                        TC = EvaluateCostAllRoutes(ManyTR);

                        if(TC < *PDExchange_BestTotalCost)
                        {
                           *PDExchange_BestRecord1 = RandomRecord1;
                           *PDExchange_BestRoute1   = RandomRoute1;
                           *PDExchange_BestRecord2 = RandomRecord2;
                           *PDExchange_BestRoute2   = RandomRoute2;
                           *PDExchange_BestOriginPosition_P1intoR2      =
OriginPosition_P1intoR2;
                           *PDExchange_BestDestinationPosition_P1intoR2 =
DestinationPosition_P1intoR2;
                           *PDExchange_BestOriginPosition_P2intoR1      =
OriginPosition_P2intoR1;
                           *PDExchange_BestDestinationPosition_P2intoR1 =
DestinationPosition_P2intoR1;
                           *PDExchange_BestTotalCost = TC;
                        }
                     }
                  }
               }
            }
         }
      }
      n = n+1;
   }

   return BestNeighborPDExchange;
}


/*************************************************************/
/* Function : RearrangeRoute_MinimumWaitingTime(int *RT)
/*
/*************************************************************/
int RearrangeRoute_MinimumWaitingTime(int *RT)
{
   int n,r,i,s;
   int IT,p,pd;

   int NoPositions,OriginPosition,DestinationPosition;

   int FeasibleRoute, FeasibleInsertforanyRecord;


   int RecordinRoute, Prod, BestRecordtoInsert;
   int CaninsertAllRecordsinSameRoute;

   double WaitingTimeOldRoute, WaitingTimeNewRoute, InsertionCost,
MinimumInsertionCost;
   int NORoute;

   ClearRoute(OneTRSolomon);
   CopyOneRoute(RT, OneTRSolomon);

   RecordinRoute = CountNORecordsinRoute(OneTRSolomon);

   /* Clear PDList */
   for(i=1; i<=Records; i++)
```

```
    {
       PDList[i] = 0;
    }


    /*Set PDList for Route */
    s = 1;
    while(OneTRSolomon[s] != 0)
    {
       if(OneTRSolomon[s] <= Records)
       {
          PDList[OneTRSolomon[s]] = 1;
       }
       s = s+1;
    }

    /* Clear Route */
    for(i=0; i<=2*Records+1; i++)
    {
       OneTRSolomon[i] = 0;
    }


    /* Find First Record to insert into Route by Earliest Ready Time */
    Prod = FindUnroutedRecordwithEarliestReadyTime(PDList);

    /* Initial First Route with First Record */
    OneTRSolomon[0] = 0;
    OneTRSolomon[1] = Prod; /* Go to Pickup Point */
    OneTRSolomon[2] = Records+Prod; /* Go to Delivery Point */
    OneTRSolomon[3] = 0; /* Go back to Hub */

    /* Set First Record to 0 */
    PDList[Prod] = 0;

    CaninsertAllRecordsinSameRoute = TRUE;


    /* Solomon Insertion Heuristic by minimal Waiting time */
    for(IT=1; IT<=RecordinRoute-1; IT++)
    {
       FeasibleInsertforanyRecord = NOTFOUND;

       for(p=1; p<=Records; p++)
       {
          n=0;
          FeasibleInsertionPlace[p] = NOTFOUND;

          if(PDList[p] == 1)
          {
             i =  CountNORecordsinRoute(OneTRSolomon);
             NoPositions = 2*i+1;

             for(OriginPosition = 1; OriginPosition <= NoPositions;
OriginPosition++)
             {
                for(DestinationPosition = OriginPosition+1; DestinationPosition
<= NoPositions+1; DestinationPosition++)
                {
                   ClearRoute(OneTR1);
                   CopyOneRoute(OneTRSolomon, OneTR1); /* Route for Insert */

                   InsertRecord(p, OneTR1, OriginPosition,
DestinationPosition);
```

```
                    FeasibleRoute = CheckFeasibleRoute(OneTR1);

                    if(FeasibleRoute == TRUE)
                    {
                       FeasibleInsertforanyRecord = FOUND;


                       WaitingTimeOldRoute =
EvaluateWaitingTimeOneRoute(Route[r]);
                       WaitingTimeNewRoute = EvaluateWaitingTimeOneRoute(OneTR1);

                       InsertionCost = WaitingTimeNewRoute - WaitingTimeOldRoute;

                       n = n+1;

                       if(n==1)
                       {
                          MinimumInsertionCostofRecord[p] = InsertionCost;
                          BestOriginPositionofRecord[p] = OriginPosition;
                          BestDestinationPositionofRecord[p] =
DestinationPosition;
                          FeasibleInsertionPlace[p] = FOUND;
                       }
                       else if(n > 1)
                       {
                          if(InsertionCost < MinimumInsertionCostofRecord[p])
                          {
                             MinimumInsertionCostofRecord[p] = InsertionCost;
                             BestOriginPositionofRecord[p] = OriginPosition;
                             BestDestinationPositionofRecord[p] =
DestinationPosition;
                          }
                       }
                    }
                 }
              }
           }
        }


        /* Choose Best Records that make route has minimum waiting time */
        if(FeasibleInsertforanyRecord == FOUND)
        {
           n=0;
           for(pd=1; pd<=Records; pd++)
           {
              if(PDList[pd] == 1 && FeasibleInsertionPlace[pd] == FOUND)
              {
                 n = n+1;
                 if(n==1)
                 {
                    MinimumInsertionCost = MinimumInsertionCostofRecord[pd];
                    BestRecordtoInsert = pd;
                 }
                 else if(n > 1)
                 {
                    if(MinimumInsertionCostofRecord[pd] < MinimumInsertionCost)
                    {
                       MinimumInsertionCost = MinimumInsertionCostofRecord[pd];
                       BestRecordtoInsert = pd;
                    }
                 }
              }
```

```
        }

        /* get Best Record to Insert in Route r */
        InsertRecord(BestRecordtoInsert, OneTRSolomon,
BestOriginPositionofRecord[BestRecordtoInsert],
BestDestinationPositionofRecord[BestRecordtoInsert]);
        PDList[BestRecordtoInsert] = 0;


      }
      else if(FeasibleInsertforanyRecord == NOTFOUND)
      {
         CaninsertAllRecordsinSameRoute = FALSE;
         NORoute = 0;
         break;
      }
   }

   /*If it can insert all Records in same route */
   if(CaninsertAllRecordsinSameRoute == TRUE)
   {
      NORoute = 1;
      CopyOneRoute(OneTRSolomon, RT); /* Update RT */
   }

   return NORoute;
}


/*************************************************************/
/* Function : ConstructandPickupBestBeighborOneRoute(int *OneRoute)
/* - Search the best solution when that solution has only one route.
/* - Use Swap 3-ConsecutivePositionsWithinRoute move to construct neighbor
solution.
/*
/*************************************************************/
void ConstructandPickupBestBeighborOneRoute(int *OneRoute)
{
   int i,nb=0;
   int RC,Position;
   int temp;
   int Feasible, CSTL;

   double MinCost, Cost;

   RC = CountNORecordsinRoute(OneRoute);

   /* Position until swap 3 onsecutive-RC */
   Position = 2*RC - 2;

   for(i=1; i<=Position; i++)
   {
      /* Type 2 - 1 3 2 */
      CopyOneRoute(OneRoute, NBRoute[1]);
      temp = NBRoute[1][i+2];
      NBRoute[1][i+2] = NBRoute[1][i+1];
      NBRoute[1][i+1] = temp;

      Feasible = CheckFeasibleRoute(NBRoute[1]);
      CSTL = CheckSolutioninTabuList(NBRoute);

      if(Feasible == TRUE && CSTL == NOTFOUND)
      {
         nb = nb+1;
```

```
    Cost = EvaluateCostOneRoute(NBRoute[1]);
    if(nb==1)
    {
       MinCost = Cost;
       CopyOneRoute(NBRoute[1], BestNBRoute[1]);
    }
    else if(nb > 1)
    {
       if(Cost < MinCost)
       {
          MinCost = Cost;
          CopyOneRoute(NBRoute[1], BestNBRoute[1]);
       }
    }
}

/* Type 3 - 2 1 3 */
CopyOneRoute(OneRoute, NBRoute[1]);
temp = NBRoute[1][i+1];
NBRoute[1][i+1] = NBRoute[1][i];
NBRoute[1][i] = temp;

Feasible = CheckFeasibleRoute(NBRoute[1]);
CSTL = CheckSolutioninTabuList(NBRoute);

if(Feasible == TRUE && CSTL == NOTFOUND)
{
   nb = nb+1;
   Cost = EvaluateCostOneRoute(NBRoute[1]);
   if(nb==1)
   {
      MinCost = Cost;
      CopyOneRoute(NBRoute[1], BestNBRoute[1]);
   }
   else if(nb > 1)
   {
      if(Cost < MinCost)
      {
         MinCost = Cost;
         CopyOneRoute(NBRoute[1], BestNBRoute[1]);
      }
   }
}

/* Type 4 - 2 3 1 */
CopyOneRoute(OneRoute, NBRoute[1]);
temp = NBRoute[1][i];
NBRoute[1][i] = NBRoute[1][i+1];
NBRoute[1][i+1] = NBRoute[1][i+2];
NBRoute[1][i+2] = temp;

Feasible = CheckFeasibleRoute(NBRoute[1]);
CSTL = CheckSolutioninTabuList(NBRoute);

if(Feasible == TRUE && CSTL == NOTFOUND)
{
   nb = nb+1;
   Cost = EvaluateCostOneRoute(NBRoute[1]);
   if(nb==1)
   {
      MinCost = Cost;
      CopyOneRoute(NBRoute[1], BestNBRoute[1]);
   }
   else if(nb > 1)
```

```
    {
       if(Cost < MinCost)
       {
          MinCost = Cost;
          CopyOneRoute(NBRoute[1], BestNBRoute[1]);
       }
    }
}

/* Type 5 - 3 1 2 */
CopyOneRoute(OneRoute, NBRoute[1]);
temp = NBRoute[1][i+2];
NBRoute[1][i+2] = NBRoute[1][i+1];
NBRoute[1][i+1] = NBRoute[1][i];
NBRoute[1][i] = temp;

Feasible = CheckFeasibleRoute(NBRoute[1]);
CSTL = CheckSolutioninTabuList(NBRoute);

if(Feasible == TRUE && CSTL == NOTFOUND)
{
   nb = nb+1;
   Cost = EvaluateCostOneRoute(NBRoute[1]);
   if(nb==1)
   {
      MinCost = Cost;
      CopyOneRoute(NBRoute[1], BestNBRoute[1]);
   }
   else if(nb > 1)
   {
      if(Cost < MinCost)
      {
         MinCost = Cost;
         CopyOneRoute(NBRoute[1], BestNBRoute[1]);
      }
   }
}

/* Type 6 - 3 2 1 */
CopyOneRoute(OneRoute, NBRoute[1]);
temp = NBRoute[1][i];
NBRoute[1][i] = NBRoute[1][i+2];
NBRoute[1][i+2] = temp;

Feasible = CheckFeasibleRoute(NBRoute[1]);
CSTL = CheckSolutioninTabuList(NBRoute);

if(Feasible == TRUE && CSTL == NOTFOUND)
{
   nb = nb+1;
   Cost = EvaluateCostOneRoute(NBRoute[1]);
   if(nb==1)
   {
      MinCost = Cost;
      CopyOneRoute(NBRoute[1], BestNBRoute[1]);
   }
   else if(nb > 1)
   {
      if(Cost < MinCost)
      {
         MinCost = Cost;
         CopyOneRoute(NBRoute[1], BestNBRoute[1]);
      }
   }
```

```
      }
   }

   /* Copy Back */
   CopyOneRoute(BestNBRoute[1], OneRoute);
}


/*************************************************************/
/* Function : CheckGuaranteedTime(int *Route)
/* - Check Guaranteed Time of "Any 1 Route".
/*
/*************************************************************/
int CheckGuaranteedTime(int *Route)
{
   int CheckGT;
   int t;
   double Time, BetweenTime;

   int PickupPoint, DeliveryPoint;

   CheckGT = TRUE;
   t = 1;
   Time = 0;
   BetweenTime = 0;

   if(Route[1] != 0)
   {
      do
      {
         Time = Time + TravelTime[Route[t-1]][Route[t]];

         if(Route[t] <= Records) /* Pickup Point */
         {
            if(Time < ReadyTime[Route[t]])
            {
               Time = ReadyTime[Route[t]];
               TimetoBeginService[Route[t]] = Time;
               Time = TimetoBeginService[Route[t]] + ServiceTime;
            }
            else if(Time >= ReadyTime[Route[t]])
            {
               TimetoBeginService[Route[t]] = Time;
               Time = TimetoBeginService[Route[t]] + ServiceTime;
            }
         }
         else if(Route[t] > Records) /* Delivery point */
         {
            TimetoBeginService[Route[t]] = Time;
            Time = TimetoBeginService[Route[t]] + ServiceTime;

            DeliveryPoint = Route[t];
            PickupPoint = Route[t]-Records;

            BetweenTime  = TimetoBeginService[DeliveryPoint] -
ReadyTime[PickupPoint];

            if(BetweenTime > GuaranteedTime)
            {
               CheckGT = FALSE;
               break;
            }
         }
```

```
            t = t+1;
         }
         while(Route[t] != 0);
      }

      return CheckGT;
}

/*************************************************************/
/* Function : CheckCapacity(int *Route)
/* - Check Capacity of "Any 1 Route".
/*
/*************************************************************/
int CheckCapacity(int *Route)
{
      int CheckCP;
      int t,Load;

      CheckCP = TRUE;
      t = 1;
      Load = 0;

      if(Route[1] != 0)
      {
         do
         {
            LoadinVehicle[Route[t]] = Load + Demand[Route[t]];
            Load = LoadinVehicle[Route[t]];

            if(Load > Capacity)
            {
               CheckCP = FALSE;
               break;
            }

            t = t+1;
         }
         while(Route[t] != 0);
      }

      return CheckCP;
}


/*************************************************************/
/* Function : CheckPredecessorSuccessor(int *Route)
/* - Check Predecessor and Successor of "Any 1 Route".
/*
/*************************************************************/
int CheckPredecessorSuccessor(int *Route)
{
      int CheckPS;
      int i,j,p,Position;
      int Predecessor,Successor;

      CheckPS = TRUE;

      p = CountNORecordsinRoute(Route);

      Position = 2*p;

      for(i=1; i<=Position; i++)
      {
         if(Route[i] <= Records)
```

```
      {
         Predecessor = i;
         for(j=1; j<=Position; j++)
         {
            if(Route[j] == Route[i]+Records)
            {
               Successor = j;
               break;
            }
         }
         if(Predecessor > Successor)
         {
            CheckPS = FALSE;
            break;
         }
      }
      else if(Route[i] > Records)
      {
         Successor = i;
         for(j=1; j<=Position; j++)
         {
            if(Route[j] == Route[i]-Records)
            {
               Predecessor = j;
               break;
            }
         }
         if(Predecessor > Successor)
         {
            CheckPS = FALSE;
            break;
         }
      }
   }

   return CheckPS;
}


/*************************************************************/
/* Function : CheckFeasibleRoute(int *Route)
/* - Check Feasible Route (Ony One Route)
/*
/*************************************************************/
int CheckFeasibleRoute(int *Route)
{
   int FeasibleRoute;
   int CheckGT, CheckCP, CheckPD;

   FeasibleRoute = FALSE;
   CheckGT = FALSE;
   CheckCP = FALSE;

   CheckGT = CheckGuaranteedTime(Route);
   CheckCP = CheckCapacity(Route);
   CheckPD = CheckPredecessorSuccessor(Route);

   if(CheckGT == TRUE && CheckCP == TRUE && CheckPD)
   {
      FeasibleRoute = TRUE;
   }

   return FeasibleRoute;
}
```

```
/*************************************************************/
/* Function : CheckFeasibleSolution(int **Route)
/* - Check Feasible Solution
/*
/*************************************************************/
int CheckFeasibleSolution(int **Route)
{
   int FeasibleSolution;
   int CheckGT, CheckCP;
   int V,Vehicles;


   FeasibleSolution = TRUE;
   RearrangeAllRoutes(Route);
   Vehicles = CountVehicles(Route);

   for(V=1; V<=Vehicles; V++)
   {
      CheckGT = CheckGuaranteedTime(Route[V]);
      CheckCP = CheckCapacity(Route[V]);

      if(CheckGT == FALSE || CheckCP == FALSE)
      {
         FeasibleSolution = FALSE;
         break;
      }
   }

   return FeasibleSolution;
}


/*************************************************************/
/* Function : PDShiftOperator()
/* - PD-Shift Operator.
/*
/*************************************************************/
void PDShiftOperator(int BestRecord, int *RT1, int *RT2, int
BestOriginPosition, int BestDestinationPosition)
{
   RemoveRecord(BestRecord, RT1);
   InsertRecord(BestRecord, RT2, BestOriginPosition,
BestDestinationPosition);
}


/*************************************************************/
/* Function : PDExchangeOperator(int *RT1, int *RT2, int P1, int P2, int
OriginPosition_P1intoR2, int DestinationPosition_P1intoR2, int
OriginPosition_P2intoR1, int DestinationPosition_P2intoR1)
/* - PD-Exchange Operator.
/*
/*************************************************************/
void PDExchangeOperator(int P1, int *RT1, int P2, int *RT2, int
OriginPosition_P1intoR2, int DestinationPosition_P1intoR2, int
OriginPosition_P2intoR1, int DestinationPosition_P2intoR1)
{
   RemoveRecord(P1, RT1);
   RemoveRecord(P2, RT2);

   InsertRecord(P1, RT2, OriginPosition_P1intoR2,
DestinationPosition_P1intoR2);
```

```
   InsertRecord(P2, RT1, OriginPosition_P2intoR1,
DestinationPosition_P2intoR1);
}


/*************************************************************/
/* Function : RearrangeOperator_WithinRouteInsertion(int Node, int *Route,
int BestPosition)
/* - Rearrange Operator - Within Route Insertion.
/*
/*************************************************************/
void RearrangeOperator_WithinRouteInsertion(int Node, int *Route, int
BestPosition)
{
   RemoveNode(Node, Route);
   InsertNode(Node, Route, BestPosition);
}


/*************************************************************/
/* Function : RemoveNode(int RemoveRecord, int *Route)
/* - Remove Record out of any 1 Route.
/*
/*************************************************************/
void RemoveNode(int RemoveNode, int *Route)
{
   int s,t;

   ClearRoute(OneTRF);

   s = 0;
   t = 0;

   do
   {
      if(Route[s] != RemoveNode)
      {
         OneTRF[t] = Route[s];
         t = t+1;
      }

      s = s+1;
   }
   while(Route[s] != 0);

   OneTRF[t] = 0;

   /* Copy back Temp Route to Real Route */
   /* Clear Route */
   for(s=0; s<=2*Records+1; s++)
   {
      Route[s] = 0;
   }

   /* Copy Back */
   s = 0;

   do
   {
      Route[s] = OneTRF[s];
      s = s+1;
   }
   while(OneTRF[s] != 0);
   Route[s] = 0;
```

```
}

/*************************************************************/
/* Function : InsertNode(int Node, int *Route, int Position)
/* - Remove Record out of any 1 Route.
/*
/*************************************************************/
void InsertNode(int Node, int *Route, int Position)
{
   int s,t;
   int rn;


   /* Allocate Memory for Temp Route */
   ClearRoute(OneTRF);

   /* Insert Origin */
   t = 0;
   s = 0;

   do
   {
      OneTRF[t] = Route[s];
      s = s+1;
      t = t+1;
   }
   while(s < Position);
   OneTRF[t] = Node;
   t = t+1;

   do
   {
      if(Route[s] != 0)
      {
         OneTRF[t] = Route[s];
         s = s+1;
         t = t+1;
      }
   }
   while(Route[s] != 0);
   OneTRF[t] = 0;

   /* Copy back Temp Route to Real Route */
   /* Clear Route */
   for(s=0; s<=2*Records+1; s++)
   {
      Route[s] = 0;
   }

   /* Copy Back */
   s = 0;

   do
   {
      Route[s] = OneTRF[s];
      s = s+1;
   }
   while(OneTRF[s] != 0);
   Route[s] = 0;
}

/*************************************************************/
```

```
/* Function : RemoveRecord(int *Route, int RemoveRecord)
/* - Remove Record out of any 1 Route.
/*
/*************************************************************/
void RemoveRecord(int RemoveRecord, int *Route)
{
   int s,t;


   /* TempRoute in Function */
   ClearRoute(OneTRF);

   s = 0;
   t = 0;

   do
   {
      if((Route[s] != RemoveRecord) && (Route[s] != RemoveRecord+Records))
      {
         OneTRF[t] = Route[s];
         t = t+1;
      }

      s = s+1;
   }
   while(Route[s] != 0);

   OneTRF[t] = 0;

   /* Copy back Temp Route to Real Route */
   /* Clear Route */
   for(s=0; s<=2*Records+1; s++)
   {
      Route[s] = 0;
   }

   /* Copy Back */
   s = 0;

   do
   {
      Route[s] = OneTRF[s];
      s = s+1;
   }
   while(OneTRF[s] != 0);
   Route[s] = 0;
}


/*************************************************************/
/* Function : InsertRecord(int RandomRecord, int *Route, int OriginPosition,
int DestinationPosition)
/* - Remove Record out of any 1 Route.
/*
/*************************************************************/
void InsertRecord(int RandomRecord, int *Route, int OriginPosition, int
DestinationPosition)
{
   int s,t;
   int rn;

   /* TempRoute in Function */
   ClearRoute(OneTRF);
```

```
/* Insert Origin */
t = 0;
s = 0;

do
{
   OneTRF[t] = Route[s];
   s = s+1;
   t = t+1;
}
while(s < OriginPosition);
OneTRF[t] = RandomRecord;
t = t+1;

do
{
   if(Route[s] != 0)
   {
      OneTRF[t] = Route[s];
      s = s+1;
      t = t+1;
   }
}
while(Route[s] != 0);
OneTRF[t] = 0;

/* Copy back Temp Route to Real Route */
/* Clear Route */
for(s=0; s<=2*Records+1; s++)
{
   Route[s] = 0;
}

/* Copy Back */
s = 0;

do
{
   Route[s] = OneTRF[s];
   s = s+1;
}
while(OneTRF[s] != 0);
Route[s] = 0;


/*********************************************************/

ClearRoute(OneTRF);

/* Insert Destination */
t = 0;
s = 0;
do
{
   OneTRF[t] = Route[s];
   s = s+1;
   t = t+1;
}
while(s < DestinationPosition);
OneTRF[t] = RandomRecord+Records;
t = t+1;

do
{
```

```
      if(Route[s] != 0)
      {
         OneTRF[t] = Route[s];
         s = s+1;
         t = t+1;
      }
   }
   while(Route[s] != 0);
   OneTRF[t] = 0;


   /* Copy back Temp Route to Real Route */
   /* Clear Route */
   for(s=0; s<=2*Records+1; s++)
   {
      Route[s] = 0;
   }

   /* Copy Back */
   s = 0;

   do
   {
      Route[s] = OneTRF[s];
      s = s+1;
   }
   while(OneTRF[s] != 0);
   Route[s] = 0;
}


/*************************************************************/
/* Function : CountNORecordsinRoute(int *Route)
/* - Cound No. of Records in Any 1 RouteRe.
/*
/*************************************************************/
int CountNORecordsinRoute(int *Route)
{
   int s,p;

   s = 1;
   p = 0;

   while(Route[s] != 0)
   {
      p = p+1;
      s = s+1;
   }

   p = p/2;

   return p;
}


/*************************************************************/
/* Function : CopyOneRoute(int *R1, int *R2)
/* - Copy One Route from R1 to R2.
/*
/*************************************************************/
void CopyOneRoute(int *R1, int *R2)
{
   int s;
```

```
   s = 0;

   do
   {
      R2[s] = R1[s];
      s = s+1;
   }
   while(R1[s] != 0);
   R2[s] = 0;
}


/*************************************************************/
/* Function : CopyAllRoutes(int **R1, int **R2)
/* - Copy All Routes from R1 to R2.
/*
/*************************************************************/
void CopyAllRoutes(int **R1, int **R2)
{
   int s;
   int i;

   for(i=1; i<=Records; i++)
   {
      s = 0;

      do
      {
         R2[i][s] = R1[i][s];
         s = s+1;
      }
      while(R1[i][s] != 0);

      R2[i][s] = 0;
   }
}


/*************************************************************/
/* Function : RearrangeAllRoutes(int **Route)
/* - Re-arrange All Routes.
/*
/*************************************************************/
void RearrangeAllRoutes(int **Route)
{
   int i,n,r,s;

   int V;


   V = CountVehicles(Route);

   ClearRoutes(ManyTRSF);

   /* Re-arrange Rounts */
   /* Copy to Temp Routes */
   r = 0;
   for(n=1; n<=Records; n++)
   {
      if(Route[n][1] != 0) /* Exist Route */
      {
         r = r+1;
         s = 0;
```

```
            do
            {
               ManyTRSF[r][s] = Route[n][s];
               s = s+1;
            }
            while(Route[n][s] != 0);
            ManyTRSF[r][s] = 0;
         }
      }

      /* Clear Route */
      for(n=1; n<=Records; n++)
      {
         for(s=0; s<=2*Records+1; s++)
         {
            Route[n][s] = 0;
         }
      }

      /* Copy Back from ManyTR to Route */
      for(n=1; n<=V; n++)
      {
         s = 0;

         do
         {
            Route[n][s] = ManyTRSF[n][s];
            s = s+1;
         }
         while(ManyTRSF[n][s] != 0);
         Route[n][s] = 0;
      }
   }


   /*************************************************************/
   /* Function : FindRoute(int PD)
   /* - Find Route NO. for any Record.
   /*
   /*************************************************************/
   int FindRoute(int PD)
   {
      int RT;
      int V;
      int s;

      V = CountVehicles(Route);

      for(i=1; i<=V; i++)
      {
         s = 0;

         while(Route[i][s+1] != 0)
         {
            if(Route[i][s] == PD)
            {
               RT = i;
               break;
            }
            s = s+1;
         }
      }

      return RT;
```

```
}

/*************************************************************/
/* Function : EvaluateCostOneRoute(int *Route)
/* - Calculate Total Distances of All Routes.
/*
/*************************************************************/
double EvaluateCostOneRoute(int *Route)
{
   int i,j,s,V;
   double TC,TD;

   TC = 0;
   TD = 0;

   if(Route[1] == 0)
   {
      V = 0;
   }
   else
   {
      V = 1;
   }

   s = 0;
   while(Route[s+1] != 0)
   {
      TD = TD + Distance[Route[s]][Route[s+1]];
      s = s+1;
   }

   TD = TD + Distance[Route[s]][Route[s+1]];

   TC = 10000*V + 1*TD;

   return TC;
}


/*************************************************************/
/* Function : EvaluateCostAllRoutes(int **Route)
/* - Calculate Total Distances of All Routes.
/*
/*************************************************************/
double EvaluateCostAllRoutes(int **Route)
{
   int i,j,s;
   double TC,TD;

   TC = 0;
   TD = 0;
   RearrangeAllRoutes(Route);
   Vehicles = CountVehicles(Route);

   for(i=1; i <= Vehicles; i++)
   {
      s = 0;

      while(Route[i][s+1] != 0)
      {
         TD = TD + Distance[Route[i][s]][Route[i][s+1]];
         s = s+1;
      }
```

```
      TD = TD + Distance[Route[i][s]][Route[i][s+1]];
   }

   TC = 10000*Vehicles + 1*TD;

   return TC;
}


/*************************************************************/
/* Function : PrintOneRoute(int *Route, int r)
/* - Print One Route.
/*
/*************************************************************/
void PrintOneRoute(int *Route, int r)
{
   int s;

   s = 0;

   printf("Route[%d] = ", r);

   do
   {
      printf("%d  ", Route[s]);

      s = s+1;
   }
   while(Route[s] != 0);

   printf("%d\n\n", Route[s]);
}

/*************************************************************/
/* Function : PrintAllRoutes(int **Route)
/* - Print All Routes.
/*
/*************************************************************/
void PrintAllRoutes(int **Route)
{
   int n,s,Vehicles;

   printf("\n**** Print All Routes ***\n");

   RearrangeAllRoutes(Route);
   Vehicles = CountVehicles(Route);

   for(n=1; n<=Vehicles; n++)
   {
      s = 0;

      printf("Route[%d]  =  ", n);

      do
      {
         printf("%d  ", Route[n][s]);

         s = s+1;
      }
      while(Route[n][s] != 0);

      printf("%d\n", Route[n][s]);
   }
}
```

```
/*************************************************************/
/* Function : FPrintAllRoutes(FILE *f, int **Route)
/* - Print All Routes.
/*
/*************************************************************/
void FPrintAllRoutes(FILE *f, int **Route)
{
   int n,s,Vehicles;


   RearrangeAllRoutes(Route);
   Vehicles = CountVehicles(Route);

   for(n=1; n<=Vehicles; n++)
   {
      s = 0;

      fprintf(f, "Route[%d] = ", n);

      do
      {
         if(Route[n][s] == 0)
         {
            fprintf(f, "%d ", Route[n][s]);
         }
         else if(Route[n][s] <= Records)
         {
            fprintf(f, "P%d ", Route[n][s]);
         }
         else if(Route[n][s] > Records)
         {
            fprintf(f, "D%d ", Route[n][s]-Records);
         }

         s = s+1;
      }
      while(Route[n][s] != 0);

      fprintf(f, "%d\n", Route[n][s]);
   }
}


/*************************************************************/
/* Function : CountVehicles(int **Route)
/* - Count The number of Vehicles.
/*
/*************************************************************/
int CountVehicles(int **Route)
{
   int i;
   int V;

   V = 0;

   /*NO. of Route = NO. of Records (jobs) */
   for(i=1; i<=Records; i++)
   {
      if(Route[i][1] != 0) /* Exist Route */
      {
         V = V+1;
      }
```

```
    }

    return V;
}


/*************************************************************/
/* Function : CalculateTotalDistance(int **Route)
/* - Calculate Total Distance.
/*
/*************************************************************/
double CalculateTotalDistance(int **Route)
{
   int i,s;
   double TD;

   TD = 0;
   RearrangeAllRoutes(Route);
   Vehicles = CountVehicles(Route);

   for(i=1; i <= Vehicles; i++)
   {
      s = 0;

      while(Route[i][s+1] != 0)
      {
         TD = TD + Distance[Route[i][s]][Route[i][s+1]];
         s = s+1;
      }
      TD = TD + Distance[Route[i][s]][Route[i][s+1]];
   }

   return TD;
}


/*************************************************************/
/* Function : CalculateTotalTravelTime(int **Route)
/* - Calculate Total Travel Time.
/*
/*************************************************************/
double CalculateTotalTravelTime(int **Route)
{
   int V,t;
   double Time,TotalTravelTime;


   RearrangeAllRoutes(Route);
   Vehicles = CountVehicles(Route);
   TotalTravelTime = 0;

   for(V=1; V<=Vehicles; V++)
   {
      t = 1;
      Time = 0;

      while(Route[V][t] != 0)
      {
         Time = Time + TravelTime[Route[V][t-1]][Route[V][t]];

         if(Route[V][t] <= Records) /* Pickup Point */
         {
            if(Time < ReadyTime[Route[V][t]])
            {
```

```
               Time = ReadyTime[Route[V][t]];
               TimetoBeginService[Route[V][t]] = Time;
               Time = TimetoBeginService[Route[V][t]] + ServiceTime;
            }
            else if(Time >= ReadyTime[Route[V][t]])
            {
               TimetoBeginService[Route[V][t]] = Time;
               Time = TimetoBeginService[Route[V][t]] + ServiceTime;
            }
         }
         else if(Route[V][t] > Records) /* Delivery Point */
         {
            TimetoBeginService[Route[V][t]] = Time;
            Time = TimetoBeginService[Route[V][t]] + ServiceTime;
         }

         t = t+1;
      }
      Time = Time + TravelTime[Route[V][t-1]][Route[V][t]];

      TotalTravelTime = TotalTravelTime + Time;

   }

   return TotalTravelTime;
}


/*********************************************************/
/* Function : CalculateTotalWaitingTime(int **Route)
/* - Calculate Total Waiting Time.
/*
/*********************************************************/
double CalculateTotalWaitingTime(int **Route)
{
   int V,t;
   double Time,WaitingTime,TotalWaitingTime;


   RearrangeAllRoutes(Route);
   Vehicles = CountVehicles(Route);
   TotalWaitingTime = 0;

   for(V=1; V<=Vehicles; V++)
   {
      t = 1;
      Time = 0;
      WaitingTime = 0;

      while(Route[V][t] != 0)
      {
         Time = Time + TravelTime[Route[V][t-1]][Route[V][t]];

         if(Route[V][t] <= Records) /* Pickup Point */
         {
            if(Time < ReadyTime[Route[V][t]]) /* Vehicle arrive point before
Ready Time, So Vehicle must wait for service */
            {
               WaitingTime = WaitingTime + (ReadyTime[Route[V][t]] - Time);
               Time = ReadyTime[Route[V][t]];
               TimetoBeginService[Route[V][t]] = Time;
               Time = TimetoBeginService[Route[V][t]] + ServiceTime;
            }
            else if(Time >= ReadyTime[Route[V][t]])
```

```
            {
               TimetoBeginService[Route[V][t]] = Time;
               Time = TimetoBeginService[Route[V][t]] + ServiceTime;
            }
         }
         else if(Route[V][t] > Records) /* Delivery Point */
         {
            TimetoBeginService[Route[V][t]] = Time;
            Time = TimetoBeginService[Route[V][t]] + ServiceTime;
         }

         t = t+1;
      }


      TotalWaitingTime = TotalWaitingTime + WaitingTime;
   }

   return TotalWaitingTime;
}


/*************************************************************/
/* Function : EvaluateWaitingTimeOneRoute(int *Route)
/* - Calculate Waiting Time in one route.
/* - Use in function: RearrangeRoute_MinimumWaitingTime
/*
/*************************************************************/
double EvaluateWaitingTimeOneRoute(int *Route)
{
   int t;
   double Time,WaitingTime;


   WaitingTime = 0;

   t = 1;
   Time = 0;
   WaitingTime = 0;

   while(Route[t] != 0)
   {
      Time = Time + TravelTime[Route[t-1]][Route[t]];

      if(Route[t] <= Records) /* Pickup Point */
      {
         if(Time < ReadyTime[Route[t]]) /* Vehicle arrive point before Ready
Time, So Vehicle must wait for service */
         {
            WaitingTime = WaitingTime + (ReadyTime[Route[t]] - Time);
            Time = ReadyTime[Route[t]];
            TimetoBeginService[Route[t]] = Time;
            Time = TimetoBeginService[Route[t]] + ServiceTime;
         }
         else if(Time >= ReadyTime[Route[t]])
         {
            TimetoBeginService[Route[t]] = Time;
            Time = TimetoBeginService[Route[t]] + ServiceTime;
         }
      }
      else if(Route[t] > Records) /* Delivery Point */
      {
         TimetoBeginService[Route[t]] = Time;
         Time = TimetoBeginService[Route[t]] + ServiceTime;
```

```
      }

      t = t+1;
   }

   return WaitingTime;
}


/*************************************************************/
/* Function : SetInitialTabuList()
/*
/*************************************************************/
void SetInitialTabuList()
{
   int i;

   for(i=1; i<=TabuIteration; i++)
   {
      NV[i]  = 0;
      TD[i]  = 0;
      TTT[i] = 0;
      TWT[i] = 0;
   }
}


/*************************************************************/
/* Function : UpdateTabuList(int **Route)
/* - Update Tabu List.
/*
/*************************************************************/
void UpdateTabuList(int **Route)
{
   int i,TempNV=0;
   double TempTD=0,TempTWT=0;

   for(i=2; i<=TabuIteration; i++)
   {
      /* Move Solution */
      NV[i-1]  = NV[i];
      TD[i-1]  = TD[i];
      TWT[i-1] = TWT[i];
   }

   NV[TabuIteration]  = CountVehicles(Route);
   TD[TabuIteration]  = CalculateTotalDistance(Route);
   TWT[TabuIteration] = CalculateTotalWaitingTime(Route);
}


/*************************************************************/
/* Function : CheckSolutioninTabuList(int **TempRoute)
/* - Check Solution is in Tabu List or not.
/*
/*************************************************************/
int CheckSolutioninTabuList(int **TempRoute)
{
   int i,CSTL;
   int TempNV=0;
   double TempTD=0, TempTTT=0, TempTWT=0;

   double DiffTD,DiffTTT,DiffTWT;
```

```
    CSTL = NOTFOUND;

    TempNV = CountVehicles(TempRoute);
    TempTD = CalculateTotalDistance(TempRoute);
    TempTWT = CalculateTotalWaitingTime(TempRoute);

    for(i=1; i<=TabuIteration; i++)
    {
       DiffTD  = fabs(TempTD-TD[i]);
       DiffTWT = fabs(TempTWT-TWT[i]);

       if((TempNV == NV[i]) && (DiffTD <= EPS) && (DiffTWT <= EPS))
       {

          CSTL = FOUND;
          break;
       }
    }

    return CSTL;
}


/*************************************************************/
/* Function : CreateRoutes(FILE *f, int **Route, int *PickupClinic, int
*DeliveryClinic)
/* - Create Routes from solution.
/*
/*************************************************************/
void CreateRoutes(FILE *f, int **Route, int *PickupClinic, int
*DeliveryClinic)
{
   int n,t;

   double Time;
   int recentlyClinic;

   fprintf(f, "\n\n***** Create Routes *****\n");
   for(n=1; n<=Records; n++)
   {
      t=1;
      Time = 0;

      if(Route[n][1] != 0)
      {
         fprintf(f, "Route[%d] = C0",n);
         recentlyClinic = 0;

         while(Route[n][t] != 0)
         {
            /* 1st service */
            if(t==1)
            {
               if(Route[n][t] <= Records)
               {
                  if(PickupClinic[Route[n][t]] == recentlyClinic)
                  {
                     fprintf(f, "(P%d", Route[n][t]);
                  }
                  else if(PickupClinic[Route[n][t]] != recentlyClinic)
                  {
                     fprintf(f, "  C%d(P%d", PickupClinic[Route[n][t]],
Route[n][t]);
```

```
                }
                recentlyClinic = PickupClinic[Route[n][t]];
            }
            else if(Route[n][t] > Records)
            {
                if(DeliveryClinic[Route[n][t]-Records] == recentlyClinic)
                {
                    fprintf(f, "(D%d", Route[n][t]-Records);
                }
                else if(DeliveryClinic[Route[n][t]-Records] !=
recentlyClinic)
                {
                    fprintf(f, "  C%d(D%d", DeliveryClinic[Route[n][t]-
Records], Route[n][t]-Records);
                }
                recentlyClinic = DeliveryClinic[Route[n][t]-Records];
            }
        }
        else if(t>=2)
        {
            if(Route[n][t] <= Records)
            {
                if(PickupClinic[Route[n][t]] == recentlyClinic)
                {
                    fprintf(f, ",P%d", Route[n][t]);
                }
                else if(PickupClinic[Route[n][t]] != recentlyClinic)
                {
                    fprintf(f, ")  C%d(P%d", PickupClinic[Route[n][t]],
Route[n][t]);
                }
                recentlyClinic = PickupClinic[Route[n][t]];
            }
            else if(Route[n][t] > Records)
            {
                if(DeliveryClinic[Route[n][t]-Records] == recentlyClinic)
                {
                    fprintf(f, ",D%d", Route[n][t]-Records);
                }
                else if(DeliveryClinic[Route[n][t]-Records] !=
recentlyClinic)
                {
                    fprintf(f, ")  C%d(D%d", DeliveryClinic[Route[n][t]-
Records], Route[n][t]-Records);
                }
                recentlyClinic = DeliveryClinic[Route[n][t]-Records];
            }
        }

        t = t+1;
    }

    fprintf(f, ")  C0\n");

  }
 }
}


/*********************************************************/
/* Function : TransformSolution(int **Route, int *PickupClinic, int
*DeliveryClinic)
/* - Transform Routes to Answer of Origin Problem.
/*
```

```
/*************************************************************/
void TransformSolution(FILE *f, int **Route, int *PickupClinic, int
*DeliveryClinic)
{
   int n,t;

   double Time;
   double ArrivalTime, WaitingTime;

   fprintf(f, "\n***** Summary Routes *****\n\n");
   for(n=1; n<=Records; n++)
   {
      t=1;
      Time = 0;

      if(Route[n][1] != 0)
      {
         fprintf(f, "*** Route %d ***\n",n);
         fprintf(f, "Clinic NO.     Pickup        Delivery       Arrival
Time to Begin   Waiting    Beginning/Ending\n");
         fprintf(f, "               Record NO.     Record NO.       Time
Service        Time           Time\n");
         fprintf(f,
"*****************************************************************************
**********************\n");
         fprintf(f, "   0 (start)
0.00\n");

         while(Route[n][t] != 0)
         {
            Time = Time + TravelTime[Route[n][t-1]][Route[n][t]];

            if(Route[n][t] <= Records)
            {
               if(Time < ReadyTime[Route[n][t]])
               {
                  ArrivalTime = Time;
                  Time = ReadyTime[Route[n][t]];
                  TimetoBeginService[Route[n][t]] = Time;
                  Time = TimetoBeginService[Route[n][t]] + ServiceTime;
                  WaitingTime = TimetoBeginService[Route[n][t]] - ArrivalTime;

               }
               else if(Time >= ReadyTime[Route[n][t]])
               {
                  ArrivalTime = Time;
                  TimetoBeginService[Route[n][t]] = Time;
                  Time = TimetoBeginService[Route[n][t]] + ServiceTime;
                  WaitingTime = 0;
               }

               fprintf(f, "%7d", PickupClinic[Route[n][t]]);
               fprintf(f, "%11d", Route[n][t]);
               fprintf(f, "                -");
               fprintf(f, "%16.2f", ArrivalTime);
               fprintf(f, "%14.2f", TimetoBeginService[Route[n][t]]);
               fprintf(f, "%13.2f\n", WaitingTime);

            }
            else if(Route[n][t] > Records)
            {
               ArrivalTime = Time;
               TimetoBeginService[Route[n][t]] = Time;
               Time = TimetoBeginService[Route[n][t]] + ServiceTime;
```

```
                WaitingTime = 0;

                fprintf(f, "%7d", DeliveryClinic[Route[n][t]-Records]);
                fprintf(f, "          -");
                fprintf(f, "%16d", Route[n][t]-Records);
                fprintf(f, "%16.2f", ArrivalTime);
                fprintf(f, "%14.2f", TimetoBeginService[Route[n][t]]);
                fprintf(f, "%13.2f\n", WaitingTime);
             }
             t = t+1;
          }
          Time = Time + TravelTime[Route[n][t-1]][Route[n][t]];
          fprintf(f, "   0 (end)");
          fprintf(f, "      %77.2f\n\n", Time);
       }
    }
}


/*************************************************************/
/* Function : PrintOutput()
/*************************************************************/
void PrintOutput()
{
   int n,s,t,V;
   double Time,BestTTT, BestTWT;


   fprintf(fpo, "Computational Time = %0.4f sec\n\n", duration);

   fprintf(fpo, "\n***** Best Solution *****\n");
   FPrintAllRoutes(fpo, BestRoute);

   CreateRoutes(fpo, BestRoute, PickupClinic, DeliveryClinic);

   Vehicles = CountVehicles(BestRoute);
   fprintf(fpo, "\nThe number of used vehicles = %d\n", Vehicles);
   TotalCost = EvaluateCostAllRoutes(BestRoute);
   fprintf(fpo, "Total Travel Distance = %.2lf\n", TotalCost-10000*Vehicles);
   BestTTT = CalculateTotalTravelTime(BestRoute);
   fprintf(fpo, "Total Travel Time     = %.2lf\n", BestTTT);
   BestTWT = CalculateTotalWaitingTime(BestRoute);
   fprintf(fpo, "Total Waiting Time    = %.2lf\n\n", BestTWT);

   TransformSolution(fpo, BestRoute, PickupClinic, DeliveryClinic);

   fclose(fpo);
}


/*************************************************************/
/* Function : ClearRoute(int *Route)
/*
/*************************************************************/
void ClearRoute(int *Route)
{
   int i;

   for(i=0; i<=2*Records+1; i++)
   {
      Route[i] = 0;
   }
}
```

```
/*************************************************************/
/* Function : ClearRoutes(int **Route)
/*
/*************************************************************/
void ClearRoutes(int **Route)
{
   int i,j;

   for(i=0; i<=Records; i++)
   {
      for(j=0; j<=2*Records+1; j++)
      {
         Route[i][j] = 0;
      }
   }
}


/*************************************************************/
/* Function : AllocateMemory()
/* - Allocate Memory.
/*
/*************************************************************/
void AllocateMemory()
{

   PickupClinic = (int *) malloc((Records+1)*sizeof(int));
   DeliveryClinic = (int *) malloc((Records+1)*sizeof(int));

   Demand = (int *) malloc((2*Records+1)*sizeof(int));

   ReadyTime = (double *) malloc((Records+1)*sizeof(double));

   LoadinVehicle = (int *) malloc((2*Records+1)*sizeof(int));///***///

   TimetoBeginService = (double *) malloc((2*Records+1)*sizeof(double));

   Route = (int **) malloc((Records+1)*sizeof(int));
   for(i=0; i<=Records; i++)
   {
      Route[i] = (int *) malloc((2*Records+2)*sizeof(int));
   }

   BestRoute = (int **) malloc((Records+1)*sizeof(int));
   for(i=0; i<=Records; i++)
   {
      BestRoute[i] = (int *) malloc((2*Records+2)*sizeof(int));
   }

   BestNBRoute = (int **) malloc((Records+1)*sizeof(int));
   for(i=0; i<=Records; i++)
   {
      BestNBRoute[i] = (int *) malloc((2*Records+2)*sizeof(int));
   }

   Distance = (double **) malloc((2*Records+1)*sizeof(double));
   for(i=0; i<=2*Records; i++)
   {
      Distance[i] = (double *) malloc((2*Records+1)*sizeof(double));
   }

   STDistance = (double **) malloc((Clinics+1)*sizeof(double));
   for(i=0; i<=Clinics; i++)
```

```
    {
       STDistance[i] = (double *) malloc((Clinics+1)*sizeof(double));
    }

    TravelTime = (double **) malloc((2*Records+1)*sizeof(double));
    for(i=0; i<=2*Records; i++)
    {
       TravelTime[i] = (double *) malloc((2*Records+1)*sizeof(double));
    }

    /* Tabu Structure */
    NV  = (int *) malloc((TabuIteration+1)*sizeof(int));
    TD  = (double *) malloc((TabuIteration+1)*sizeof(double));
    TTT = (double *) malloc((TabuIteration+1)*sizeof(double));
    TWT = (double *) malloc((TabuIteration+1)*sizeof(double));

    PDList = (int *) malloc((Records+1)*sizeof(int));
    RecordList = (int *) malloc((Records+1)*sizeof(int));

    /*TempRoute*/
    OneTR1 = (int *) malloc((2*Records+2)*sizeof(int));
    OneTR2 = (int *) malloc((2*Records+2)*sizeof(int));
    OneTRF = (int *) malloc((2*Records+2)*sizeof(int));
    OneTRSolomon = (int *) malloc((2*Records+2)*sizeof(int));
    OneTRFSolomon = (int *) malloc((2*Records+2)*sizeof(int));

    ManyTR = (int **) malloc((Records+1)*sizeof(int));
    for(i=0; i<=Records; i++)
    {
       ManyTR[i] = (int *) malloc((2*Records+2)*sizeof(int));
    }

    ManyTRSF = (int **) malloc((Records+1)*sizeof(int));
    for(i=0; i<=Records; i++)
    {
       ManyTRSF[i] = (int *) malloc((2*Records+2)*sizeof(int));
    }

    ManyBTR = (int **) malloc((Records+1)*sizeof(int));
    for(i=0; i<=Records; i++)
    {
       ManyBTR[i] = (int *) malloc((2*Records+2)*sizeof(int));
    }

    /* Solomon Insertion */
    MinimumInsertionCostofRecord = (double *)
malloc((Records+1)*sizeof(double));
    BestOriginPositionofRecord = (int *) malloc((Records+1)*sizeof(int));
    BestDestinationPositionofRecord = (int *) malloc((Records+1)*sizeof(int));
    FeasibleInsertionPlace = (int *) malloc((Records+1)*sizeof(int));

    /* Variable for One Route */
    NBRoute = (int **) malloc((Records+1)*sizeof(int));
    for(i=0; i<=Records; i++)
    {
       NBRoute[i] = (int *) malloc((2*Records+2)*sizeof(int));
    }
}


/*********************************************************/
/* Function : FreeMemory()
/* - Free Memory.
/*
```

```
/*************************************************************/
void FreeMemory()
{
   int i,j;

   free(PickupClinic);
   free(DeliveryClinic);

   free(Demand);

   free(ReadyTime);

   free(LoadinVehicle);

   free(TimetoBeginService);

   for(i=0; i<=Records; i++)
   {
      free(Route[i]);
   }

   for(i=0; i<=Records; i++)
   {
      free(BestRoute[i]);
   }

   for(i=0; i<=Records; i++)
   {
      free(BestNBRoute[i]);
   }

   for(i=0; i<=2*Records; i++)
   {
      free(Distance[i]);
   }

   for(i=0; i<=Clinics; i++)
   {
      free(STDistance[i]);
   }

   for(i=0; i<=2*Records; i++)
   {
      free(TravelTime[i]);
   }

   free(NV);
   free(TD);
   free(TTT);
   free(TWT);

   free(PDList);
   free(RecordList);

   free(OneTR1);
   free(OneTR2);
   free(OneTRF);
   free(OneTRSolomon);
   free(OneTRFSolomon);

   for(i=0; i<=Records; i++)
   {
      free(ManyTR[i]);
   }
```

```
    for(i=0; i<=Records; i++)
    {
       free(ManyTRSF[i]);
    }

    for(i=0; i<=Records; i++)
    {
       free(ManyBTR[i]);
    }


    free(MinimumInsertionCostofRecord);
    free(BestOriginPositionofRecord);
    free(BestDestinationPositionofRecord);
    free(FeasibleInsertionPlace);

    for(i=0; i<=Records; i++)
    {
       free(NBRoute[i]);
    }
}
```

ประวัติผู้เขียนวิทยานิพนธ์

นายไตรภูมิ ปันกิติ เกิดเมื่อวันที่ 23 กันยายน พ.ศ.2523 บ้านเกิดอยู่ที่ อำเภอเมือง จังหวัดสมุทรปราการ สำเร็จการศึกษาระดับปริญญาวิทยาศาสตรบัณฑิต (เกียรตินิยมอันดับสอง) ภาควิชาคณิตศาสตร์ สาขาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์ จากจุฬาลงกรณ์มหาวิทยาลัย เมื่อพ.ศ. 2546 จากนั้นในปี พ.ศ. 2547 ได้เข้าศึกษาต่อระดับบัณฑิตศึกษา วิศวกรรมศาสตรมหาบัณฑิต ภาควิชาวิศวกรรมอุตสาหการ คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

รางวัลและทุนการศึกษาที่เคยได้รับขณะเรียนระดับปริญญาตรี ได้แก่ โล่รางวัลสอบไล่ได้คะแนนสูงสุดของภาควิชาคณิตศาสตร์ คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย ประจำปีการศึกษา 2545 และทุนการศึกษาสำหรับผู้มีผลการเรียนดีเยี่ยมเป็นลำดับที่หนึ่งของภาควิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย จากเครือซีเมนต์ไทย ประจำปี 2545

ปัจจุบันมีผลงานทางวิชาการ 2 เรื่อง ได้แก่

1. Tripoom Punkiti and Wipawee Tharmmaphornphilas, "A Methodology to Create Pick-up and Delivery Routes with Guaranteed Time Constraint", The 36th International Conference on Computers and Industrial Engineering, June 20-23, 2006, Taipei, Taiwan, CD-Rom Format.

2. Tripoom Punkiti and Wipawee Tharmmaphornphilas, "A MIP Model for an Out-Patient Record Delivery Problem", The 7th Asian Pacific Industrial Engineering and Management Systems Conference, December 17-20, 2006, Bangkok, Thailand, CD-Rom Format.