

# Capacitated Arc Routing Problem

Project 2 Report of

CS303 Artificial intelligence

Department of Computer Science and Engineering

BY

**Kebin Sun**

**11410151**



November 2018

# Table of Content

Capacitated Arc Routing Problem.....	i
Table of Content.....	ii
1 Preliminaries .....	1
1.1 Software & Hardware.....	1
1.2 Algorithms.....	1
2 Methodology.....	2
2.1 Representations .....	2
2.1.1 Mathematical Representation of CARP.....	2
2.1.2 Data Representation of the Program.....	3
2.2 Architecture.....	4
2.2.1 Flow Chart of Whole Program .....	4
2.2.2 Path Scanning.....	4
2.2.3 Operators .....	5
2.2.4 Crossover Operator .....	5
2.2.5 Local Search Operators.....	7
2.2.6 MAENS.....	8
2.3 Details of Algorithms .....	9
2.3.1 Implementations.....	9
2.3.2 Pseudo Code .....	10
3 Performances & Time Control.....	13
3.1 Theoretical Performance & Time Control.....	13
3.2 Empirical Performance .....	14
3.2.1 Platform Test.....	14
3.2.2 Sample Test .....	14
3.3 Analysis.....	16
Further Discussion .....	17
Acknowledgment .....	17
References .....	17



# 1 Preliminaries

*Arc Routing* is the process of selecting the best path in a network based on the route. The goal of many arc routing problems is to produce a route with the minimum amount of dead mileage, while also fully encompassing the edges required. The study of arc routing is rooted in the seminal work of the Swiss mathematician Leonhard Euler who was called to study the famous Königsberg bridges problem in the 18<sup>th</sup> century while he was chair of mathematics at the St. Petersburg Academy of Sciences. The study of this problem led Euler to lay the foundation for modern graph theory. Over the following years and to this day, arc routing has evolved into a rich research area, firmly embedded within the broader domain of combinatorial optimization.[\[1\]](#)

In this project, however, we are specifically interested in the undirected *Capacitated Arc Routing Problem* (CARP). The undirected CARP consists of demands placed on the edges, and each edge must meet the demand. An example is garbage collection, where each route might require both a garbage collection and a recyclable collection.[\[2\]](#)

The CARP has attracted much attention during the last decade due to its wide applications in real life. Since it is NP-hard and exact methods are only applicable to small instances, heuristic and metaheuristic methods are widely adopted when solving the problem.[\[3\]](#)

## 1.1 Software & Hardware

This project is written in *Python* () with editor *Visual Studio Code* (). The main testing platform is *Windows 10 Professional Edition* (version 1803) with Intel<sup>®</sup> Core<sup>™</sup> i7-8700K @ 3.70~4.70GHz with 6 cores and 12 threads.

## 1.2 Algorithms

This project involves the published research of Prof. Tang – the *Memetic Algorithm with Extended Neighborhood Search* (MAENS), a metaheuristic method and a heuristic method, *Path Scanning* with ellipse rule, to generate the initial population of solutions.

## 2 Methodology

In this part, I will introduce the representations of the CARP & the applied approaches, the structure of my program as well as and the detailed algorithms.

### 2.1 Representations

#### 2.1.1 Mathematical Representation of CARP

The undirected CAPR we concern can be described as follows: a graph  $G = (V, E)$ , with a set of vertices denoted by  $V$ , a set of (undirected) edges denoted by  $E$  is given. There is a central depot vertex  $dep \in V$ , where a set of vehicles are based. A subset  $E_R \subseteq E$  composed of all the edges required to be served. The elements of this subset are called *edge tasks*, each of which is associated with a demand and a pass-by cost. The demand is zero for the edges that do not require service  $C_E E_R$ . A solution to the problem is a routing plan that consists of a number of routes for the vehicles, and the objective is to minimize the total cost of the routing plan subject to the following constraints:

- 1) Each route starts and ends at the depot;
- 2) Each task is served exactly once;
- 3) The total demand of each route must not exceed the capacity  $Q$ .

Each edge is assigned a unique positive ID, say  $t$ . It's reversed edge is not considered a different edge but can appear in the solution to represent a reversed service direction. Thus, for convenience, the ID of the reversed edge is set to be the its minus, say  $-t$ . Each ID  $t$  is associated with four features, namely  $source(t)$ ,  $target(t)$ ,  $cost(t)$ , and  $demand(t)$  (sometimes it is written as  $t.source$ ,  $t.cost$ , etc.). I represent a solution to CARP as an ordered list of tasks (IDs), denoted by

$$S = \left( 0, \overbrace{t_{11}, t_{12}, \dots, t_{1n_1}}^{R_1}, 0, \overbrace{t_{21}, \dots, t_{2n_2}}^{R_2}, 0, \dots, \overbrace{t_{p1}, \dots, t_{pn_p}}^{R_p}, 0 \right). \quad (2.1)$$

Where 0 is the ID of depot and  $p$  is the number of routes (typically it is also the number of vehicles  $N_V$  for a feasible solution). Therefore, the total cost of  $S$  is:

$$TC(S) = \sum_{i=1}^{|S|-1} [cost(S_i) + sp(S_i, S_{i+1})]. \quad (2.2)$$

Where  $sp(S_i, S_{i+1})$  stands for the shortest path's cost between  $target(S_i)$  and  $source(S_{i+1})$  and of course  $cost(0) = 0$ . The load of each route is

$$load(R_i) = \sum_{k=1}^{n_i} demand(t_{ik}). \quad (2.3)$$

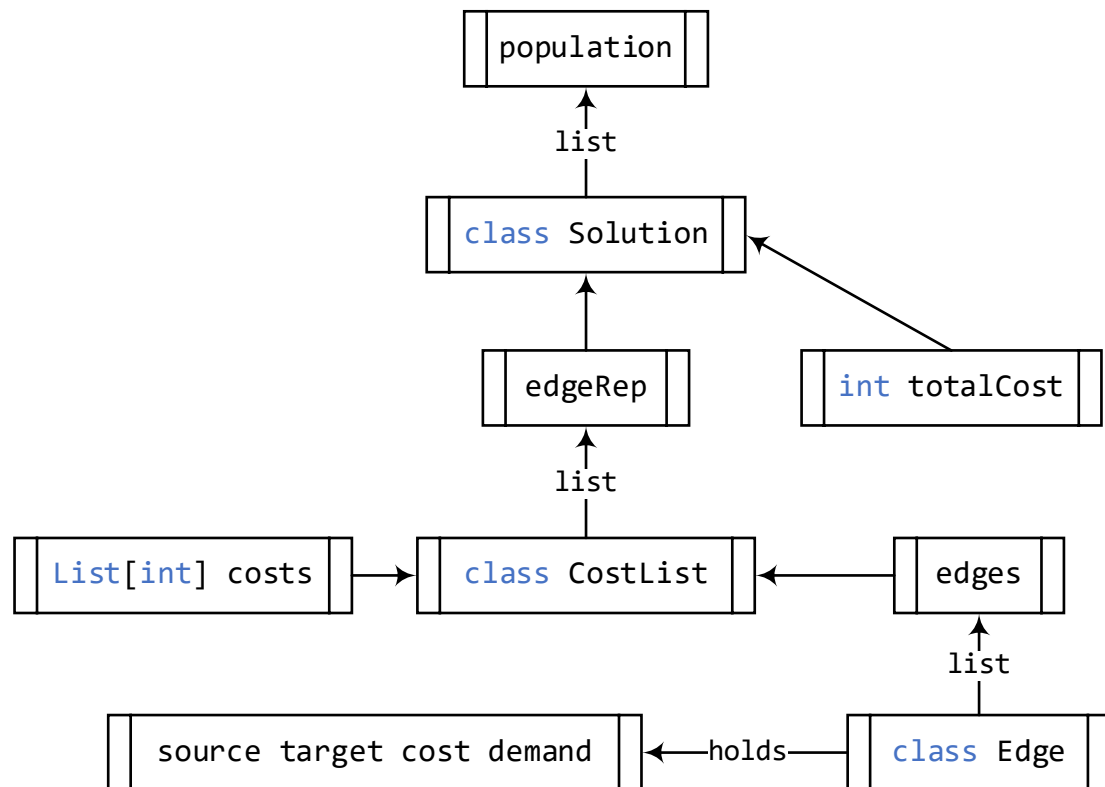
Given all the above notations and the aforementioned three constraints of CARP, we now arrive at the following representation of it:

$$\begin{aligned} \min_S TC(S) &= \sum_{i=1}^{|S|-1} [cost(S_i) + sp(S_i, S_{i+1})] \\ \text{s.t. : } app[abs(S_i)] &= 1, \forall S_i \in E_R \\ p &\leq N_V \\ load(R_i) &\leq Q, \forall i \in 1, 2, \dots, p \end{aligned} \quad (2.4)$$

Where  $app[abs(S_i)]$  counts the times that task  $S_i$  (and its reverse) appears in the whole sequence,  $N_V$  is the number of vehicles available at the depot, and  $Q$  is the vehicle's capacity.

### 2.1.2 Data Representation of the Program

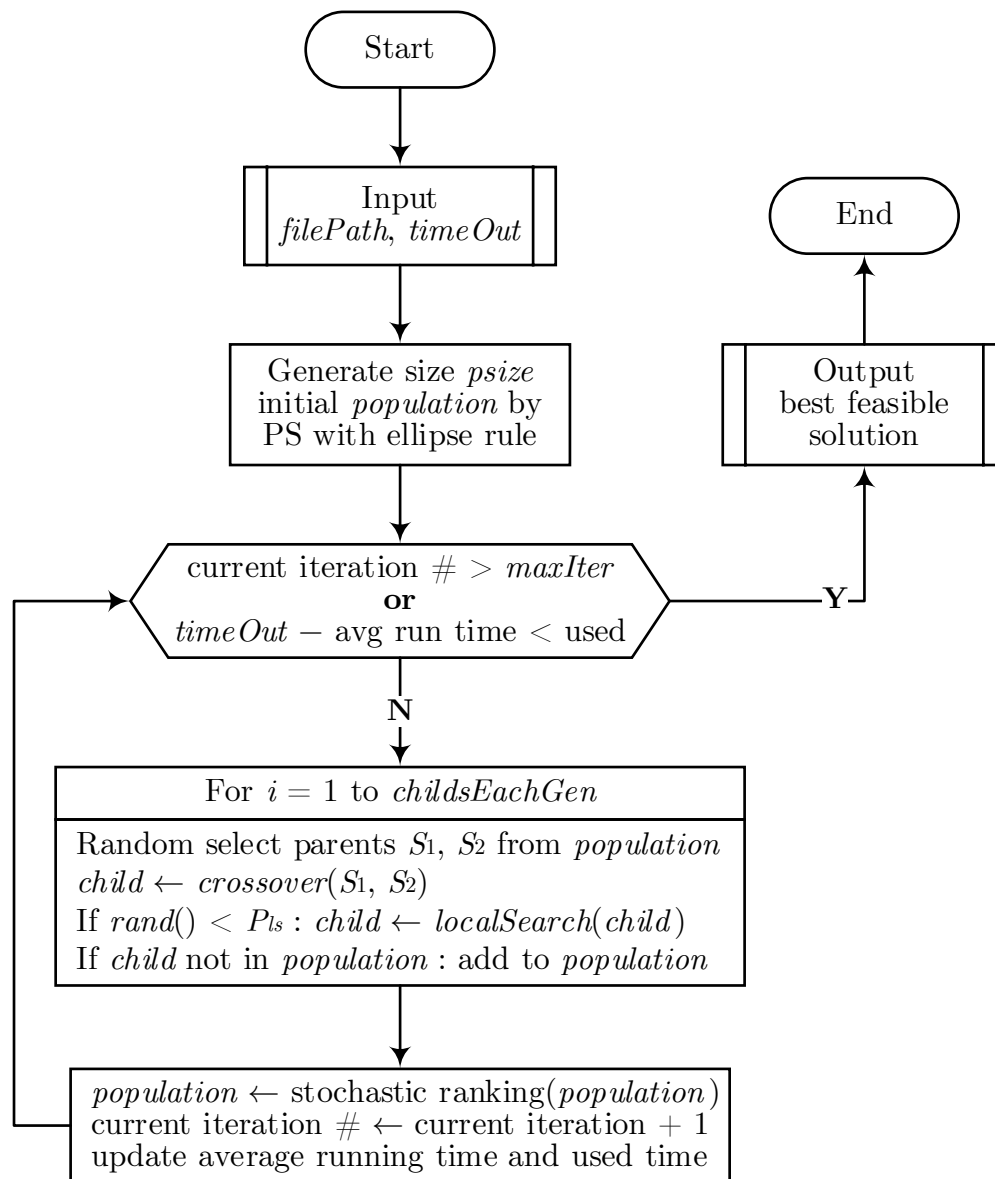
Any solution in the program is a **Solution** class whose edge representation **edgeRep** is a list of **CostList** class. An instance of **CostList** class maintains a list of **(Undirected)Edge** class—**edges** and a list of integers which represents the costs between the edges—**costs**. The **(Undirected)Edge** class holds the edge's source, target, cost and demand. And of course, the population in the MAENS is a list of **Solution**.



**Figure 1** The data structure of the program

## 2.2 Architecture

### 2.2.1 Flow Chart of Whole Program



**Figure 2** The flow chart of the whole program

### 2.2.2 Path Scanning

At first of my program, a *path scanning* (PS) heuristic with ellipse rule is applied. A traditional PS starts by initializing an empty path. At each iteration, PS finds out the tasks that do not violate the capacity constraints. If no task satisfies the constraints, it connects the end of the current path to the depot with the shortest path between them to form a route, and then initializes a new empty path. If there is(are) task(s) satisfy the constraints, the one closest to the end of the current path is chosen. If multiple tasks not only satisfy the

capacity constraints but are also the closest to the end of the current path, we may use different rules to determine which of them will be chosen.

However, unlike the traditional PS heuristic, the “ellipse rule” is defined as follows. Let  $td = \sum_{t \in E_R} demand(t)$  the total demand to be collected,  $tc = \sum_{t \in E_R} cost(t)$  the total cost assigned to edges with demand,  $\alpha$  be a real parameter, and  $le$  be the last serviced edge on the route. If the remaining capacity of the vehicle is less than or equal to the average demand on the edges,

$$Q - load(R_{\text{now}}) \leq \alpha \cdot \frac{td}{|E_R|}, \quad (2.5)$$

then the next edge to be serviced  $ne$  must be the nearest edge to  $le$  ( $le.target = ne.source$ , if the edges are adjacent) satisfying the condition:

$$sp(le, ne) + cost(ne) + sp(ne, dep) \leq \frac{tc}{|E_R|} + sp(le, dep). \quad (2.6)$$

Where  $dep$  is the depot, as mentioned above. If no feasible edge satisfies (2.6) then the vehicle returns directly to the depot. If the remaining capacity of the vehicle is large, the traditional PS heuristic will be used. [4]

### 2.2.3 Operators

The operators used by the MAENS can be categorized as the *crossover* operator and the *local search* operator. The local search operator can be divided into the *traditional local search* operators and the *merge-split* operator furtherly.

To express operators mathematically, we need to regard the solution as a vector with length  $|E_R| + p + 1$ :

$$S = (0, t_{11}, \dots, t_{1n_1}, 0, t_{21}, \dots, t_{2n_2}, 0, \dots, t_{p1}, \dots, t_{pn_p}, 0)^T. \quad (2.7)$$

Then all these operators occurred in the MAENS can be expressed as a *linear map* from  $S$  to  $S'$  and may then be written as a  $(|E_R| + p + 1) \times (|E_R| + p + 1)$  matrix as the *Linear Algebra* says. So, any operator  $\hat{O}$  acted on solution  $S$  will generate a new solution as follow:

$$\hat{O} \cdot S = S' = (0, t'_{11}, \dots, t'_{1n_1}, 0, t'_{21}, \dots, t'_{2n_2}, 0, \dots, t'_{p1}, \dots, t'_{pn_p}, 0)^T. \quad (2.8)$$

Where the dot is simply the matrix product.

### 2.2.4 Crossover Operator

In reference [3], the crossover operator for the MAENS is the *sequence-based crossover* (SBX) operator. It can be described as follow. Given two parent solutions  $S_1$  and  $S_2$ , SBX randomly selects two routes  $R_1$  and  $R_2$  from them,

respectively. Then, both  $R_1$  and  $R_2$  are further randomly split into two sub routes, say  $R_1 = (R_{11}, R_{12})$  and  $R_2 = (R_{21}, R_{22})$ . After that, a new route is obtained by replacing  $R_{12}$  with  $R_{22}$ . Finally, it is possible that some tasks appear more than once in the new route, or some tasks in  $R_1$  are no longer served in the new route. In the former case, the duplicated tasks are simply removed. In the latter case, the missing tasks are re-inserted into the new route that any other re-insertions will not induce additional cost and violation of the capacity constraints. The SBX operator can be written mathematically:

$$\widehat{SBX} = \begin{pmatrix} \text{diag}(1, \dots) & & \\ & \text{shuffle matrix}(n, n) & \\ & & \text{diag}(1, \dots) \end{pmatrix}. \quad (2.9)$$

Where one of  $S_1$  is crossed by  $S_2$  and after deletion and re-insertion, the final result is nothing more than a  $n \times n$  shuffle matrix.

However, when my program applies the result of path scanning as initial population, the SBX of any two solution will not generate an unfeasible solution since all solutions in the population is feasible. Furthermore, the result of SBX of solutions  $S_1$  and  $S_2$  obtained by the path scanning is commonly the original  $S_1$  or a small change that can be achieved by applying local search operators, which leads to a loss of long step-size search feature of a crossover operator.

Hence, I've altered the SBX a little – duplicated task which is not in  $R_{11}$  is not deleted from  $R_{22}$  but from other routes in  $S_1$ . This new SBX is able to generate unfeasible child from feasible parents and vice versa. The matrix form of the new SBX operator is too complicated to be expressed generally:

$$\widehat{SBX}_{\text{new}} \sim \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & 0 & 0 & & & \\ & & & 1 & & & \\ & & & & 1 & & \\ & & & & & 0 & 1 \\ & & & & & 1 & 0 \\ & & & & & & \ddots \end{pmatrix}. \quad (2.10)$$

Where in this case, the second route of  $S_1$  is crossed by  $S_2$  somewhat and as a result, the first route of  $S_1$  also lose an edge.

Though these matrixes are all *sparse matrix*, it is still time-inefficient to multiply the operator and the solution vector directly with sparse matrix product algorithm. Therefore, I use the data structure above and operate on it.



## 2.2.5 Local Search Operators

### 2.2.5.1 Single Insertion

In the single insertion move, an edge is removed from its current position and re-inserted into another position of the solution. Because the edge can be served in two directions, both will be considered when inserting the task into the “target position”. The matrix representation of this operator is simple:

$$\widehat{SI} = \begin{pmatrix} \text{diag}(1, \dots) & & \\ & sm \begin{pmatrix} [i+1, i]_{i=1}^n \rightarrow 1 \\ [n, 1] \rightarrow \pm 1 \end{pmatrix} & \\ & & sm([i+1, i]_{i=1}^m \rightarrow 1) \end{pmatrix}. \quad (2.11)$$

Where  $sm$  stands for sparse matrix. This matrix represents a re-insertion after the remove position. The corresponding operator with re-insertion before the remove is simply its transpose.

### 2.2.5.2 Double Insertion

The double insertion move is similar to the single insertion except that two consecutive edges are moved instead of a single one:

$$\widehat{DI} = \begin{pmatrix} \text{diag}(1, \dots) & & \\ & sm \begin{pmatrix} [i+2, i]_{i=1}^n \rightarrow 1 \\ [n-1, 2] \rightarrow \pm 1 \\ [n, 1] \rightarrow \pm 1 \end{pmatrix} & \\ & & sm([i+2, i]_{i=1}^m \rightarrow 1) \end{pmatrix}. \quad (2.12)$$

Same as single insertion, the operator with re-insert before the remove is simply the transpose. However, we can see that for any  $\widehat{DI}$ , we may conduct

$$\widehat{SI}^2 = \widehat{DI}, \quad (2.13)$$

which means that a double insertion can be replaced as two consecutive single insertion. Thus, unlike MAENS in reference [3], I will not apply the double insertion to save computation time.

### 2.2.5.3 Swap and Flip

The swap operator swaps two edges while considering the direction of serve. Its matrix representation is therefore

$$\widehat{swap} = \begin{pmatrix} \text{diag}(1, \dots) & & \\ & \begin{pmatrix} \text{diag}(1, \dots) & \pm 1 \\ \pm 1 & \text{diag}(1, \dots) \end{pmatrix} & \\ & & \text{diag}(1, \dots) \end{pmatrix}. \quad (2.14)$$

Of course,  $\forall \widehat{swap} \exists \widehat{SI}$  such that  $\widehat{swap} = \widehat{SI} \cdot \widehat{SI}^T$ . So, swap will not be applied.

The flip operator is even easier:

$$\widehat{flip} = \begin{pmatrix} \text{diag}(1, \dots) & & \\ & -1 & \\ & & \text{diag}(1, \dots) \end{pmatrix}. \quad (2.15)$$

Comparing to (2.11), we can conclude that  $\{\widehat{flip}\} \subset \{\widehat{SI}\}$  and it will not be applied consequently as well.

#### 2.2.5.4 2-Opt

There are two types of 2-opt move operators, one for a single route and the other for double routes. In the 2-opt move for a single route, a sub route (i.e., a part of the route) is selected and its direction is reversed. This type of 2-opt, by my verification, merely generates better solutions and its matrix representation can be expressed as a power of one single insertion operator:

$$\widehat{2\text{-opt}1} = \begin{pmatrix} \text{diag}(1, \dots) & & \\ & \text{diag}(-1, \dots)^T & \\ & & \text{diag}(1, \dots) \end{pmatrix} = \widehat{SI}^n, \quad (2.16)$$

and  $n$  is typically small. For this reason, 2-opt for single route is not applied. The 2-opt move of double routes is like SBX. It first selects two routes, say  $R_1$  and  $R_2$ , each one is then cut into two sub routes, say  $R_1 = (R_{11}, R_{12})$  and  $R_2 = (R_{21}, R_{22})$ . And new solutions are generated by reconnecting them:

$$\begin{cases} R_{1,\text{new}} = (R_{11}, R_{22}), R_{2,\text{new}} = (R_{21}, R_{12}) \\ R_{1,\text{new}} = (R_{11}, -R_{21}), R_{2,\text{new}} = (-R_{12}, R_{22}) \end{cases} \quad (2.17)$$

Where the minus sign represents a total reverse of the sub route (i.e., both the edges themselves and the arrangement of list are reversed). When applying this operator, the qualities of the solutions after each iteration have significantly increased, but the time cost of 2-opt can be ten times larger than the single insertion. Thus, it is not actually used at last.

#### 2.2.6 MAENS

In this project, I implement the MAENS mostly based on reference [3], however, there are still few differences. The whole procedure of my MAENS implementation can be expressed as:

- 1) Start by initializing a population using the path scan with ellipse rule.
- 2) At each iteration, two parents are selected from the population with weights reportorial to their fitness values. The SBX and the local search operator are then conducted to generate new candidate solutions, i.e., the offspring population.

- 3) After that, the parent population and offspring population are combined and individuals are sorted using *stochastic ranking* designed to deal with constrained optimization problems.
- 4) For both the initial population and all intermediate populations during the search, identical solutions are not allowed to appear in a population simultaneously, in order to maintain diversity of the population.

During the parent selection and the local search, we need a fitness function to evaluate the best solution. It must be the form of:

$$f(S) = TC(S) + \lambda \cdot TV(S). \quad (2.18)$$

where  $TV(S) = \sum_{i=1}^q \max\{0, load(R_i) - Q\}$  is total violation of the constraint.

According to [3], before the local search of  $S$ ,  $\lambda$  is initialize as

$$\lambda = \frac{TC(S_{\text{best}})}{Q} \left[ \frac{TC(S_{\text{best}})}{TC(S)} + \frac{TV(S)}{Q} + 1 \right], \quad (2.19)$$

where the  $S_{\text{best}}$  is the best feasible solution so far. The dynamic adjusts of  $\lambda$  can be seen in the pseudo code.

## 2.3 Details of Algorithms

### 2.3.1 Implementations

For all the new classes used in the program, as mentioned, are the **Solution** class, the **CostList** class and the **(Undirected)Edge** class, each `__eq__()` and `__hash__()` methods of which have been overridden to make sure that `in`, `is` and `==` keywords work properly. Specifically, the `__eq__()` of **UndirectedEdge** class is designed to test whether their costs and demands are the same while their sources and targets can be the same or inverted.

Moreover, to ensure the high computational speed during the local search with enormous insertions, a series of local update methods have been implemented in the **Solution** class and the **CostList** class. They work as follow:

- 1) In the beginning of any operator, the selected edge(s) is(are) removed from the **Solution** with `remove()`, which will call the `remove()` of **CostList**. It will only update a single element of the `costs` list corresponding to the edge after the remove.
- 2) After deciding where to re-insert, the `insert()` of **Solution**, which invokes the `insert()` of **CostList** will be used. The latter method will only re-calculate the inserted edges' `costs` list.

### 2.3.2 Pseudo Code

- **MAENS**: The main method of implementing MAENS.

```

Function MAENS (pop, psize, opsize)
1  While stopping criterion is not met do
2      popt ← pop
3      For i = 1 → opsize do
4          child ← SBX(SelectByFitness(pop, 2))
5          If rand() < Pls Then
6              child ← LocalSearch(child)
7          End if
8          If child ∉ popt Then
9              popt ← popt ∪ {child}
10         End if
11         pop ← StochasticRanking(popt)[[0: opsize]]
12     End for
13 End while
14 Return the best feasible solution in pop

```

- **MergeSplit**: a merge-split implementation, the same as [3], but the number of route chosen is always 2.

```

Function MergeSplit (solution, λ)
15 allPossibleRoutes ← Combinations(solution.edgeRep, 2)
16 allBest ← solution.copy()
17 For route1, route2 in allPossibleRoutes do
18     tasks ← Faltten(route1, route2)
19     allRoutes ← FivePathScans(tasks)
20     allRoutes ← UlusoySplit(allRoutes, 2, λ)
21     best ← best of {r ∈ allRoutes | replace route1, route2 by r form a new sol}
22     If best.fitness(λ) < allBest.fitness(λ) Then allBest ← best
23 End for
24 Return allBest

```

- **LocalSearch**: a local search implementation which calls **MergeSplit()** and **TraditionLocalSearch()**.

```

Function LocalSearch (solution, λ)
25 solution, λ ← TraditionLocalSearch(solution, λ)
26 newSolution ← MergeSplit(solution, λ)
27 If newSolution.fitness(λ) < solution.fitness(λ) Then
28     solution, λ ← TraditionLocalSearch(newSolution, λ)
29 End if
30 Return solution, λ

```

- **PathScan**: The base path scan method used in the program where input *rule* is a rule function receives multiple edges and decide which to choose.

```

Function PathScan (task, rule, sp=shortestPath, capacity, depot)
31  R ← empty list
32  While task is not empty Then
33      route ← {∅}; load ← 0; current ← depot
34      While task is not empty Then
35          candidates ← {u For u in task If u.demand + load ≤ capacity}
36          If candidates = {∅} Then Break
37          If |candidates| = 1 Then
38              currentTask ← candidates[[0]]
39              If sp(current → currentTask) + sp(currentTask → depot) >
sp(current → currentTask.inv) + sp(currentTask.inv → depot) Then
40                  currentTask ← currentTask.inv
41                  append currentTask to route
42                  load ← load + currentTask.demand
43                  remove currentTask and currentTask.inv from task
44                  Break
45              Else
46                  s ← arg mint ∈ candidates {sp(current → t), sp(current → t.inv)}
47                  If |s| = 1 Then
48                      currentTask ← s[[0]]
49                  Else
50                      currentTask ← rule(s, sp, depot, capacity, load)
51                  End if
52              End if
53              append currentTask to route
54              load ← load + currentTask.demand
55              remove currentTask and currentTask.inv from task
56          End while
57      append route to R
58  End while
59  Return R

```

- **UlusoySplitFor2Routes**: Ulusoy split of 2 routes is simple since it is already a boundary of the dynamic programming in the whole Ulusoy split.

```

Function UlusoySplitFor2Routes (flatPath, λ)
60  best ← maxi {∑ i ∈ flatPath.index|,
      Solution([CostList(flatPath[[0: i]]), CostList(flatPath[[i: end]]))}]
      with key = s → s.fitness(λ)
61  Return best

```

- **StochasticRanking**: a stochastic ranking implementation.

```

Function StochasticRanking (pop, prob=0.2)
62 Repeat |pop| times do
63   For  $i = 0 \rightarrow |pop| - 2$  do
64      $r \leftarrow \text{rand}(); b_1 \leftarrow pop[i].breachScore; b_2 \leftarrow pop[i+1].breachScore$ 
         $f_1 \leftarrow pop[i].cost; f_2 \leftarrow pop[i+1].cost$ 
65     If  $b_1 = b_2 = 0$  or  $r < prob$  and  $f_1 > f_2$  Then
66       swap pop[i], pop[i + 1]
67     Else if  $b_1 > b_2$  Then
68       swap pop[i], pop[i + 1]
69     End if
70   End for
71 End repeat
72 Return pop

```

- The **UlusoySplit()** method select all consecutive routes and flatten them before invoking **UlusoySplitFor2Routes()** the for each of them. It is so simple that I will not write it down.
- The **FivePathScans()** method calls the **PathScan()** for five times with five distinct rules. It is also so simple that I will not write it down.
- The **PathScanOfEllipseRule()** method used for population initialization is basically the same as the **PathScan()** method but with the rule stated in 2.2.2. So, I will not write it down.

### 3 Performances & Time Control

In this part, performance test of this program as well as the time control approach will be implemented. As mentioned above, the testing platform is *Windows* 10 with Intel<sup>®</sup> Core<sup>™</sup> i7-8700K @ 3.70~4.70GHz.

#### 3.1 Theoretical Performance & Time Control

As in reference [3], the most time-consuming procedure of MAENS is the local search part, especially the traditional local search whose iterations try to find the most significant improvement. And according to my own practices, the time cost of traditional local search can be 5 times larger than the merge-split, let alone the SBX. Therefore, we may consider the time complexity of traditional local search alone when computing the theoretical performance.

Let  $n = |E_R|$  the number of required edges, it is not hard to prove that each traditional local search has a time complexity of

$$T(\widehat{SI}) = T(\widehat{2\text{-opt}}) = \Theta(n^2) + \Theta(n) = \Theta(n^2), \quad (3.1)$$

where the operation is a remove & a re-insert. The second term is the cost of selecting best solution. However, this operation itself has a time complexity of

$$T(\text{non-atom operation}) = \Theta\left(\frac{n}{N_V}\right), \quad (3.2)$$

where  $N_V$  is the vehicles we have in the CARP.

Let  $l$  be the maximum iteration index of traditional local search, we have

$$T(\text{local search}) = O\left(l \frac{n^3}{N_V}\right), \quad (3.3)$$

Hence, letting the  $m$  be the maximum iteration index of MAENS, and typically choosing  $l \sim N_V$ , the total time complexity becomes

$$T(\text{all}) = O\left(\frac{lmn^3}{N_V}\right) \sim O(mn^3). \quad (3.4)$$

With this guidance that the time cost of each iteration is approximately proportional to the cubic of the amount of required edges, we can apply a precise time control by average true time cost and the initial estimation of it:

$$T_{\text{one iteration},0} = \gamma \cdot P_{ls} \cdot (\text{childs each generation}) \cdot |E_R|^3 \cdot \frac{f_{\text{base}}}{f_{\text{CPU}}}, \quad (3.5)$$

where  $\gamma = 9.34776 \times 10^{-6}$ ,  $f_{\text{base}} = 4400$  and  $f_{\text{CPU}}$  are the time of a single local search operation on test platform, the base CPU frequency where the test performs and the CPU frequency that the program runs at, respectively.

## 3.2 Empirical Performance

### 3.2.1 Platform Test

SubmitTime	Info	ExitCode	RunTime(s)	Cost	Dataset
11-14 16:41:45	Solution Accepted.	0	13.83	173	val1A
11-14 16:39:34	Solution Accepted.	0	52.37	279	val7A
11-14 16:49:50	Solution Accepted.	0	118.55	5262	egl-s1-A
11-14 16:45:28	Solution Accepted.	0	70.20	3548	egl-e1-A
11-14 16:44:56	Solution Accepted.	0	5.32	275	gdb10
11-14 16:44:14	Solution Accepted.	0	4.82	316	gdb1
11-14 16:42:44	Solution Accepted.	0	56.53	402	val4A

	Rank	User	submitTime	time	cost
	#1	11410151	11-14 16:39:34	52.368	279
	#2	11612405	11-14 10:59:31	59.607	279
val7A	#1	11410151	11-14 16:49:50	118.554	5262
egl-s1-A	#2	11610914	11-13 22:35:01	119.538	5344
	#1	11611908	11-13 22:28:54	1.626	316
gdb1	#2	11410151	11-14 16:44:14	4.820	316
val1A	#3	11612110	11-14 09:02:35	20.835	316
gdb10	#1	11410151	11-14 16:41:45	13.826	173
	#2	11611002	11-13 21:59:37	48.347	173
egl-e1-A	#1	11410151	11-14 16:44:56	5.316	275
	#2	11611002	11-13 21:55:39	39.504	275
val4A	#1	11410151	11-14 16:45:28	70.198	3548
	#2	11612405	11-14 12:41:09	119.653	3626
	#1	11410151	11-14 16:42:44	56.530	402
	#2	11611002	11-13 22:13:09	54.932	408

Figure 3 Rank of CARP platform, retrieved on Nov 14<sup>th</sup>, 2018

### 3.2.2 Sample Test

Along with the output formulations, 7 distinct samples are provided as well. The names and the best costs have ever reached in the previous published researches (a.k.a. lower bounds or for short LB) of these samples and other used samples are:

Name	LB	Name	LB	Name	LB
egl-e1-A	3548	egl-e1-B	4498	val4D	526
egl-s1-A	5018	egl-e1-C	5566	val5D	573
gdb1	316	egl-e2-A	5018	val9D	385
gdb10	275	egl-e2-B	6305	val10D	525
val1A	173	egl-e2-C	8243	val1C	245
val4A	400	egl-s1-B	6384	val4C	428
val7A	279			val7C	334

**Table 1** The name and the best cost have ever reached (LB) of provided samples and other used samples. The *gdb* dataset is so simple that it is not further tested.



Using my program which implemented an altered MAENS (Memetic Algorithm with Extended Neighborhood Search) with 8 process parallel and a fixed initial random seed 123554 (a randomly selected integer by myself) to deal with the CARP (Capacitated Arc Routing Problem) described in the 7 samples, I am able to make the chart below.

	<i>Timeout:</i> 60 s		<i>Timeout:</i> 150 s		<i>Timeout:</i> 300 s		<i>Timeout:</i> 600 s		<i>LB</i> ( <a href="#">[3]</a> )
<i>Before best</i>	<i>n=2</i>	<i>t=40s</i>	<i>n=2</i>	<i>t=40s</i>	<i>n=2</i>	<i>t=40s</i>	<i>n=2</i>	<i>t=40s</i>	N.A.
egl-e1-A	3548		3548		3548		3548		3548
<i>Before best</i>	<i>n=1</i>	<i>t=48s</i>	<i>n=3</i>	<i>t=140s</i>	<i>n=8</i>	<i>t=246s</i>	<i>n=34</i>	<i>t=1k s</i>	<i>n~500</i>
egl-s1-A	5327		5201		5116		5018		5018
<i>Before best</i>	<i>n=1</i>	<i>t=3.5s</i>	<i>n=1</i>	<i>t=3.5s</i>	<i>n=1</i>	<i>t=3.5s</i>	<i>n=1</i>	<i>t=3.5s</i>	N.A.
gdb1	316		316		316		316		316
<i>Before best</i>	<i>n=1</i>	<i>t=3.5s</i>	<i>n=1</i>	<i>t=3.5s</i>	<i>n=1</i>	<i>t=3.5s</i>	<i>n=1</i>	<i>t=3.5s</i>	N.A.
gdb10	275		275		275		275		275
<i>Before best</i>	<i>n=1</i>	<i>t=7.2s</i>	<i>n=1</i>	<i>t=7.2s</i>	<i>n=1</i>	<i>t=7.2s</i>	<i>n=1</i>	<i>t=7.2s</i>	N.A.
val1A	173		173		173		173		173
<i>Before best</i>	<i>n=2</i>	<i>t=57s</i>	<i>n=2</i>	<i>t=57s</i>	<i>n=2</i>	<i>t=57s</i>	<i>n=2</i>	<i>t=57s</i>	N.A.
val4A	400		400		400		400		400
<i>Before best</i>	<i>n=1</i>	<i>t=25s</i>	<i>n=1</i>	<i>t=25s</i>	<i>n=1</i>	<i>t=25s</i>	<i>n=1</i>	<i>t=25s</i>	N.A.
val7A	279		279		279		279		279

**Table 2** The benchmark result. The *Before best* rows indicate when the best result under the timeouts will be obtained, specifically, after how many iterations  $n$  in MANES and the time cost  $t$  of these iterations. Note that since the egl-s1-A problem requires longer time to find the LB solution, the actual timeout of it is set to 1200 s.

Then, the stabilities and the best performances of all samples are focused instead. Therefore, I used 6 process parallel to match the core number and opened the early return functionality to observe the best my program can reach. The used 10 random seeds are 100, 200, ..., 1000.

<i>Name</i>	<i>Best of</i> 60 s		<i>Average</i> $\pm Std, NB$		<i>Best of</i> 600 s		<i>Average</i> $\pm Std, NB$		<i>Best</i> ( <a href="#">[3]</a> )	<i>Average</i> $\pm Std$ ( <a href="#">[3]</a> )
egl-e1-A	3548	$t=9s$	3549	$\pm 1,9$	3548	$t=9s$	3548	$\pm 0$	3548	3548 $\pm 0$
egl-s1-A	5249	$t=ro$	5292	$\pm 29,0$	5018	$t=550s$	5079	$\pm 30,2$	5018	5040 $\pm 36,6$
gdb1	316	$t=0.3s$	316	$\pm 0,10$	316	$t=0.3s$	316	$\pm 0,10$	316	316 $\pm 0,10$
gdb10	275	$t=0.4s$	275	$\pm 0,10$	275	$t=0.4s$	275	$\pm 0,10$	275	275 $\pm 0,10$
val1A	173	$t=0.5s$	173	$\pm 0,10$	173	$t=0.5s$	173	$\pm 0,10$	173	173 $\pm 0,10$
val4A	400	$t=11s$	401	$\pm 1,4$	400	$t=11s$	400	$\pm 0,10$	400	400 $\pm 0,10$
val7A	279	$t=3s$	279	$\pm 0,10$	279	$t=3s$	279	$\pm 0,10$	279	316 $\pm 0,10$

<i>Name</i>	<i>Best of</i> 60 s	<i>Average</i> $\pm Std, NB$	<i>Best of</i> 600 s	<i>Average</i> $\pm Std, NB$	<i>Best</i> ( <a href="#">[3]</a> )	<i>Average</i> $\pm Std$ ( <a href="#">[3]</a> )
egl-e1-B	4538 $t=ro$	4547 $\pm$ 8,0	4498 $t=325s$	4506 $\pm$ 10,6	4498	4517 $\pm$ 18,4
egl-e1-C	5765 $t=ro$	5812 $\pm$ 40,0	5613 $t=ro$	5620 $\pm$ 11,0	5595	5602 $\pm$ 10,6
egl-e2-A	5027 $t=ro$	5080 $\pm$ 43,0	5018 $t=255s$	5018 $\pm$ 0,10	5018	5018 $\pm$ 0,10
egl-e2-B	6397 $t=ro$	6451 $\pm$ 30,0	6342 $t=ro$	6346 $\pm$ 4,0	6317	6341 $\pm$ 12,2
egl-e2-C	8665 $t=ro$	8699 $\pm$ 36,0	8395 $t=ro$	8420 $\pm$ 25,0	8335	8356 $\pm$ 36,7
egl-s1-B	6722 $t=ro$	6799 $\pm$ 34,0	6435 $t=ro$	6472 $\pm$ 30,0	6388	6433 $\pm$ 9,1
val4D	546 $t=ro$	550 $\pm$ 3,0	536 $t=ro$	539 $\pm$ 3,0	530	533 $\pm$ 3,5
val5D	597 $t=ro$	602 $\pm$ 3,0	585 $t=ro$	590 $\pm$ 2,0	577	583 $\pm$ 2,7
val9D	411 $t=ro$	414 $\pm$ 2,0	397 $t=ro$	400 $\pm$ 1.5,0	391	391 $\pm$ 0,10
val10D	544 $t=ro$	546 $\pm$ 2,0	531 $t=590s$	536 $\pm$ 3,1	531	534 $\pm$ 2,0
val1C	260 $t=ro$	265 $\pm$ 4,0	255 $t=ro$	257 $\pm$ 2,0	245	245 $\pm$ 0,10
val4C	440 $t=ro$	443 $\pm$ 3,0	428 $t=550s$	434 $\pm$ 5,2	428	431 $\pm$ 3,5
val7C	335 $t=ro$	338 $\pm$ 3,0	334 $t=72s$	334 $\pm$ 0,10	334	334 $\pm$ 0,10

**Table 3** The stabilities and the best performances of all samples where  $t=ro$ , *Std* and *NB* stands for running out of time, standard error and the number of runs that reaches the LB in 10 runs, respectively. Note that *Best* ([\[3\]](#)) column is the best result obtained by [\[3\]](#), with fixed max iteration number 500.

### 3.3 Analysis

You may find out that few of my result is poorer than the original one in [\[3\]](#), but notice that in [\[3\]](#), a fixed amount of iterations – 500 is used instead of a timeout. Unlike the *C* implementation in [\[3\]](#), when using this implementation by *Python*, it is nearly impossible to wait 500 iterations since each one takes tens of seconds. At this rate, my program is not worse than the one in [\[3\]](#).

It is also obvious that the convergence speed is significantly improved by changing the population initialization method from random generate used in reference [\[3\]](#) to the path scanning with ellipse rule under different random seeds, the SBX parent selection heuristic, and altering the SBX operator to make it able to generate unfeasible child from feasible parents. On the provided samples, we can imply that the CARP with lower true best cost tends to converge quicker. But even when it is large, my implementation has improvement each few iterations. On most test samples, the results converge to LB of MAENS after  $\sim 100$  iterations (not shown in tables), much faster than [\[3\]](#). I may say that this program of solving CAPR has bring about the desired result.

## Further Discussion

Like reference [5], we can also apply heuristics for the local search probability  $P_{ls}$ . Unfortunately, due to the low performance of raw *Python*, this better but more time-consuming method may not achieve same goal within same timeout, thus I did not implement it in this project. But for further usage, it may be better choices.

## Acknowledgment

I would like to thank Prof. *Tang Ke* for his brilliant paper which inspired me to implement this MAENS. Also, I want to thank TA *Zhao Yao* for her guidance.

## References

- [1] Corberán, Á., & Laporte, G. (Eds.). (2013). *Arc routing: problems, methods, and applications*. Society for Industrial and Applied Mathematics.
- [2] H. A., Eiselt; Michel, Gendreau (1995). "Arc Routing Problems, Part II: The Rural Postman Problem". *Operations Research*. 43 (3): 399–414. Retrieved from [https://en.wikipedia.org/wiki/Arc\\_routing](https://en.wikipedia.org/wiki/Arc_routing) on Oct 27<sup>th</sup>.
- [3] Tang, K., Mei, Y., & Yao, X. (2009). Memetic algorithm with extended neighborhood search for capacitated arc routing problems. *IEEE Transactions on Evolutionary Computation*, 13(5), 1151-1166.
- [4] Santos, L., Coutinho-Rodrigues, J., & Current, J. R. (2009). An improved heuristic for the capacitated arc routing problem. *Computers & Operations Research*, 36(9), 2632-2637.
- [5] Jia, T. (2016). Memetic approach to capacitated arc routing problem. *Beijing Jiaotong University*.