CARP Report

HE Wanning 11912936
Department of Computer Science and Engineering
Southern University of Science and Technology
11912936@mail.sustech.edu.cn

1. Preliminaries

1.1. Problem Description

Capacitated Arc Routing Problem (CARP) is a well known combinatorial problem that requires the identification of the minimum total distance travelled by a set of vehicles to service a given set of roads subject to the vehicle's capacity constraints. Contrary to the well-known Vehicle Routing Problem (VRP), in which goods must be delivered to client nodes in a network, CARP consists of visiting a subset of edges.

1.2. Problem Applications

CARP has applications in areas such as household refuse collection. street sweeping. inspection of electrical and pipeline networks for faults, mail delivery, etc. Our goal is to finished these tasks with reasonable financial and temporal cost. Therefore, research about CARP can serve as a guide in these real-world problems.

2. Methodology

In this section, we will introduce the notation that used in our report, data structures that we store related information, project's module design as well as algorithms.

2.1. Notations

Notations used in this report are shown in TABLE 1.

TABLE 1. NOTATIONS AND THEIR MEANING.

Notation	Meaning
T(i, j)	A required edge from node i to node j
S	A CARP solution
R	A route in a CARP solution S
Min_Dist(i, j)	Minimum cost travel from node i to node j
C$(Task)$	Cost for a task
L$(Edge)$	Load for an edge
Q(S)	Quality (i.e. total cost) of solution S
EXL(S)	Exceed loads of a solution S
f(S)	Fitness of a solution S
$psize$	Population size
$offsize$	Offspring population size ($6 * psize$ specifically)

2.2. Model Design

We divide the program into three parts:

1) Problem Formulate:
CARP can be represented as a graph $G = (V, E, A)$, seeking a minimum cost routing plan for vehicles to serve all the required edges (tasks) $E_R \subseteq E$ and required arcs $A_R \subseteq A$, subject to a capacity constraint for each route. We represent solution S of CARP as $S = (T1, T2, \ldots, Tn)$ where Ti for $i \in 1, 2, \ldots n$ is a task (required edge) in CART. Quality of a solution can be written as

$$Q(S) = \sum_{i=1}^{n-1} [Cost(Ti) + Min\_Dist(Ti, Ti + 1)]$$

From another point of view, each solution S is consist of multiple routes each of which starts from the depot and ends at the depot. Hence, S can also be written as

$$S = (R1, R2, K, Rm)$$

$$= (0, (R_{11}, R_{12}, K), 0, (R_{21}, R_{22}, K), \ldots, (R_{m1}, R_{m2}, K), 0)$$

where m is the number of routes in S and each Ri denotes a single route. Obviously, every Ri consists of a subsequence of tasks, and the load (i.e., total demand) of it is

$$Load(Ri) = \sum_{k=1}^{length(Ri)-1} demand(R_{ik})$$

Thus, the constraint can be represented as

$$\forall i (i = 1, 2, \ldots n), Load(Ri) \leq Capacity$$

2) Problem Solving:
We need a *CARP_solver* to read CARP problems' data from user, timing the program execution time and calculate the terminate status. It also also call the service from *CARP_ai* to get the solution.
CARP_ai is where we implement MAENS algorithm, who gets the CARP problem information and return the current best feasible solution to *CARP_solver*. We store all related information like Task in *CARP_info*, and pass their instances as

parameter between *CARP_solver* and *CARP_ai*. *CARP_solver* instantiate a *CARP_ai* instance and pass related information to it. *CARP_ai* implement the solving logit and return the most desirable solution back to *CARP_solver*. *CARP_solver* start timing as it is reading from CARP data and calculate the duration of each iteration that the *CARP_ai* takes to get the most feasible solution. Average time for one generation is computed in defense of time out.

3) Data Structure: We use a list of Task instance to store the information of each class and set the list index of them as their ID. ID is used rather than the Task instances inside *CARP_ai*. We use $Task[ID]$ to get the Task instance and read their features such as the from-node, the to-node, cost and load. ID of tasks are stored in numpy array in implementation, in order to speed up our program. Minmum distance is also stored in numpy array. Min_Dist(u, v) implies the minimum distance between node u and node v nodes.

2.3. Algorithm Detail

We typically implement MAENS introduced by Tang Ke, Yao Xin and Mei Yi in 2009 [1] in our program. The algorithm can be divided into three parts:

- Population initialization,
- Offspring generation,
- Selection for next generation.

The general procedure for MAENS is shown in Algorithm 1:

Algorithm 1 Random Scanning

Input: A set of required edges $Task$
Output: A feasible solution S
  $Population \leftarrow$ Initialize($Task, Capacity$)
  for $Gmax$ iterations **do**
    $Offsprings \leftarrow anEmptyList$
    while Off_Size offsprings are generated **do**
      $S1, S2 \leftarrow$ two different solutions in $Population$
    $Offsprings \leftarrow Offsprings \cup Gen\_OffSpring$(S1, S2)
    end while
    $Population \leftarrow$ Selection($Population, Offsprings$)
    $S \leftarrow$ Solution with smallest $Q(S)$ in Population
  end for
  return S

2.3.1. Population Initialize. We use iterative Random scanning (Algorithm 2) method to obtain a set of feasible solutions which forms the initial generation of MAENS. In each iteration, an arbitrary unserved task is selected and append it to current route. If there is no valid task (i.e., a task with load within constraint), we allocate a new route and do the previous steps repeatedly. Algorithm terminates when all required edges are served. We eventually get a solution represented by a set of task IDs stored in 2-D numpy array.

Algorithm 2 Random Scanning

Input: A set of required edges $Task$
Output: A feasible solution S
  $Unserved\_Tasks \leftarrow Tasks$
  $S \leftarrow anEmptyList$
  for move **do**
    $Route \leftarrow anEmptyList$
    $Curr\_Task \leftarrow arandomtaskinUnserved\_Tasks$
    $Route \leftarrow Route \cup Curr\_Task$
    for move in $Unserved\_Tasks$ sub **do**
      $Curr\_Task \leftarrow arandomtaskinUnserved\_Tasks$
      $Route \leftarrow Route \cup Curr\_Task$
    end for
    $S \leftarrow S \cup Route$
  end for
  return S

2.3.2. Offspring Generate. Offspring generation plays a key role in MAENS, which can be further divided into two parts:

- Apply sequence-based crossover (SBX) to get an offspring Sx,
- Local search around Sx with probability P_{ls}. This include three traditional move operators Single Insertion, Double Insertion and Swap, as well as Merge-Split (M-S) operator.

The overall procedure to generate a set of offsprings is displayed in Algorithm 3.

Algorithm 3 Gen_Offspring

Input: Individuals $S1, S2$
Output: New solution S_{new}
  $Sx \leftarrow$ SBX($S1, S2$)
  Generate a random number r where $r \in [0, 1)$
  if $r < P_{ls}$ **then**
    $S_{ls} \leftarrow$ LocalSearch(Sx)
    return S_{ls}
  end if
  return S_x

1) Sequence-Based Crossover At each iteration of MAENS, crossover is implemented by applying the sequence-based crossover (SBX) operator to two parent individuals $S1, S2$ randomly selected from the current population. Each pair of parent individuals leads to a single offspring individual Sx after SBX.

Given two parent solutions S1 and S2, SBX randomly selects two routes R1 and R2 from them, respectively. Then, both R1 and R2 are further randomly split into two subroutes, say $R1 = (R_{11}, R_{12})$ and $R2 = (R_{21}, R_{22})$.

new route is obtained by the combination of one subroute from $R1$ and another from $R2$. Suppose $Rx =$

(R_{11}, R_{22}), then Sx is obtained by replacing R_{12} by R_{22} from $S1$. Since the brute replacement will lead to duplicated tasks and unserved tasks, we remove duplicated tasks from Sx in the former case and insert unserved tasks into Sx.The re-insertions may induce additional cost and violation of the capacity constraints. Hence, each missing task is re-inserted into such a position that re-insertion into any other position will not induce both lower additional cost and smaller violation of the capacity constraints. If multiple positions satisfy this condition, one of them will be chosen arbitrarily. [1]

Algorithm 4 SBX

Input: Individuals $S1, S2$
Output: Individual Sx
$R1 \leftarrow anarbitraryrouteinS1$
$R2 \leftarrow anarbitraryrouteinS2$
$(R_{11}, R_{12}) \leftarrow$ Random_Split($R1$)
$(R_{21}, R_{22}) \leftarrow$ Random_Split($R2$)
$Rx \leftarrow (R_{11}, R_{22})$
$Sx \leftarrow (S1 \setminus R1) \cup Rx$
for $task_i$ in R_{22} **do**
  for $task_i$ in $Rx \setminus R_{22}$ **do**
    if $task_i$ equals $task_j$ **then**
      Delete $task_j$ from Sx
    end if
  end for
end for
for $task \in R_{12}$ and $task \notin R_{22}$ **do**
  if There is vaild Rn **then**
    $Rn \leftarrow anarbitraryvalidrouteinSx$
    $Rn \leftarrow Rn \cup task$
  else
    $task$ insert as a new route
  end if
end for
return Sx

2) Local Search

One major difference between Memetic Algorithms (MAs) and conventional Evolutionary Algorithms (EAs) is that the mutation operators of EAs are replaced by local search in MAs. From the evolutionary computation perspective, MAs can be viewed as a form of population-based EAs hybridized with individual learning procedures that are capable of performing local refinements [1].

In vehicle routing problem (VRP), there are four commonly used move operators [1], three of which can be directly applied in CARP problem whose solution is encoded as a sequence of tasks' ID without loss of generality:

- Single Insertion: We remove an arbitrary task from one route and re-insert it into another route or a new route,
- Double Insertion: Similarly, we select two consecutive tasks and perform what we do in single insertion,
- Swap: Two candidate tasks are selected and their positions are exchanged,
- 2-Opt: A move operator that is only applicable to edge tasks, so we do not implement it in CARP problem.

We repeatedly do traditional local search to generate new solution until the new solution can not be improved. Suppose the move operator we apply currently is Move, the pseudo code of Traditional Local Search is as follow:

Algorithm 5 Traditional Search

Input: Current solution S, Traditional operator $Move$
Output: New solution S_{ls}
$S_{prev} \leftarrow S$
while $f(S_{new}) < f(S_{prev})$ **do**
  Perform $Move$ on S_{prev} to obtain a new solution S_{new}
  $S_{prev} \leftarrow S_{new}$
end while
$S_{ls} \leftarrow S_{prev}$
return S_{ls}

As we introduced in Notations, $f(S)$ implies the fitness of a solution S. Since we are very likely to generate invalid (i.e. overloaded routes) solutions in traditional search step, we use Fitness rather than Quality to imply the goodness of a solution. We evaluating the fitness of a solution by a weighted sum of the extent it violates the constraints and its cost:

$$f(S) = Q(S) + \lambda * EXL(S)$$

where $Q(S)$ is the total cost of solution S and $EXL(S)$ is how much it violates the constraints. λ is a penalty parameter that balances the trade off between cost and violation. It initialized as

$$\lambda = \frac{Q(S_{best})}{Capacity} * (\frac{Q(S_{best})}{Q(S)} + \frac{EXL(S)}{Capacity} + 1)$$

S_{best} represents the best feasible solution found so far. The term "1" is included to ensure a sufficiently large λ in case S is a feasible solution with very large cost.

In the main loop of traditional search, λ will become twice if we cannot reach a feasible solution for five consecutive iterations, and will be halved if we can find feasible solutions in five consecutive iterations. This implies that λ should decrease with the total cost of the current solution while increase with the total violation [1].

Traditional operators adopt rather simple schemes to generate new solutions and thus are likely to generate new solutions that are quite similar to the current solutions. Therefore, Merge-Split (MS) operator whose search step size is larger is introduced in MAENS in order to find the global optimal. MS operator is composed of two components, i.e., the Merge and the Spilt. Given a solution, the Merge component randomly selects p ($p > 1$) routes of it and combines them together to form an unordered list of task IDs, which contains all the tasks of the selected routes. The Split component directly operates on the unordered list generated by the Merge component.

We first apply path scanning (PS) heuristic for tasks from p routes ($p = 2$ is recommanded) in the current solution S

to generate new sequence of tasks, i.e., some new routes. PS starts by initializing an empty path. At each iteration, PS finds out the tasks that do not violate the capacity constraints. Compared with what we have done in random scanning, path scanning adopt five rules to determine what task is desirable when there are multiple tasks satisfying the capacity constraint. Rules for tasks selection are as follow [1]:

1) maximize the distance from the head of task to the depot,
2) minimize the distance from the head of task to the depot,
3) maximize the term $dem(t)/sc(t)$, where $dem(t)$ and $sc(t)$ are demand and serving cost of task t, respectively;
4) minimize the term $dem(t)/sc(t)$,
5) use rule 1) if the vehicle is less than half-full, otherwise use rule 2)

PS does not use the five rules arbitrarily. Instead, it scans the unordered list of unserved tasks for five times. In each scan, only one rule is used. Hence, PS will generate five ordered lists of tasks in total. (Since pseudo code of PS is provided in Lab, we do not write it again in our report)

In the Split component, PS is followed by Ulusoy's splitting [2] procedure for further improvement.

Overall, the MS operation generate new solution as follow:

Algorithm 6 MS Search

Input: Current solution S
Output: New solution S_{ls}
  $New\_Solutions \leftarrow$ an empty list
  for $Routes$ is a combination of p routes from S **do**
    $Tasks \leftarrow$ set of tasks in $Routes$
    $M\_Routes\_List \leftarrow$ PS($Tasks$)
    $S\_Routes\_List \leftarrow$ Ulusoy(M_Routes_List)
    $Best\_Routes \leftarrow$ The routes with minimum $Quality$
    $S_{new} \leftarrow (S \setminus Routes) \cup Best\_Routes$
    $New\_Solutions \leftarrow New\_Solutions \cup S_{new}$
  end for
  return Solution S_{ls} whose $Quality$ is the smallest in $New\_Solutions$

Suppose there are m routes in the current solution S. If m is large, the number of combinations C_m^p would be really huge. Since PS and Ulusoy's Split procedure are time-consuming, we would not like to enumerate all combinations of p routes in S. According to Tang and Yao's paper, We restrict the MS operator to generate 100 solutions at most for one time.

The overall procedure for local search shows in Algorithm . Note that we apply traditional search twice if the solution obtained by MS operator is better than the original solution.

2.3.3. Selection. After obtaining sufficient number of offspring in the previous steps, the parent and offspring populations are combined to do Stochastic Ranking. The best $psize$ solutions after ranking are selected to form the population of next generation. Similar to how we evaluate a solution in traditional search, we use fitness $f(S)$ to determine whether it is better.

Stochastic ranking sorts individuals through a bubble-sort-like procedure: Each pair of adjacent individuals is compared and their ranks will be swapped if the individual with higher rank is better. If the two compared individuals are both feasible, comparison will be made solely according to the fitness. Otherwise, the two individuals will be compared either according to their constraint violations, with a predefined probability P_f [1].

Algorithm 7 Local Search

Input: Current solution S
Output: New solution S_{ls}
  $Tradition\_Solutions \leftarrow$ an empty list
  for Move in Traditional Moves **do**
    $S_{new} \leftarrow$ TraditionalSearch($Tasks, Move$)
    $Tradition\_Solutions \leftarrow Tradition\_Solutions \cup S_{new}$
  end for
  $MS\_Solutions \leftarrow$ an empty list
  for $S_traditional$ in $Tradition\_Solutions$ **do**
    $S_{new} \leftarrow$ MSSearch($S_{traditional}$)
    $MS\_Solutions \leftarrow MS\_Solutions \cup S_{new}$
  end for
  $S_{best} \leftarrow$ Solution with smallest quality in $MS\_Solutions$
  if S_{best} is better than S **then**
    Do traditional search around S_{best} again
    Obtain new S_{best}
  end if
  return S_{best}

Algorithm 8 Stochastic Ranking

Input: Combined population $Population$
Output: Sorted $Population$
  for Consecutive solutions $S1, S2$ in $Population$ **do**
    Generate a random number r where $r \in [0, 1)$
    if $S1, S2$ are feasible and $Q(S1) > Q(S2)$ **then**
      :Swap $S1, S2$
    else if $EXL(S1) > EXL(S2)$ **then**
      SwAP $S1, S2$
    end if
  end for
  return $Population$

3. Empirical Verification

In this section, we demonstrate the experimental results of our program.

3.1. Dataset

We implement MA solver (without MS operator) and MAENS solver. These solvers are tested with several CARP

samples: (i) gdb20, (ii) val9D, (iii) egl-e2-B, (iv) egl-s3-A

TABLE 2. INFORMATION ABOUT EACH TEST DATA

NAME	$\|V\|$	$\|E_{req}\|$	$\|E_{nreq}\|$	—|E| Capacity	Time
gdb20	11	22	0	—4 27	10s
val9D	50	92	0	—10 70	30s
egl-e2-B	77	72	26	—10 200	120s
egl-s2-A	140	147	43	—15 235	300s

3.2. Performance

Our work is implemented in Python3.9 with integrated development environment Pycharm. The main testing platform is Windows 10 with Intel® Core™ i7-9750H @ 2.59GHz of 6 cores and 12 threads.
For experiment, we pass a same termination time and a same seed for each solver engine and compare their best result of sample problems. We also output the running log of them in each iteration to see how fast they reach their best solution.

3.3. Hyperparameters

Most parameters like probability of conducting local search (P_{ls}), size of offspring population ($offsIze$), the number of routes p in MS operator etc., whose functionalities have been proven in MAENS's paper are adopted directly. Two hyperparameters (i) Population size $psize$, (ii) Set of traditional operators which have not been measured in Yao's paper will be tested in our research.

3.4. Experimental results

The following tables show performance among different searching strategies:

TABLE 3 demonstrates the results of two solvers with $psize = 30$, $p = 2$, traditional operators = Single Insertion, Double Insertion, Swap:

TABLE 3. QUALITY FOR EACH PROBLEM.

Problem	MA	MAENS
gdb20	123	121
val9D	502	412
egl-e2-B	6948	6450
egl-s2-A	11789	10451

TABLE 4 displays the number of iterations two solvers runs. Every iteration, a new solution is generated through their own local search methods (and crossover for MAENS):

TABLE 4. NUMBER OF ITERATIONS IN 180 SECONDS

180s(egl-e2-B)	EA	MAENS
total iteration	7141	1263

TABLE 5. QUAILTY OF SOLUTION GENERATED BY MAENS WITH DIFFERENT $psize$.

NAME	$psize = 20$	$psize = 30$
val9D	412	416
egl-e2-B	6450	6436

TABLE 5 compare the result for MAENS solver with $psize = 20$ and $psize = 30$ respectively.

TABLE 6 compare the result for MAENS solver with different traditional operator set. ($psize = 30$ for all)

TABLE 6. QUAILTY OF SOLUTION GENERATED BY MAENS WITH DIFFERENT OPERATOR SETS.

NAME	All	$\backslash Swap$	$\backslash Double - Insert$	$\backslash Single - Insert$
val9D	416	432	419	422
egl-e2-B	6436	6440	6441	6479

3.5. Conclusion

From the experimental result in TABLE 3 and 4 we can conclude that MAENS solver works significantly better than EA solver especially in CARP with great amount of tasks, though MS operator is time-consuming (shown in TABLE 5).

Also, from TABLE 6 we know that MAENS with smaller population size can find a fair solution in shorter time, but much easier to be trapped in local best solution (i.e., loss in longer time). Therefore, we check the running time in our project: if it is longer than 300 seconds, we set $psize = 30$, otherwise $psize = 20$.

TABLE 7 compared the results of MAENS with different traditional move operator sets. The result shows that Single Insertion is the most effective operator among three, and without any of the traditional operators, the performance of MAENS would be worse.

3.5.1. Advantages. We completely implement MAENS in this project and explored the functionalities of parameters that have not been measured in MAENS's paper. Apart from implementation, we also adjust the population size dynamically according to termination time in order to reach a better solution in a limited time.

3.5.2. Disadvantages. Although MS operator plays a key role in local search progress (as we measured before), it takes too much time that the program only runs 190+ iterations in egl-s2-A test, which is one generation only. Insufficient generation would also lead to an unfavorable solution.
Also, our MAENS solver converge into a local maximum solution fast and we cannot reach a solution of quality less than 5100 in egl-s1-A problem.

3.5.3. What we learned and what to achieve. Implement MAENS is a hard process, we learn a lot from the paper. As is said in Disadvantages, it is desirable for us to obtain a better implementation of MS operator in our program in the future..

References

[1] K. Tang, Y. Mei, and X. Yao, "Memetic algorithm with extended neighborhood search for capacitated arc routing problems," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 5, pp. 1151–1166, 2009.

[2] G. Ulusoy, "The fleet size and mix problem for capacitated arc routing," *European Journal of Operational Research*, vol. 22, no. 3, pp. 329–337, 1985.