A Solver for Capacitated Arc Routing Problem Based on Heuristic Search

Zean He, *12011323*

Abstract—The capacitated arc routing problem (CARP) is a variation of arc routing problem under the restriction of limited capacity for each route which is NP-hard. Existing optimization methods includes constructive heuristics, local search methods, and metaheuristics. This project aims to research the CARP problem and construct a simplified metaheuristics pipeline for solving it using heuristic search and perform additional experiments for further analysis of different strategies and operations, therefore gain a deeper understanding of the problem and optimize the solver.

1 Introduction

ARC routing problem is a type of optimization problem that involves request for certain demands on a directed or undirected graph with nodes and connections, and aims at finding a solution containing one or multiply route(s) that minimizes the total distance, or cost, of the travel. This problem provides an abstraction of many real world scenarios which requires optimization, such as transportation which provides planning for cargo delivery, waste collection and buses services. It can also be applied to service industries which provides planning for customer service or salespeople promotion [1].

One of the variations of the arc routing problem is capacitated arc routing problem (CARP), which possess the additional restrictions on the capacity of single route, namely the vehicle has a limited capacity for transportation. The overall goal still remains the same while ensures that total demands of any single route does not exceed its capacity.

Arakaki and Usberti [2] summarized the literature of CARP heuristics, and generally classified them into three categories: (1) constructive heuristics, which start from an empty solution and iteratively grow it into a feasible solution, (2) local search methods, that start from a feasible solution and attempt to improve it by exploring a neighbourhood, (3) metaheuristics, which usually comprise a combination of constructive heuristics, local search methods and likely some additional optimization techniques. Metaheuristics have higher complexity and require more processing time while often providing the best solutions comparing with the other two listed methods.

This project was intended to research the CARP problem, specifically, the author chose to construct a simplified metaheuristics pipeline of solving it with heuristic search, in the meantime performing additional experiments for further analysis of the effectiveness of different strategies and operations. And outcome of this project is expected to be educational, that could help us gain a deeper understanding of the problem itself along with further optimization of the solver.

2 Preliminary

In this section, we formulate the CARP, we will also introduce the terminologies and notations in this report.

2.1 Formulation

The problem of capacitated arc routing could be defined on a graph $\mathcal{G}$, which is consists of a set of vertices V and a set of undirected edges E, namely $\mathcal{G} = (V, E)$. Each edge $e \in E$, connecting two vertices $v_1 \in V$ and $v_2 \in V$, can be represented as $e = \{v_1, v_2\}$. In CARP, one edge could be considered as a pair of arcs, say, $\langle v_1, v_2 \rangle$ and $\langle v_2, v_1 \rangle$, one for each direction. Each arc is assigned with a unique ID t, thus each edge task would be assigned with two IDs, namely t and $inv(t)$ (the inversion task of t which shares the same properties except reversed vertices), and among t and $inv(t)$ only one can be served. Each arc t is labeled with 4 properties: $tail(t)$, $head(t)$, $sc(t)$, and $dem(t)$, standing for the tail and head vertices, serving cost, and demand of the arc, and we say an edge e needs to be served if the corresponding arc t_e meets $dem(t_e) > 0$ (or $inv(t)$, similarly). To construct a CARP solution, we use sequential aligned tasks IDs $S = (S_1, S_2, ..., S_k)$ which each S_i corresponding to a task t mentioned above to represent a route. The task with ID $= 0$ represents the zero-demand task which simply departure from or returning to depot. With the above definition, the final solution S which may contains more than one route (k routes) can be written as

$$S = (0, R_{11}, R_{12}, ..., 0, R_{21}, R_{22}, ..., 0, R_{k1}, R_{k2}, ..., 0)$$

in which, R_{ij} represents the j^{th} task on route i, and therefore the load of route i is defined as

$$load(R_i) = \sum_{j=1}^{|R_i|} dem(R_{ij}), load(R_i) \leq Q$$

with reference to Q is the capacity limit. To minimize the total cost, two consecutive tasks in the above formulation are connected with the shortest path, therefore we define the connecting cost of S_i and S_{i+1} by $sp(S_i, S_{i+1})$, which stands for the shortest path from $tail(S_i)$ to $head(S_{i+1})$, then the total cost of a route S can be defined as

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

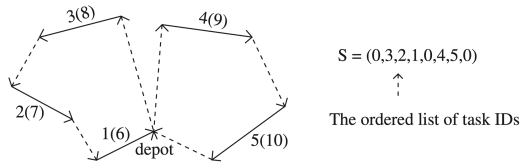

Fig. 1. Illustration of the solution representation [3], IDs in parentheses represent inversions of the current directions.

Given the above definition we can now formulate the problem as the following, where $app(S_i)$ stands for the appearance time of S_i in the solution S.

$$\min_s TC(S) = \sum_{i=1}^{|S|-1}[sc(S_i) + sp(S_i, S_{i+1})]$$
$$s.t. \quad app(S_i) = 1, \forall S_i \in E \; dem(S_i) > 0$$
$$app(S_i) + app(inv(S_i)) = 1, \forall S_i \in E \; dem(S_i) > 0$$
$$load(R_i) = \sum_{j=1}^{|R_i|} dem(R_{ij}) \leq \mathcal{C}$$

2.2 Terminologies and Notations

General notations are listed in Table. 1, while some notations appear in single equations may be listed just below their first appearance for a better understanding.

TABLE 1
Notations used in this report

Notation	Description
$\mathcal{G}$	Problem based graph
V	Vertex set in $\mathcal{G}$
E	Edge set in $\mathcal{G}$
S	Solution of CARP
$head(t)$	Starting vertex of task t
$tail(t)$	Ending vertex of task t
$sc(t)$	Serving cost of task t
$dem(t)$	Demand of task t
$inv(t)$	Inverse arc of task t
$app(S_i)$	Appearance time of S_i in S
$load(R_i)$	Load of serving route R_i
$TC(S)$	Total cost of solution S

3 METHODOLOGY

This section introduces the implementation of hybrid solver for CARP using local search and simulated annealing (SA). To be more detailed, this solver uses SA to supervise the process of optimizing the solution, and trace the best solution, in each iteration, an operator that can modify the route schedule based on a given solution will be randomly picked and applied.

3.1 General Workflow

Referring to Fig. 2, the general workflow of a "slave" could be divided into three stages: *prase the CARP* (reading the input data, and use Floyd etc. to prepare some indirect data that may be used by the operators), *find a valid initial solution* (under a specified), and use SA to *optimize the solution* (which will be detailed explained in section 3.2.3).

For the specific implementation of Python (with GIL's lamination), the program need to use multiprocessing to speed up to achieve an acceptable output. Based on the ground truth that slaves with the same input and same hyper-parameters will behave the same, to prevent such waste, it's also important to properly choose different hyper-parameters for them, this would be explained in section 3.2.4 and verified in section 4.

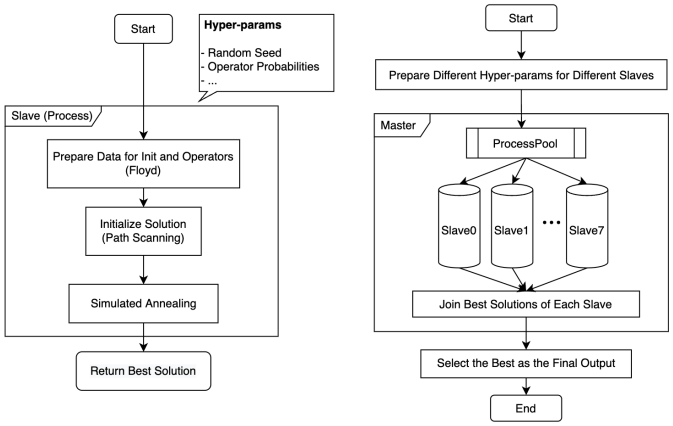

Fig. 2. General workflow for the master-slave architecture based SA process.

3.2 Algorithms and Model Design

3.2.1 Floyd-Warshall

The first step for a slave is to prase the CARP problem, during which, an important and essential part is to prepare a matrix of the shortest distance between any arbitrary two vertices, here the Floyd-Warshall algorithm with a time complexity of $\mathcal{O}(n^3)$ could help us to obtain it.

Algorithm 1 The Floyd-Warshall Algorithm

Require: Matrix⟨int⟩ $cost$: the costs, namely $sc(t)$ for arcs
Ensure: Matrix⟨int⟩ $dist$: the distance matrix

```
function FLOYD(cost) returns a matrix
    dist ← COPY-MATRIX(cost)
    n ← |costs[0]|                          ▷ number of arcs
    for k ∈ [0, n] do
        for i ∈ [0, n] do
            for j ∈ [0, n] do
                if dist[i, j] > dist[i, k] + dist[k, j] then
                    dist[i, j] ← dist[i, k] + dist[k, j]
    return dist
```

3.2.2 Path Scanning (PS)

PS is promised to give a valid, but may be far away from optimal solution [2]. The operators in the next section will do disruptions on the base solution, to simplify it, we do not allow violations (the modified solution must be still legal).

The PS attempts to assign as much tasks as possible into one route, and finally divide the tasks into one or more route(s). Each time, PS need to find the tasks that satisfies the capacity limitation of the current route, and choose the task that is closest to the end of the current route. If there are more than one task that share the same (closest) distance, the strategy input, $rule$, which is one of the PS

rules, will take effect. The author implemented five rules mentioned by Tang *et al.* [3], say, 1) maximize / 2) minimize the distance from the head of task to the depot; 3) maximize / 4) minimize the term $dem(t)/sc(t)$; 5) use rule 1) if the vehicle is less than half-full, otherwise use rule 2).

Algorithm 2 Path Scanning

Require: Matrix⟨int⟩ $dist$: the distance matrix; List⟨Edge⟩ $tasks$: edges with $dem(t) > 0$; Strategy $rule$: a function that selects one arc from the given arcs
Ensure: Solution sol: the solution, contains a list of routes and their meta-info, each route contains a ordered list of tasks

function PATH-SCANNING($tasks$, $rule$) **returns** a solution
  $sol \leftarrow$EMPTY-SOLUTION()
  while $tasks$ is not empty **do**
    $route \leftarrow$EMPTY-ROUTE()
    while there exists $tasks$ that can be put into $route$ **do**
      $closest \leftarrow$CLOSEST-TASKS($route$)
      $choice \leftarrow rule(closest)$
      add the chosen task $choice$ into $route$
      remove $choice$ from $tasks$
    record this $route$ into sol
  return sol

The author was also inspired to add another strategy, *random*. In this case, the step of initializing the solution will take longer time (for instance, 10% of the time limit) to scan the path repetitively and find the best random solution. As experimented in section 4.2.1, the random initialization not only has a satisfying outcome, but also extend the search space, which is likely to contribute increase the probability of finding the optimal solution.

3.2.3 Simulated Annealing (SA)

The key point of CARP is to "jump out" of local optimism. Apart from the merge-split operator proposed by Tang [3], the author preferred to use the SA framework, which is more simple and concise, and can reach a similar outcome. The workflow (Fig. 3) and pseudo code are helpful for understanding SA, and it's further analysed in section 3.3.

To distribute the solution and gain a new solution, the author implemented the six operators mentioned in Tang's paper [3]:

- **Single Insertion** picks a task from a route, and select a new position in either the same route, or a different route, or even a new empty route to insert it. The selection strategy is *random* to ensure the search space is feasible.
- **Double Insertion** is similar to single insertion, but two consecutive tasks are selected at once.
- **Swap** pick two tasks (either from the same route, or from different routes), and swap their position. Note that it is prevented to pick tasks from two routes that each contains only one task, since this operation contributes no difference.
- **2-opt** checks whether reverse any fragment in solution will reduce the total cost of the specific solution. The combination of the start and end is traversed to ensure all of the task in a route is arranged in the optimal sequence.
- **Flip** is a simple and low costing operator that simply try to insert a task in the reversed direction (say, the other arc of the edge), and applies the change if it does lower the cost. This operator has a satisfying trade-off, that we decide to apply it right after applying any other operator.
- **Merge-Split (MS)** is a operator that can change the solution to a greatly different one, which is another key of jumping out of a local valley (optimal). Since we already applied SA, we simply implemented a random MS, instead of the complete version in Tang's paper.

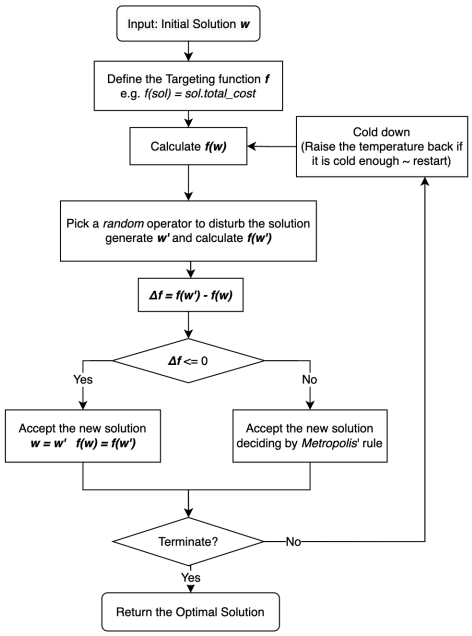

Fig. 3. Detailed workflow of simulated annealing.

Algorithm 3 Simulated Annealing

Require: Solution sol: the instance of class Solution, contains one or more route(s), each route contains one or more task(s), the object also contains essential metadata; other needed information for CARP are used as global variables
Ensure: Solution $o3$: the optimized solution

function SIMULATED-ANNEALING(sol) **returns** a solution
  $o3 \leftarrow sol$ ▷ the optimal solution during SA
  $current \leftarrow sol$
  $T \leftarrow$ initial temperature
  while terminate condition does not satisfied **do** ▷ time limit for here
    $op \leftarrow$ randomly choose an operator ▷ different slaves have different probabilities of choosing operators
    $res \leftarrow op(current)$
    $\Delta f \leftarrow$EVALUATE(res)-EVALUATE($current$)
    if $\Delta f \leq 0$ or enter with a probability of $e^{-\Delta f/T}$ **then**
      $current \leftarrow res$
      keep tracking (update) the optimal solution
    $T \leftarrow T \times \alpha$ ▷ colden
    if T is low and stable **then**
      $T \leftarrow$ initial temperature ▷ restart SA
  return $o3$

3.3 Analysis

For the slave's workflow, the Floyd-Warshall algorithm is firstly applied, using a $\mathcal{O}(n^3)$ cost to save a grate part of time in the following operator's decisions. The path scanning was

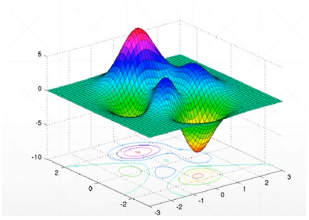

Fig. 4. A visual example of local optimal[1].

then used to initialize our metaheuristics problem solving. These two basic steps are essential, and we focus more on the SA process.

SA is a general control for both gradually reaching the optimal solution, and providing a chance for jumping out of the possible local optimal. The Metropolis' rule that makes this decision contributes the major part of SA's competitiveness, making SA to be robustness, global convergence, implicit parallelism, and extensiveness and adaptability [4]. Take Fig. 4 as an example, the local optimism could be jumped out by restarting SA (raising the temperature to make the probability of entering the update-if-block high again) or the MS operator.

4 Experiments

4.1 Setup

The local test could be separated into two phases. The first phase focuses more on the correctness and the performance of certain operator or strategy, while the second one takes a overall control of the solver's performance.

For phase one, the author wrote a class method that accepts the string representation of solution as input, then directly calculate the cost and verify the correctness. A decorator was also applied to automatically get trace of the optimism of solution during the SA process. For phase two, a script was applied to automatically run tests in batch and record the performance of each configuration.

The data sets used in the experiments including: the officially provided 7 problems for validity verification, and an additional dataset, DIMACS[2], was adapted and tested for further efficiency analysis and hyper-parameters' fine-tuning.

The configurations of the test environment are listed in Table 2.

TABLE 2
Configurations of the test environment.

System	Hardware	Environment	Performance
macOS 13.0	Apple M1 Pro (aarch64) 16GB RAM	Python 3.9.13 NumPy 1.21.5	1 process, 20 s (P1) 8 processes, 60 s (P2)

4.2 Results

4.2.1 Path Scanning

Path scanning works as the most fundamental part of the whole pipeline which offers an initial solution and a ground truth could be observed that the final solution cost is closely related to the initial solution. As it was mentioned above that it has certain heuristic rules, here we further experiments with different rules independently and intended to reveal the inter-relationship between them. Some of the representative data sets' results applied with path scanning only are listed in Table 3, with local search not being involved.

TABLE 3
Total cost of the initial solution found by path scanning, with different rules applied.

case \rule	1	2	3	4	5	all	all+rand
val4A	212	212	211	196	212	196	173
val7A	364	342	339	373	342	339	292
gdb1	370	370	378	378	370	370	316
egl-e1-A	4317	4317	4317	4317	4317	4317	3853
egl-s1-A	6468	6468	6468	6468	6468	6468	5899

In the table above, rule $1-5$ corresponding to the five rules mention before and *all* represents the combination of them and *all* + *random* extends the path scanning with a complete random selector which uses it for all of the task selection.

4.2.2 Operator & Combination

With the initial solution offers by path scanning, we can perform local search which further optimize the overall costs, and this step requires different operators to perform operations to the routes that consists the solution. To test out the effectiveness of each operator, we test them separately first and the results are shown below in Table 4.

TABLE 4
Performance of applying each operators. Samples are the total cost of running the single operator selected for 20 seconds.

case \operator	insertion	swap	2-opt	MS
val4A	173	173	173	173
val7A	279	292	292	292
gdb1	316	316	316	316
egl-e1-A	3708	3853	3853	3853
egl-s1-A	5582	5899	5899	5877

Apart from single operator performance, the combination of different operators may posses unexpected effect due to the pairing search ability, therefore we start with the simplest single insertion operator and add the rest operator one by one to reveal the interaction between them, which is shown in the Table 5.

TABLE 5
Performance of combining a set of operators, they share the same probabilities of being chosen.

case \operator	insertion	+ 2-opt	+ swap	+ MS
val4A	173	173	173	173
val7A	279	279	279	279
gdb1	316	316	316	316
egl-e1-A	3708	3799	3590	3652
egl-s1-A	5582	5539	5545	5428

1. https://www.cnblogs.com/yanjy-onlyone/p/11600259.html
2. http://dimacs.rutgers.edu/index.php/download_file/view/277/320/383/

4.3 Analysis

4.3.1 Heuristic Scanning

From the results above, we can see that different path scanning rules may differs according to the test sample. For instance, the rule 3 has slightly advantages in *val7A* while a lower performance on *gdb1*, and rule 4 has obvious advantage on *val4A* that even reached optimal while having the worse performance on *val7A* among five rules. These can great vary due to the fitness of strategies to the special case, which certain strategies may gain performance on a paring scenario. Therefore there aren't a constant evaluation criteria for either rule and the simplest way is to combine their results together and choosing the better solutions as local search start point for further optimization.

Apart from the five rules and their combination, another strategy which only use stochastic for task selection is introduced and receive unexpected results, though this takes longer time. Different from the path scanning rules above, which terminates once a legal, complete solution is found, the stochastic strategies repeatedly constructing possible solution, which uses random every time multiple tasks are presented for selection. And this simply update the solution when a new one with less total cost is found and terminates when specific time limit is reached ($\frac{2}{5}$ of the time limit in the final implementation). Though through brute-force scanning which requires much more time for initial solution, but it reaches a much optimized solution and considerable improvement, even reaching optimal solution, and by utilizing multiprocessing and assigning certain fraction of process performing such stochastic process the overall performance is boosted.

4.3.2 Comparison on Operators

Different operators are said to have different step size, based on the modifications on a solution. Operators with small step size are more likely to converge but also more likely to be stuck in local optima. While operators with large step size possess the advantage of jumping out of local minima and finding the global optimal, while might have problem of converging. Among the four operators we implemented, we perform experiments to analysis the strength of them separately and provides ideas about the way of combining them.

The operator comparison provides us with a view of the performance of each operator. From the results above we can see that the *single insertion*, which is also the simplest one, achieves the highest score among all sample cases, especially in *egl-s1-A*, which possess a advantage around 300 cost unit. From its definition and implementation it's obvious that this is a operator with small step size and therefore more likely to converge to an more optimized solution. Instead the *swap*, *2-opt* and *MS* have lower performance then *single insertion* due to the larger step size they possess, but in the following experiments we can see the power of combining them.

4.3.3 Analysis for Operators' Combinations

The local search of solution requires both convergence and not to be constrained to the local optimal, which therefore requires us to combine different operators and achieves a neutral state. We still start with the simplest *single insertion* operator but later added with operators with large step size among it to enable a larger local search spaces. From the results above we can see that the performance of every operator gain certain boost with the help of others. Among this the *swap* operator gain advantage in the *egl-e1-A* sample and *MS* operator reaches the least costs on sample *egl-s1-A*. Also we can see that for larger case *MS* operator and *swap* operator have the ability to further optimization due to their larger step size then *2-opt* and *single insertion*. Therefore in the final implementation, we uses different combination of operators with both the large step size operator for broader search and also the small step size operator for local converging and achieves most optimal solutions.

5 Conclusion

In this project, we combined multiple heuristic search with stochastic strategies, ensemble learning, simulated annealing together to construct a CARP pipeline solver and further optimize the final results. Also through the experiments of different scanning rules, operators and their combination, we further analyze the existing problem by constructing the abstract model of this problem and solve them by methods such as expanding the local search spaces, increasing step size while remaining convergence, etc.

This project offers a chance for us to explore the efficacious heuristic search and its application in the capacitated arc routing problem which offers an abstraction of wide range of real world problem. Through this project, I not only learned the problem itself and solutions related to it along, but more importantly the skills of analyzing optimization problem drawbacks and further optimize them with strategies I've learned through this course. These two together form an continues integration of theoretical knowledge and real-world problem implementations which is meaningful and intriguing. Future work could be consisting of two ways, namely one that exploring more powerful operators combinations which enhanced the search span while remains convergence, and the other which further enrich the population by introducing EA or MA into the overall structure.

References

[1] Á. Corberán, R. Eglese, G. Hasle, I. Plana, and J. M. Sanchis, "Arc routing problems: A review of the past, present, and future," *Networks*, vol. 77, no. 1, pp. 88–115, 2021.

[2] R. K. Arakaki and F. L. Usberti, "An efficiency-based path-scanning heuristic for the capacitated arc routing problem," *Comput. Oper. Res.*, vol. 103, pp. 288–295, 2019.

[3] K. Tang, Y. Mei, and X. Yao, "Memetic algorithm with extended neighborhood search for capacitated arc routing problems," *IEEE Trans. Evol. Comput.*, vol. 13, no. 5, pp. 1151–1166, 2009.

[4] O. Catoni, "Metropolis, simulated annealing, and iterated energy transformation algorithms: Theory and experiments," *J. Complex.*, vol. 12, no. 4, pp. 595–623, 1996.