CS303 · Artificial Intelligence

Project 2 CARP

周思呈 / 12110644

May 15, 2023

1 Introduction

The Capacitated Arc Routing Problem (CARP) is a combinatorial optimization problem that involves finding the optimal set of routes for a fleet of vehicles to serve a set of customers located on a network of arcs or edges, while respecting capacity constraints on the vehicles and satisfying the demand of each customer.

The goal in CARP is to find an optimal or near-optimal solution that balances the sequencing of the customers on the routes and the efficient utilization of the vehicles, while satisfying all the constraints. CARP has numerous real-world applications, including waste collection, package delivery, and public transportation planning.

2 Preliminary

2.1 Notation

Symbol	Description
G	undirected connected graph
V	vertex set
v_i	the i^{th} vertex
v_0	depot
E	edge set
e_i	the i^{th} edge
T	task set
τ_i	the $i^{th}task$
Q	capacity of vehicle
$c(e)$	cost of edge
$d(\tau)$	demand of task
R_k	the i_{th} route
$RC(R_k)$	route cost of route R_k
S	solution
$TC(S)$	total cost of solution s

Table 1: Notation List

2.2 Problem Formulation

The CARP problem can be formulated as follow:

Input graph $G(V, E)$, cost $c(e) > 0$ and demand $d(e) \geq 0$ for $\forall e \in E$. The task set $T = \{\tau \in E | d(\tau) > 0\}$. Vehicle with capacity Q is waiting at the depot $v_0 \in V$.

Solution $S = (R_1, R_2, ..., R_m)$. The k^{th} route $R_k = (0, \tau_{k1}, \tau_{k2}, ..., \tau_{kl_k}, 0)$ where

$$\sum_{i=1}^{l_k} d\left(\tau_{ki}\right) \leq Q, \forall k = 1, 2, \ldots, m \tag{1}$$

$$\sum_{k=1}^{m} l_k = |T| \tag{2}$$

$$\tau_{k_1 i_1} \neq \tau_{k_2 i_2}, \forall (k_1, i_1 \neq k_2, i_2) \tag{3}$$

$$\tau_{k_1 i_1} \neq \text{inv}\left(\tau_{k_2 i_2}\right), \forall (k_1, i_1 \neq k_2, i_2) \tag{4}$$

The goal is to minimize

$$TC(S) = \sum_{k=1}^{m} RC\left(R_k\right) \tag{5}$$

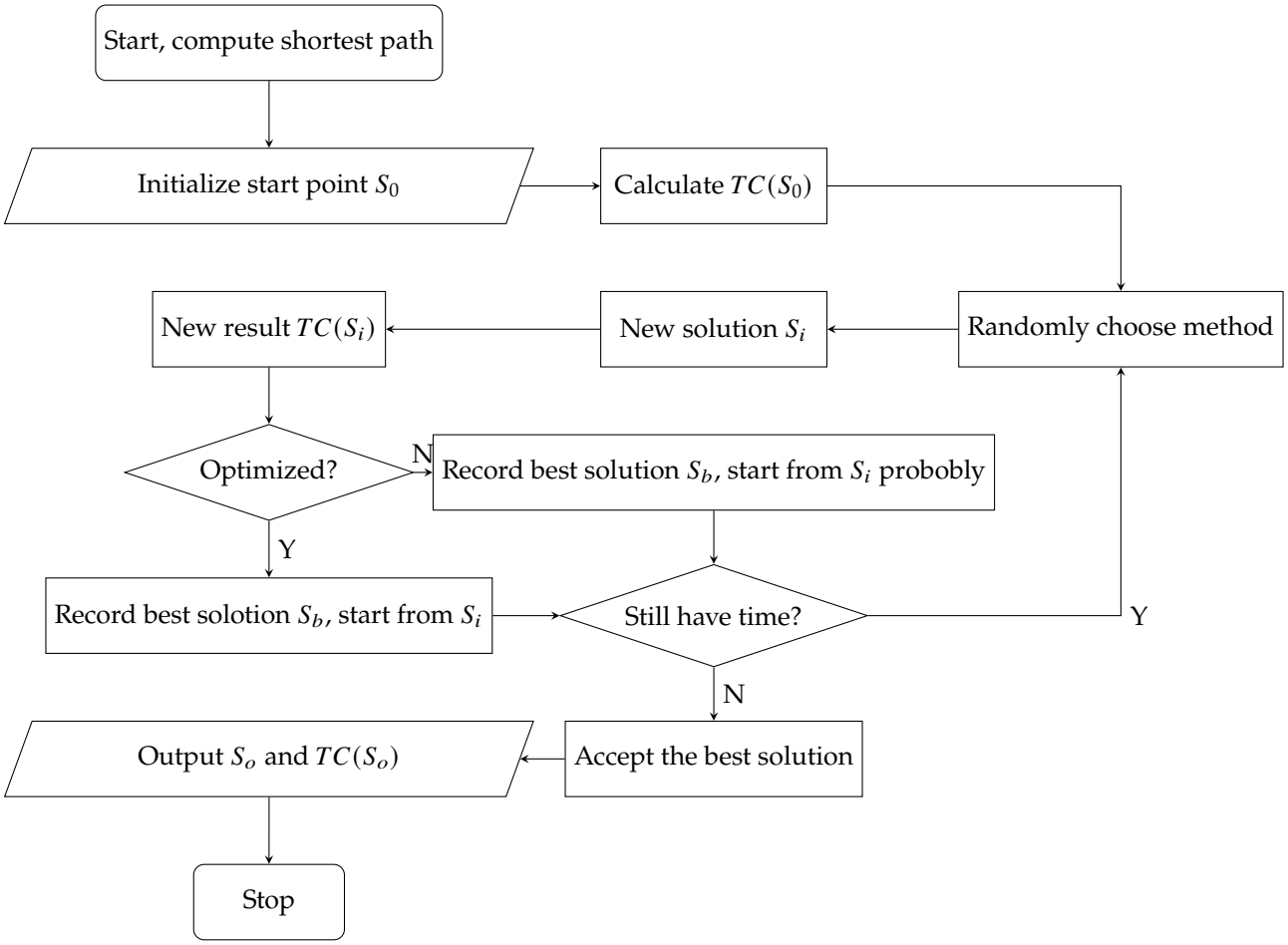
where

$$RC\left(R_k\right) = \sum_{i=1}^{l_k} c\left(\tau_{ki}\right) + dc\left(v_0, \text{head}\left(\tau_{k1}\right)\right) + \sum_{i=2}^{l_k} dc\left(\text{tail}\left(\tau_{k(i-1)}\right), \text{head}\left(\tau_{ki}\right)\right) + dc\left(\text{tail}\left(\tau_{kl_k}\right), v_0\right) \tag{6}$$

3 Methodology

3.1 General Description and Flow Chart

Generally, I use simulated annealing algorithm to solve this problem. First I use Path-Scanning to find the start point. Then, from this start point the program generate next search point by using Stochastic-Disturbance and Split-Merge algorithm. The algorithm maintain a globally optimal solution. After that, it decide what the next start point is. If the new solution is better than the previous one, the new one becomes start point, otherwise it becomes the new start point with probability.



3.2 Detailed Model Design

3.2.1 Path Scanning

The Path Scanning algorithm start from depot with full capacity. Then repeatedly select the closest unvisited task and add it into current R. If several tasks are with the same distance, select one based on rules considering factors such as distance, demand satisfaction, or capacity constraints. If the remain capacity is less than the smallest task, go back to the depot. Repeat this procedure until every task is served.

Algorithm 1: Path Scanning($G = (V, E, T), Q$)

```
T' ← T; S ← ∅; Q' ← Q;
while T' is not empty do
    R ← ∅; pos ← v_0;
    while Q' > 0 and T' is not empty do
        τ ← closest task; // could be more than one task, apply five rules;
        if d(τ) > Q' then
            return S;
        end
        pos ← position after doing the task τ;
        Q' − = d(τ); add τ to R;
    end
    S add R;
end
return S
```

Notice that in real implementation, there could be multiple uncompleted tasks with the same shortest distance from current position. In this case, use the following five rules to select next task:

1) Maximize the distance from the task to the depot;

2) Minimize the distance from the task to the depot;

3) Maximize the term $d(\tau)/c(\tau)$;

4) Minimize the term $d(\tau)/c(\tau)$;

5) Use rule 1) if $Q' < Q/2$, otherwise use rule 2).

Using different rules in task selection may bring different results. The function called Complex Path Scanning implements all the rules above and chose the best solution at last. However, producing one fair result by calculating five times is somehow time consuming. Therefore, I also implement Simple Path Scanning, which simply randomly select one task in multiple closest situation. The ratio of the two methods should be determined by the size of the solution space.

Also notice that while selecting the closest and fair task, there could be two ways of implementation: one is to get all the fair tasks and select the closest one, the other is sort all the tasks in distance order then select the first fair one. Although it may sound stupid to distinct the above two seemingly equivalent situations, they are in fact different. The latter one searches bigger solution space thus have better search result.

3.2.2 Merge Split

Merge Split algorithm can be used to generate new solution based on the old one. It can jump in big steps compared with methods in Stochastic Disturbance.

Firstly it randomly select several R_{old} in S_{old} and split them into single tasks. Next, those tasks are once more processed by Path Scanning, thus producing new R_{new}. Replacing R_{old} with R_{new} we can get S_{new} which is the new fair solution.

Algorithm 2: Merge Split(G, Q, S)

randomly select several R_{old} and split them into task set $T^{'}$;
return Path Scanning($G = (V, E, T^{'}), Q$);

3.2.3 Stochastic Disturbance

Stochastic Disturbance methods involves Reverse, Swap and Single Insertion. All of these three methods are jumping in small steps, which can search in a more precise way.

Notice that in the implementation of Single Insertion, the algorithm first pick out every fair insertion pairs then randomly select one from it. However, in Swap, the algorithm only tries ten times in looking for fair solution. This is because Swap algorithm is more likely to generate a fair solution.

Algorithm 3: Reverse(S)

$L \leftarrow$ all the R_i which contains more than one task;
$R \leftarrow$ random route in L;
$t \leftarrow$ random number which is less than the size of R;
reverse the order of tasks from the t^{th} to the last one;
replace the old R with the new one;
return new S

Algorithm 4: Swap(S)

randomly select two routes R_a and R_b;
randomly select task τ_a from R_a and task τ_b from R_b;
swap τ_a and τ_b;
cnt $\leftarrow 1$;
while *cnt < 10 or the new R_a or R_b out of capacity* **do**
    randomly select task τ_a from R_a and task τ_b from R_b;
    swap τ_a and τ_b;
    cnt += 1;
end
return new S

3.2.4 Simulated Annealing

Simulated Annealing is a metaheuristic algorithm inspired by the annealing process in metallurgy. It is commonly used to solve combinatorial optimization problems, including the Capacitated Arc Routing Problem (CARP). This algorithm is the main frame of my implementation here, and the main process has been demonstrated in section 3.1.

Merge Split and Stochastic Disturbance can both be used to generate new solutions from the old one. The former one jump in larger step and the latter one smaller. The ratio of the two methods should be determined by the size of the solution space.

The probability of starting from the worse solution is also important. If it is too large, the algorithm may search ineffectively. If it is too small, the algorithm may fall into local optimal solution. Here we update t according to the chosen solution generating method. If bigger step, update t to 1, otherwise $t = t \times 0.99$.

Algorithm 5: Single Insertion(S)

$L \leftarrow \emptyset$;
for *every* τ **do**
    find R able to accept it;
    add τ and R pair into L;
end
randomly select one pair from L;
randomly chose a position in R_L;
insert τ into this position;
return new S;

Algorithm 6: Schedule(k, λ, limit, t)

if $t >$ *limit* **then**
    return $k \times e^{-\lambda t}$
end
return 0;

Algorithm 7: Simulated Annealing(S_{start})

$S_{optimal} \leftarrow S_{start}$; $TC_{optimal} \leftarrow TC(S_{optimal})$;
while *still have time* **do**
    $T \leftarrow$ Schedule(k, λ, limit, t);
    generate new solution S_{new} from S_{start};
    $d \leftarrow$ TC(S_{new}) - TC(S_{start});
    if $d > 0$ *or* $e^d/T >$ *random(0,1)* **then**
        $S_{start} \leftarrow S_{new}$
    end
    update the value of $S_{optimal}$ and $TC_{optimal}$;
    update t;
end
return $S_{optimal}$

Type	Configuration
Software	Python 3.8.16, numpy 1.23.5
Hardware	MacBook Pro, Apple M1, 8 cores, 3.2GHz
Random Number	100
Run Time	60

Table 2: Personal Judge Environment

Data Set	Required Edges	Non-requred Edges	Total Cost	Capacity
egl-e1-A.dat	51	47	1468	305
egl-s1-A.dat	75	115	1394	210
gdb1.dat	22	0	252	5
gdb10.dat	25	0	252	10
val1A.dat	39	0	146	200
val4A.dat	69	0	343	225
val7A.dat	66	0	249	200

Table 3: Data Set Characteristics

4 Experiments

4.1 Setup

Table 2 is my personal judge environment. Table 3 is the data set characteristics.

4.2 Results

Table 4 shows my judge result. Two 'Result' column comes from the final optimal algorithm. The Brute Force used the strategy of continuous Merge Split and complicated Path Scanning.

Data Set	Platform Result	Personal Result	Brute Force
egl-e1-A.dat	3800	3840	3958
egl-s1-A.dat	5761	5770	6125
gdb1.dat	340	328	328
gdb10.dat	297	297	294
val1A.dat	188	191	196
val4A.dat	414	417	419
val7A.dat	289	288	287

Table 4: Judge Result

4.3 Analysis

Combining the characteristics of the data set to analyze the two sets of results, we can see that the final algorithm have better performance with larger solution space. In data set egl-e1-A.dat and egl-s1-A.dat, there are a lot of non-required edges, thus have bigger solution space. For these kind of data set, SA algorithm is able to search efficiently. For the others, Merge Split has already gotten good performance, so the improvement looks small.

5 Conclusion

In this project, I implement Simulated Annealing to solve CARP problem. Instead of treating every data set similarly, the algorithm can judge the size of the solution space thus chose search methods in a more effective way. Also, when implementing Path Scanning, I notice the difference between "first fair then closest" and "first closest then fair" and search more effectively. After that, I run tests on both big solution space and small ones, all acquire good results.