CS303 AI Project2: Carp

Name: Shimin Luo
SID: 12012939

I. INTRODUCTION

A. Origin and Brief Description

Capacitated Arc Routing Problem(CARP) is a classical NP-hard arc routing problem proposed by Golden and Wong in 1981, which is normally solved by heuristic and meta-heuristic search algorithms. CARP can be visualized as a concrete problem: Giving a connected graph(the vertex means city and edge means road) with corresponding edge-weight(the cost to travel this road), and there are several tasks on some certain roads, each one associated with a task-demand. A fleet of vehicles, each of capacity Q, will start from a specific depot v_0 to serve all the tasks.

The goal of CARP is to determine a set of routes for the fleet to accomplish all the tasks with minimal cost while satisfying:

1) : Each route must start and end at v_0.

2) : Demand serviced on each route must not exceed Q.

3) : Every task must be accomplished in one time.

B. Application

CARP was widely used in daily life, especially in **municipal services**. Specific applications include but are not limited to road sprinkler path planning, garbage recycling vehicle path planning, postal express delivery route planning, road deicing vehicle path planning, school bus routing problems and so on.

C. Project Purpose

The purpose of this project is to find a path plan making the total cost as minimal as possible. Above all, we need to ensure the correctness of the solution, and then using some optimization methods to improve the solution. In addition, the algorithm needs to read the input-file and output the result with specific format.

II. PRELIMINARY

A. Problem Input Formulation and Notation

1) G(V, E): Given an undirected connected graph $G(V, E)$ with a vertex set V and an edge set E.

2) c(e): For each edge $e \in E$ incurs a cost whenever a vehicle passes it or provides service on it.

3) d(e), t and T: In the edge set E, there are several edges that have requirements to be served. For a edge e, it has a demand $d(e) \geq 0$. Those edges whose $d(e)>0$ form a set $T = \{t \in E | d(t)>0\}$, which is a subset of edge set E. For each required edge $t \in T$, the requirement $d(t)$ on it must be served at once, but this edge can be traveled multiple times.

4) v_0 and Q: Vehicles take a certain vertex $v_0 \in V$ as a depot. Each vehicle has same maximum load capacity Q.

B. Problem Output Formulation and Notation

The goal of CARP is to find a set of routes that would minimize the total cost of driving while satisfying the constraints above. And for each route, vehicle must depart from the depot, arrive at the depot after completing all scheduled tasks, and not be overloaded (the total demands served in this route not exceeding the maximum load capacity Q). Some notations are listed following:

1) $head(t)$ and $tail(t)$: $head(t)$ and $tail(t)$ are endpoints of task t. It means that task t is served from $head(t)$ to $tail(t)$.

2) R_k: A list, represents one scheduled route for the a vehicle to serve some tasks. $R_k = [t_1, t_2, \ldots t_{l_k}]$

3) s: The solution of CARP problem $s = [R_1, R_2 \ldots R_m]$. It is a two dimension list.

4) $sd(v_i, v_j)$: The shortest distance(cost) between vertex v_i and v_j.

5) RC(R_k): The route-cost of R_k. $R_k = \sum_{i=1}^{l_k} cost[t_{ki}] + sd(v_0, head(t_{k1})) + \sum_{i=2}^{l_k} sd(tail(t_{k(i-1)}, head(t_{ki})) + sd(tail(t_{kl_k}), v_0)$

6) TC(s): $TC(s) = \sum_{k=1}^{m} RC(R_k)$. The total cost of a solution s.

7) RL(R_k): The route-load of R_k.

8) TL(s): A list including the load of each route in s. $TL(s) = [RL(R_1), RL(R_2), \ldots, RL(R_m)]$

C. Notations of data structure and variables during algorithm

1) cost: A directory stores all edge-cost, key is e and value is $c(e)$, where $e \in E$.

2) demand: A directory stores all demands, key is t and value is $d(t)$, where $t \in T$.

3) short_distance: A two dimension matrix stores all $sd(v_i, v_j)$ where $v_i, v_j \in V$.

4) best_routes,best_cost,best_load: The optimal solution s discovered so far, and its $TC(s)$, $TL(s)$.

5) weight_list: A list records the weights of improved methods $Flipping()$, $Self\_insertion()$, $Swap()$, $Single\_insertion()$, $Double\_insertion()$, $Single\_2\_opt()$ and $Cross\_2\_opt()$ respectively.

III. METHODOLOGY

A. General workflow

This section will introduce the core methods and algorithms of this project. The proposed method divides into:

1) Step1 Preparation: **Process the input file** of CARP problem, extract $G(V, E), v_0, Q, c(e)$ and $d(e)$ for each edge, store them into $cost$, $demand$ and other corresponding variables respectively. And then, using **Floyd algorithm** to compute $sd(v_i, v_j)$ between any two vertices, store the information in $shortdistance$.

2) Step2 Construction: Using **Path-scanning** algorithm to initiate five reasonable solutions with 5 different priority selection rules(more details in next part).

3) Step3 Improvement: I accomplished lots of improvement methods: $flipping()$, $self\_insertion()$, $swap()$, $single\_insertion()$, $double\_insertion()$, $single\_2\_opt()$ and $cross\_2\_opt()$. Then, I incorporated them into my code to optimize my solution, which use a little bit of the idea of genetic algorithms.

B. Detailed algorithm

For $Step1$, since we have learned Floyd algorithm before and there's no other special method to introduce, I introduce my core algorithms from $Step2$.

1) Path-Scanning: It is used to find out some reasonable solutions as initial solutions, which will be improved later. The key idea is: choose the next task which is closest to the end of current route, not yet serviced and compatible with vehicle capacity. If multiple tasks are the closest to the end of current route, I use function $Better(u, u_j, rule, cur\_load)$ to prioritize them, where u is the task being taken into account, u_j is the best task chosen before, integer $rule$ represents the priority rule to use, cur_load represents current load of this route. The function return $True$ if u is better than u_j under priority rule represented by $rule$.

Algorithm 1 : Better

Input: $u, u_j, rule, cur\_load, Q$
Output: $True$ or $False$

global $v_0, demand, cost, Q$
if rule == 0 **then**
  return $True$ if the end of u is farther to v_0 than u_j
end if
if rule == 1 **then**
  return $True$ if the end of u is closer to v_0 than u_j
end if
if rule == 2 **then**
  return $True$ if $\frac{demand[u]}{cost[u]} > \frac{demand[u_j]}{cost[u_j]}$
end if
if rule == 3 **then**
  return $True$ if $\frac{demand[u_j]}{cost[u_j]} > \frac{demand[u]}{cost[u]}$
end if
if rule == 4 **then**
  if cur_load $\leq$ 0.5 $\times$ Q **then**
    same as situation rule == 0
  else
    same as situation rule == 1
  end if
end if
return $False$

As for the Path-Scanning algorithm, I read the reference book [1] to understand its procedure. And using five different priority rules above, I generated **five initial solutions**.

Algorithm 2 :Path-Scanning

Input: rule
Output: routes, costs, loads

global $v_0, demand, cost, Q, short\_distance$
$k \leftarrow 0$
$free \leftarrow$ list of *demamd.key()*
while len($free$) != 0 **do**
  $k \leftarrow k + 1$, $R_k \leftarrow \emptyset, load_k \leftarrow 0, cost_k \leftarrow 0$
  $i \leftarrow v_0, d_j \leftarrow 0$
  while len($free$) != 0 && d_j != INF **do**
    $d_j \leftarrow INF$,$u_j \leftarrow None$
    for $u \in free \&\& load_k + demand[u] \leq Q$ **do**
      if $sd(i, head(u)) < d_j$ **then**
        $d_j \leftarrow sd(i, head(u))$, $u_j \leftarrow u$
      elseif $sd(i, head(u)) == d_j \&\&$ $Better(u, u_j, rule, load_k)$
        $u_j \leftarrow u$
      end if
    end for
    R_k.append(u_j)
    remove u_j and $(tail(u_j), head(u_j))$ form $free$
    $load_k \leftarrow load_k + demand[u_j]$
    $cost_k \leftarrow cost_k + d_j + cost[u_j]$
    $i \leftarrow tail(u_j)$
  end while
  $costs[k] \leftarrow cost_k + sd(i, v_0)$
  $routes[k] \leftarrow R_k, loads[k] \leftarrow load_k$
end while
return routes, costs, loads

2) Improvement Method: For $Step3$, I used several improvement methods to optimize the solutions from Path-Scanning algorithm.

Flipping: Flip a service direction of a task to see whether the solution can be optimized.

Algorithm 3 : Flipping

Input: best_routes,best_cost,best_load
Output: routes, costs, loads

global $v_0, short\_distance$
$t \leftarrow$ select a task randomly
$start \leftarrow$ the $tail$ of the previous task of t, or v_0
$end \leftarrow$ the $head$ of the next task of t, or v_0
$old\_cost \leftarrow sd(start, head(t)) + sd(tail(t), end)$
$new\_cost \leftarrow sd(start, tail(t)) + sd(head(t), end)$
if $old\_cost \geq new\_cost$ **then**
  $routes \leftarrow$ best_routes replaced t with (tail(t),head(t))
  $costs \leftarrow best\_cost - (old\_cost - new\_cost)$
  return $routes, costs, best\_load$
end if
return $best\_routes, best\_cost, best\_load$

Self_insertion: Select a task randomly and insert it to another position **in its route** to see whether the solution can be optimized. Since the algorithm's logic and implementation is similar to $Swap()$ expect the recalculation of the route-load, I omit the pseudocode for the space of more experimental exposition in next part.

Swap: Select 2 tasks randomly in different routes and swap them to see whether the solution can be optimized.

Algorithm 4 : Swap

Input: best_routes,best_cost,best_load
Output: routes, costs, loads

global $v_0, demand, short\_distance, Q$
$t_1, t_2 \leftarrow$ select two tasks randomly which can be swapped(the load of routes need to be taking into account)
$start1 \leftarrow$ the $tail$ of the previous task of t_1, or v_0
$end1 \leftarrow$ the $head$ of the next task of t_1, or v_0
$start2 \leftarrow$ the $tail$ of the previous task of t_2, or v_0
$end2 \leftarrow$ the $head$ of the next task of t_2, or v_0
old_cost= $sd(start1, head(t_1)) + sd(tail(t_1), end1) + sd(start2, head(t_2)) + sd(tail(t_2), end2)$
new_cost= $sd(start1, head(t_2)) + sd(tail(t_2), end1) + sd(start2, head(t_1)) + sd(tail(t_1), end2)$
if $old\_cost \geq new\_cost$ **then**
  $routes \leftarrow$ copy $best\_routes$ and swap t_1 and t_2
  $costs \leftarrow best\_cost - (old\_cost - new\_cost)$
  $loads \leftarrow best\_load$ updating t_1's, t_2's route- load
  return $routes, costs, loads$
end if
return $best\_routes, best\_cost, best\_load$

Single_insertion: Select a task randomly and insert it to other position in **another route** to see whether the solution can be optimized.

Algorithm 5 : Single_insertion

Input: best_routes,best_cost,best_load
Output: routes, costs, loads

global $v_0, demand, short\_distance, Q$
$t \leftarrow$ select a task randomly
$exchange \leftarrow$ a list of routes where t can insert into
$r2, ixd \leftarrow$ randomly pick a route and insert position
$start1 \leftarrow$ the $tail$ of the previous task of t, or v_0
$end1 \leftarrow$ the $head$ of the next task of t, or v_0
$routes \leftarrow$ remove t to its new position
$start2 \leftarrow$ the $tail$ of the new previous task of t, or v_0
$end2 \leftarrow$ the $head$ of the new next task of t, or v_0
compute the old_cost,new_cost
if $old\_cost \geq new\_cost$ **then**
  $costs \leftarrow best\_cost - (old\_cost - new\_cost)$
  $loads \leftarrow$ copy $best\_load$, update t's route- load
  return $routes, costs, loads$
end if
return $best\_routes, best\_cost, best\_load$

Double_insertion: Select two consecutive tasks randomly and insert them to other position in another route to see whether the solution can be optimized. The algorithm is similar with $Single\_insertion$.

Single_2_opt: First, I choose a route with length ≥ 3 randomly in $best\_route$, and pick task t_1 in the first half of the route, task t_2 in the second half of the route. And then reverse the direction of subroute from t_1 to t_2, compute the new $TC(s)$ to see weather solution can be improved. Since the algorithm is not very difficult, I omit pseudocode here.

Cross_2_opt: Pick two routes, for each route, split into two parts. Each subroute recombines with another from different route. Ensure that the reorganization is reasonable, to see whether the two kinds recombined solutions outweight original one.

Algorithm 6 : Cross_2_opt

Input: best_routes,best_cost,best_load
Output: routes, costs, loads

global $v_0, demand, short\_distance, Q$
$r1, r2, r3, r4 \leftarrow$ four subroutes with demands $d1, d2, d3, d4$, which are splitted from two randomly selected routes.
$new\_cost1 \leftarrow infinity$, $new\_cost2 \leftarrow infinity$
if $d1 + d3$ && $d2 + d4 \leq Q$ **then**
  $route1 \leftarrow [r1, r4]$
  $route2 \leftarrow [r3, r2]$
  $new\_cost1 \leftarrow$ copy $best\_cost$ and update
  $new\_load1 \leftarrow$ copy $best\_load$ and update
end if
if $d1 + d4 \leq Q$ && $d2 + d3 \leq Q$ **then**
  $route3 \leftarrow [r1, reversed(r3)]$
  $route4 \leftarrow [reversed(r2), r4]$
  $new\_cost2 \leftarrow$ copy $best\_cost$ and update
  $new\_load2 \leftarrow$ copy $best\_load$ and update
end if
if new_cost1_cost2 && $new\_cost1 \leq best\_cost$ **then**
  $routes \leftarrow$update $best\_routes$ with $route1, route2$
  $costs \leftarrow new\_cost1$
  $loads \leftarrow$ copy $best\_load$, update it
  return $routes, costs, loads$
end if
if new_cost2_cost1 && new_cost2_cost **then**
  $routes \leftarrow$update $best\_routes$ with $route3, route4$
  $costs \leftarrow new\_cost2$
  $loads \leftarrow$ copy $best\_load$, update it
  return $routes, costs, loads$
end if
return $best\_routes, best\_cost, best\_load$

Combining Above methods: After Path-Scanning, I need to invoke above methods appropriately to improve my solution. Since these methods are kinds of local-search — just the degree of change to the solution is large or small—the solution is easily falling into local optimum. I have written codes in different versions, which will be introduced in next part. I tried different strategies to avoid solver falling into local optimum too early, which includes **roulette wheel selection** , **dynamic parameters** and the **thought of genetic algorithm**.

C. Analysis

The time complexity of Step1, including processing file and Floyd is $O(N^3)$, N is the number of vertices. For Path-Scanning, the complexity is $O(k^2)$, k is number of demands. And for improvement methods, each time, the time complexity is $O(N)$ at most, since in the $single\_insertion$ and $double\_insertion$, it needs to scan all routes to find inserted route, and in other method, if the task(s) picked by algorithm doesn't satisfy the requirement, the method will break and return the original solution. Therefore, the total time complexity is $O(N^3) + O(k^2) + C \times O(N)$, N is the number of vertices, k is number of demands and C is the repeated time of Step3, which depends on the terminal time from online judge, because in the final version of my code, I keep algorithm running until the last 5s.

IV. Experiments

A. Setup

1) Data set: I only used the data sets given by teacher to test my algorithm. I used the local judge to test my output format, used cmd and $onlinejudge$ to test its ability. In addition, I wrote a cost-checker to check whether my output routes is consistent with with output cost. This is efficiency for debugging during coding improvement methods above.

2) Environment:

software:

· python 3.10.6
· numpy 1.23.4
· online usability: SUSTech AI CARP platform

hardware:

· Processor: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30GHz
· RAM: 16GB(15.8GB is available)

B. Improved versions & Results & Analysis

I wrote the approximately 7 versions of CARP_solver during this project, whose characters are listed as $Table1$.

For the first seven rows, "Y" means the solver uses corresponding improvement methods in Step3.

For the row "$roulette$", "Y" means the solver uses **roulette wheel selection** to choose improvement method to execute each time, each method is associated with a weight, $weight/1$ represents the possibility the method is chosen.

For row "$initial\ s$", "1" means only the best one of 5 Path-Scanning solutions can be improved later, and "5" means all 5 solutions will be improved.

Row "$replace\ unit$" is "one" means that, if solver finds a better solution, it replaces current $best\_route$ with the better solution immediately. And "G" means that the replacement operates on a generation. For each generation, the solver will repeatedly pick one solution randomly and do improvement, then put the result back to the generation. And I assigned a limited time for one generation, when the *evolving time* of this generation runs out, solver will pick top M ones becoming next generation's parents.

The last row is "Y" means that, the $weight\_list$ of improvement methods will change during processing time.

TABLE I: Different versions of CARP_solver

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Flipping	Y	Y	Y	Y	Y	Y	Y
Self_insertion	Y	Y	Y	Y	Y	Y	Y
Swap	Y	Y	Y	Y	Y	Y	Y
Single_insertion	Y	Y	Y	Y	Y	Y	Y
Double_insertion	Y	Y	Y	Y	Y	Y	Y
single_2_opt				Y	Y	Y	Y
cross_2_opt					Y	Y	Y
roulette			Y	Y	Y	Y	Y
initial s	1	5	5	5	5	5	5
replace unit	one	one	one	one	one	G	G
dynamic parameters							Y

The performance measures are the different inputs given by teacher. And the output results show ability of different version solvers. The smaller the output cost, the better this solver is. I tested the solvers both in *cmd* and *online judge*. The table listed below are the best result in *online judge* among several submissions.

TABLE II: Best performance of different CARP_solver

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
gdb10	288	288	288	288	288	288	288
gdb1	345	345	342	316	329	316	316
egl_s1_A	5862	5759	5607	5488	5482	5226	5196
egl_e1_A	3978	3958	3908	3969	3808	3896	3749
val7A	323	323	312	309	311	311	305
val4A	428	428	428	431	428	428	418
val1A	188	188	188	187	186	186	186

The overall test results showed that, for each time, adding a new optimization appropriately can really make the solver perform better. Comparing the result of v_1 and v_2, we can find that the solver v_1 stuck in local optimum so early, while promoting all 5 initial solutions and choose the best solution finally can indeed improve solver's ability. However, v_2 just perform better in large graph relatively, it's still a less capable one.

Solver v_1 and v_2 only execute five improvement methods in turn for many times. To avoid the solution falling into local optimum too early again, I add the **roulette wheel selection** in solver v_3 and later versions. In my opinion, the weights of $Swap()$, $Double\_insertion$, $cross\_2\_opt$ higher, the chance of jumping out of the local optimum higher. So, I designed a $weight\_list$ to assign the possibility of each improvement methods to be chosen. From the results, **roulette wheel selection** did improve the solver's performance.

For the further promotion, I add $Single\_2\_opt$ in solver

v_4 and $Double\_2\_opt$ in v_5. The results show that it has obvious optimization effect, especially on large graph.

In previous versions of solver, it will replace the $best\_route$, $best\_cost$, $best\_load$ immediately as long as it finds a better solution, which is also very easy for solution to fall into local optimum. Therefore, I assigned 5 initial solutions from path-scanning to be the first generation's parents. And assigned a time limit e seconds for one generation to evolve. The solver will repeatedly choose one solution from generation, do improvement and put it back. When e seconds times out, solver will pick top M solutions as next generation's parents. However, the choice of **hyperparameters** e and M is not easy. Some experiments' results listed following:

TABLE III: Influence of hyperparameters e and M

e / M	10/30	10/20	20/30	15/30	5/30
gdb10	288	288	288	288	288
gdb1	316	316	316	316	316
egl_s1_A	5226	5420	5492	5352	5479
egl_e1_A	3896	3965	3849	3837	3686
val7A	311	311	311	311	305
val4A	428	428	428	428	428
val1A	186	185	186	186	186

From the table, we cannot tell that which is the absolute best parameter combination. Actually, hyperparameters e,M and *weight_list* in *roulette wheel selection*, the combination of these three factors determines the rate of variation of generation. To some extent, the larger e and M are, the greater the mutation rate is, the greater the possibility of jumping out of the local optimum. However, it has no guarantee that the solution keep evolving in an optimal direction. Therefore, from the idea of last project, I use **dynamic parameters** to control the mutation rate during the executing time.

I divided the total execution time into three phases: Phases 1, I assigned e, M relatively large and *weight_list* appropriately, making variation rate higher to increase the exploratory. Phase 2, I make the variation rate a little bit smaller than the first part of time. And in the last phase, I choose the best solution in two phases before and improve it with a low mutation rate. I did many experiments with different **hyperparameters** e, M, *the time of three phases* and *weight_list* to determine the best parameter combination. The final parameters are as follows:

TABLE IV: Final parameter combination

	time(s)	e
phase1	(terminal_time - phase3_time) / 5 × 3	30
phase2	(terminal_time - phase3_time) / 5 × 2	20
phase3	min(30,terminal_time / 3)	/

	weight	M
phase1	[0,0.05,0.2,0.2,0.2,0.1,0.25]	20
phase2	[0,0.1,0.25,0.2,0.2, 0.1,0.15]	10
phase3	[0.02, 0.2,0.25,0.15, 0.2, 0.03,0.15]	/

In general, the final version of CARP_solver's performance meets my expectation. During this project, I improved my solver step by step. Meanwhile, it's obvious to see the effect of every improved **component** and **hyperparameter** as I analyzed above.

The experiment's result is also **consistent with the theoretical analysis in Methodology part**. The experiment result shows that the improvement methods I used do make solutions fall into local optimum, and the strategies I have adopted above slows down this process to some extent.

V. Conclusion

A. Summery & Disadvantages

In this project, I implement CARP_solver using the **Floyd** algorithm, **Path-Scanning** and **several local search algorithms** to improve the solution. I improved the solver step by step: adding more improvement methods, using the idea of genetic algorithm, changing the mutation rate dynamically and so on. After practicing different version solvers in *cmd* and *online judge*, I adjusted and selected the one with the best performance. However, it still has some limitations:

1) : The local search methods in my solver are not effective for the small graph input. It still falls into local optimum too early. For example, 7 versions solvers all found the result of input-file *gdb10* is 288, whereas others can find 275.

2) : The parameters combination are not adjusted to the best. It still has room for tuning.

3) : My solver is a little bit sensitive to the *random seed*. Using different random seeds, the performance will vary.

B. Gains & Regrets

During the process, I am familiar with reading file and processing data in python. What's more, this project has trained my ability to think independently and solve problems in an all-round way. I keep thinking how can I further optimize the solution step by step in this project. And every time I completed a small optimization, I feel a sense of accomplishment when I see an improvement in performance. However, because of time limitation and academic pressure, I didn't try to write some other algorithms like simulated annealing, which I am desired to try. I am a little bit regret about it.

C. Future Work:

(1) Read the literature related to CARP problem, for example, the reference article [2], to explore other algorithm to improve my CARP_solver.

(2) Try to code other algorithm like simulated annealing algorithm and genetic algorithm, to see whether the solver can perform better.

References

[1] Ángel Corberán and Gilbert Laporte. Arc routing problem method and application. volume 410, pages 134–135, 10 2016.

[2] Luís Santos, João Coutinho-Rodrigues, and John R. Current. An improved heuristic for the capacitated arc routing problem. *Computers Operations Research*, 36(9):2632–2637, 2009.