CS311 Project 2 Report

Kangrui Chen

I. Introduction

Capacitated Arc Routing Problem, CARP for short, is a combinatorial optimization problem which is usually used to model problems that is similar to assigning vehicles to service edges and minimizing the total cost. Some of the real cases are well-known such as garbage collection and snow removal.

As a NP-Hard problem, CARP can not be solved easily using deterministic algorithm. In this project, I tried to designed a GA (genetic algorithm), a kind of local search algorithm, to solve the required problem, which is simplified to a question that have only one depot and can use unlimited number of vehicles. And I found GA work in optimizing initial solutions.

II. Preliminary

CARP can be described as follows: consider an undirected connected graph $G = (V, E)$, with a vertex set V and an edge set E and a set of required edges $T \subseteq E$. A fleet of identical vehicles, each of capacity C, is based at a designated depot vertex $v_0 \in V$. Each edge $e \in E$ incurs a cost $c(e)$ whenever a vehicle travels over it or serves it (if it is a task).Each required edge (task) $t \in T$ has a demand $d(t) > 0$ associated with it. The objective of CARP is to determine a set of routes for the vehicles to serve all the tasks with minimal costs while satisfying:

1) Each route must start and end at $v0$;
2) The total demand serviced on each route must not exceed Q;
3) Each task must be served exactly once (but the corresponding edge can be traversed more than once).

As the shortest path between any two vertices can be found by polynomial algorithms. Here we represent a solution for CARP as a sequence of tasks which the vehicles serves one by one: $s = (R_1, R_2, ..., R_m)$ where R_i is a route and m is the number of routes(vehicles). The k^{th} routes $R_k = (t_{k1}, t_{k2}, ..., t_{kl_k})$, where t_{ki} and kl_k denote the i^{th} task and the number of tasks for R_k. And the direction of each task should be specified so we can add the shortest path from the end of the previous task to the start of the next one while calculating costs: $t_{ki} = (head(t_{ki}), tail(t_{ki}))$

The formulated form of constraints are given below:

- $\sum_{k=1}^{m} l_k = |T|$
- $t_{k_1 i_1} \neq t_{k_2 i_2}, \forall (k_1, i_1 \neq k_2, i_2)$
- $t_{k_1 i_1} \neq inv(t_{k_2 i_2}), \forall (k_1, i_1 \neq k_2, i_2)$
- $\sum_{k=1}^{m} \sum_{i=1}^{l_k} d(t_{ki}) \leq Q$
- $t_{ki} \in T$

Now we define the total cost of a solution, the target to be minimized, as:

$$TC(s) = \sum_{k=1}^{m} RC(R_k)$$

$RC(R_k)$ is the route cost of route R_k, which can be computed as follows:

$$\begin{aligned} RC(R_k) = &\sum_{i=1}^{l_k} c(t_{ki}) + dis(v_0, head(t_{k1})) \\ &+ \sum_{i=2}^{l_k} dis(tail(t_{k(i-1)}), head(t_{ki})) \\ &+ dis(tail(t_{kl_k}), v_0) \end{aligned}$$

$dis(v_i, v_j)$ is the cost of shortest path from v_i to v_j $(i \neq j)$.

III. Methodology

A. General Workflow

The whole process is mainly consist of 3 stages. The first one is to run shortest path algorithm to get the smallest cost between any two vertices. The second is to generate initial solutions, in which greedy algorithm $Path\ Scan$ is used. The last part is local searching, aiming to optimize initial solutions through a stochastic $Genetic\ Algorithm$.

B. Detailed Algorithm/Model Design

Since the graph has no negative edge, stage one can be easily achieved with $Dijkstra$, $Floyd$ or any other shortest path algorithm. So there is no need to zoom in.

Algorithm 1 Path Scan

```
GetNext(R, G, T_rest)
    select accordingly t ∈ T_rest
    return t

Scan(G, T)
    sol ← empty
    T_rest ← T
    while rest ≠ empty do
        r = empty
        while true do
            t ← GetNext(r, G, T_rest)
            if t = null
                break
            T_rest.remove(t)
        sol.append(r)
    return sol
```

As for stage two, to generate initial solutions, one of the most intuitive idea is to continuously select an unserved closest demanding edge that fit the constraints and add it into the current route, until there is no demanding edge left or the current route can not take in any more edges. And then start

a new route and repeat the greedy procedure above. In order to generate variant initial solutions, different strategies can be apply to edge selecting, such as closest rule, highest performance rule and half-far-half-close rule. Also, by modifying some parameters and conditions or using more strategies, we can easily prepare additional initial solutions for later use.

Algorithm 2 Genetic Algorithm

Mutate$(\mathbf{sol}, \mathbf{micro})$
  select randomly $type \in TYPES$
  if $rand() < rate[type]^{micro}$ **do**
    return sol
  mutate according to $type$
  return sol

Reproduce$(\mathbf{pool}, \mathbf{micro})$
  for $sol \in pool$ **do**
    for _ **from** 1 **to** num **do**
      $child \leftarrow Mutate(sol, micro)$
      if $child < sol$ **or** $rand() > DEATH$ **do**
        $pool.append(child)$
  return $pool$

Select$(\mathbf{pool}, \mathbf{K}, \mathbf{size}, \mathbf{p})$
  if $len(pool) < K$ **do**
    return $pool$
  $newpool \leftarrow empty$
  $sort(pool)$
  $newpool.extend(pool[: size \times p])$
  $shuffle(pool)$
  $newpool.extend(pool[: size \times (1-p)])$
  return $newpool$

GA$(pool, K, size, p, micro)$
  while $runtime < TERMINATION$ **do**
    $reproduce(pool, micro)$
    $select(pool, K, size, p)$
    $micro \leftarrow micro \times 0.99$

Once the initial solutions are ready, we can add them into the gene pool. Now consider how to mutate, reproduce and select. To meet validity constraints, I only design mutation inside a solution. There are five types of mutation that I had implemented: reversing a demanding edge, swapping two edges in one route, move and edge from one route to another, swapping two edges from different routes, and adding an empty route to the solution. Different mutation type has different mutation rates. Also, after mutation, if the children get worse results, chances are that they can not survive and join the pool. In each iteration, every solution in the original pool will produce num probably mutated children into the new pool while itself will be abandoned. To simulate the population in the nature more, I set a limit for the population size, named K. Once the size of the pool hits K, which always needs less than ten iterations, it will trigger a selection. In selection procedure, the solutions in the pool will be sorted by cost, and the optimal ones are preferred. After each selection, $size$ solutions will remain in the pool and be ready to reproduce the next generation.

C. Analysis

However, many local search algorithm have the risk to be stuck at a local optimum, and so does the GA. To prevent premature, in selection, rather than simply pick the first several solutions as the new beginning of the new generation, I randomly add some other survivors to enrich the gene pool as many better solutions may need worse or even some invalid solutions as stepping-stones. Besides, after each iteration, I slightly reduced a parameter named $micro$, which enlarge the mutation rates as time goes by.

Noticed that it is allowed to use multi-process to accelerate. Choosing different parameters for the processes can make a better use of this technique. And thus gain a better robustness.

So ideally, the proposed algorithm can perform well with initial solutions and have the ability to resist precocity to some extent.

IV. Experiments

A. Setup

The only dataset used is the given examples and the parameters in the algorithm are chosen manually.

The experiment environment is listed as belows:

TABLE I: Environment

python	3.9.16
numpy	1.23.3

The parameters for each type alongside with multi-processes are:

TABLE II: Mutation Rates

	0	1	2	3	4
0	0.86	0.8	0.85	0.9	0.05
1	0.8	0.9	0.75	0.65	0.05
2	0.78	0.9	0.8	0.7	0.02
3	0.75	0.85	0.75	0.65	0.05
4	0.75	0.8	0.85	0.75	0.05
5	0.75	0.9	0.85	0.9	0.05
6	0.8	0.9	0.75	0.7	0.05
7	0.93	0.93	0.93	0.93	0.07

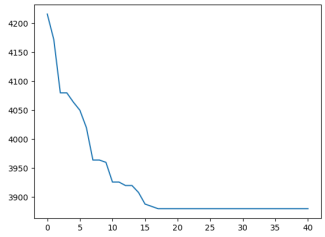
TABLE III: Other Hyper Parameters

	K	size	death rate
0	666	61	0.94
1	888	78	0.91
2	555	64	0.92
3	666	69	0.93
4	777	21	0.94
5	777	44	0.91
6	555	37	0.96
7	999	49	0.86

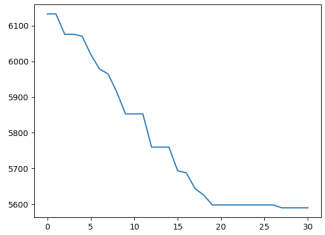
Note: The initial gene pools are also different.

$micro$ is set to 1 (no influence) in the very beginning. And $p = \frac{2}{3}$ (used in $select()$).

B. Results

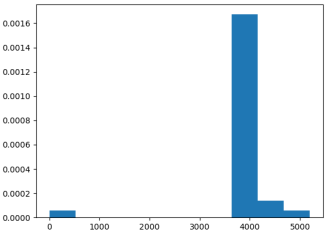


(a) Fig1

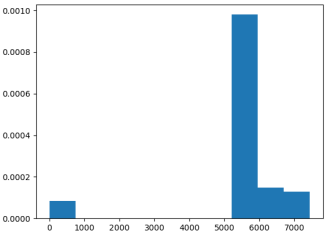


(b) Fig2

Fig. 1: Cost vs. Iteration Time



(a) Fig1



(b) Fig2

Fig. 2: The Final Pool

C. Analysis

Though the dataset is small, the results still tell the effectiveness as the two relatively larger above. Noticed that difference process perform differently. In the process of adjusting parameters, I found mutation rates, death rate and the ratio between size and K can influence the converging speed and the possibility to find better solutions prominently. In other word, these parameters have the potential for being abstract as radicalness and moderateness. If the parameter combination, such as high mutation rates, low death rate and low $size - K$ ratio, is kind of radical, t is supposed to expand more possible solutions. Nevertheless, the one that is relatively moderate is expected to converge faster to the local optimum.

At the beginning, I set size to be hundreds and K thousands, causing the algorithm ran slow. Considering in real cases the demanding edge usually will not exceed one or two hundred. Then $size$ and K don not need to be too large. After adjusting while keeping the ratio remain almost the same, the answer did not change but the process did accelerate to some extent. Balancing num and p is quite a hard and implicit experience. The sweet spot is quite unclear and varies from case to case. And initial solutions do play a decisive

However, the most severe hazard is still the so called premature" risk. Though the mutation rates have been enlarged so that close to 1 while the death rate has also been reduced approximately by half. In the figures that show the final gene pools, the local optimum in the pool is still quite overwhelming. No wonder the answer was stuck at the local optimum for so long (almost for whole rest of the time if the termination limit is extended). My superficial guess was that in the function $reproduce()$, once the old $pool$ have a chance to generate many same better solution (especially in the early iterations), it will give a burst to too many children with the same route. Then the selection will further unify the pool, making it much less likely to find new solutions.

V. Conclusion

In this project, I designed a genetic algorithm to solve simplified CARP. It effectively optimized the initial solutions generated by greedy algorithm significantly. Though I tried to make some adjustment on both the structure and the parameters to relief the premature problem, it still inevitably fell into the trap of local optimum. If further improvement is need, I mainly would like to implement two more functions. First, figure out how to solve the premature problem. My tentative idea is to modified $select()$ to limit the repetition of the same solutions. It can also refer to highly converged population been destroy by a disaster like climate change and disease. The second is to make dynamic parameter adjustment, let the algorithm take turns to be radical or moderate, in order to discover more possible solutions while not missing local optima.

In general, GA algorithm is easy to start for problems that has a fixed elements in a possible solution (such as demanding edges/routes in CARP). But it shares the shortcoming that most local search algorithms have. Using a template is not enough for real world questions. Algorithm designers should create variant proposals. Also, experience knowledge and parameters adjustment is necessary for these algorithm. Combination use of different algorithms is also promoted.

In addition to the content about AI, from this report I learnt to use command line argument and multi-process/multi-thread (if interpreted by cpython, threads can not make use of multi-core to run parallelly) in python and write my first document in LaTeX.

VI. References

(no reference)