Reversed Reversi: the first project of Artificial Intelligence

Lin Yuhang

***Abstract*—Artificial Intelligence has developed rapidly in recent centuries. In the original period of whole developing history, a problem name Capacitated Arc Routing Problem (CARP for short) is studied overall. This problem is a weel known combinatorial problem that requires the identification of the minimum total distance travelled by a set of vehecles to serve a given set of roads subject to the vehicle's capacity constraints. Plenty of algorithms have been and is being issued, in which the heuristic searching methods are most. This paper will introduce a basic genetic algorithm to this problem. Base on the legal solutions produced by an already-exist method "Path Scanning", keep reproducing new possible solutions and filter illegal solutions. Specifically, five ways of reproducing are shown: flip, single insertion, multiple insertion, sway, 2-opt. A new method of preprocessing is also introduced. Although it may produce better solution, since it contracts with online judge system, it won't be implement in code.**

***Index Terms*—CARP, AI, genetic algorithm, Path Scanning.**

I. Introduction

ARTIFICIAL Intelligence (AI for short) is a new study area rising in recent centuries, as has developed and improved rapidly thanks to improvements of calculating speed and hard ware. It aims to construct a machine which has its own intelligence and ability of studying, learning, and evolution by itself.

In the overall history of AI, there is one class of problem study much: routing problem. In general, a detailed route problem is that, given a graph containing all information about each vertex and each edge, to find a solution to make one attribute smallest or largest under certain judgment system.

A. Arc Routing Problem

Routing Problem can be divided into two main directions. One is to regard vertices as serve object, whose name is Vertex Routing Problem (VRP). Another is to regard edges as serve object, whose name is Arc Routing Problem (ARP). What this paper focus is a subclass of Arc Routing Problem.

The characteristics of Arc Routing Problem is that, a set of vehicles start from a given depot, and serve all edges which has demand to handle. It can be divided into three main subclass: Chinese Postman Problem (CPP), Rural Postman Problem (RPP), and Capacitated Arc Routing Problem (CARP). [1]

The CPP is defined by Guan in 1962[3], it's a problem of finding a cycle of minimal total cost passing at least once through each edge of an undirected graph, which is NP-hard under a mixed graph.

The RPP is introduced by Orloff in 1974[4], in which only a subset of the required edges must be traversed rather than all edges, and the main purpose ot RPP is to find the minimal cost of time to traverse edges. This problem is also NP-hard.

The CARP is introduced by Golden and Wong in 1981[2], the purpose of which is to determine a route for each vehicle such that the total cost is minimum and all positive demand arcs or edges are served. The total demand of each route must be less than or equal to the capacity of corresponding vehicle. And the undirected or directed CARP is always NP-hard.

B. CARP

CARP is defined on undirected connected graph.Each edge, also called arc, in this graph, has a non-negative cost and a non-negative demand. In detail, cost is the price to pay when traverse the edge, demand is the amount need to serve in this edge. One specific vertex in the graph is appointed as depot. a given number of vehicles are put in this depot at the beginning, each vehicle has its own capacity, the load of one vehicle can't exceed its capacity.

More specifically, the CARP problem follows rules:

- When a vehicle traverse an edge, it need to pay the cost corresponding to that edge.
- When a vehicle traverse an edge, it can choose to or not to serve this edge.
- If vehicle serve one edge, the demand of that edge will become zero, and the load of vehicle will increase by the amount of demand in this edge.
- If vehicle choose not to serve this edge, then both demand of edge and load of vehicle will not change.
- Each demand edge can only be served totally and only can be served once, its demand is an overall and can't be divided.
- All demand edges must be served.
- Whether or not vehicle serves edge, vehicle must pay price at amount of cost in corresponding edge.
- All vehicles have the same capacity.
- During the whole process, all vehicles need to meet the requirement that vehicle's load can't exceed its capacity.
- At any time, a vehicle can go back to depot to unload.
- After unload, the load on vehicle will become zero.
- The passed edges in the process of go back to depot will also be consider as price to pay.
- After serving all demand edges, vehicle must return back to depot.

The purpose of CARP is to minimize the total cost to pay while satisfying all rules above.

II. PRELIMINARIES

To formalize problem , some terms need to be introduced, all are listed in Table I, Table II, and Table III.

A. CARP Formulation

TABLE I: Notations for formulation, part I

Term	Explanation
G	Given undirected graph
E	Edge set, consist of all edges in G
e	Directed edge
e'	Undirected edge
e_i	A directed edge identified by i
e'_i	The undirected edge having same endpoints as e_i.
v_d	The depot vertex
$head(e_i)$	A vertex, the head of edge e_i
$tail(e_i)$	A vertex, the tail of edge e_i
p	A directed path, may consists of multiple edges. If consists more than one edges, then all included edges are connected end to end. Presented as: $$p = \bigcup_{k=1}^{m} e_k$$ satisfying $$m \in \mathbb{N}$$ If $m \geq 2$, then need to satisfy another condition: $$\forall k \in [1, m-1], tail(e_k) = head(e_{k+1})$$
p_i	A directed path identified by i.
$p(v_i, v_j)$	The directed path start from v_i and end at e_j. Among all satisfied path, no other path has less cost sum of all edges on itself than this path. v_i is a vertex identified by i, v_j is a vertex identified by j.
$cost(e_i)$	The minimal cost need to pay for the path traversing from $head(e_i)$ to $tail(e_i)$. If this path contains more than one edge
$dmd(e_i)$ $dmd(e'_i)$	The demand need to serve on edge e_i or e'_i, "dmd" is short for "demand".
cap	Capacity of vehicle. All vehicles are same.
r	r is a route start and end at depot. r consists of many directed edges, all demands are not zero. m is the number of edges in r, is an attribute of r rather than another requirement parameter. r is also called "route" in natural language in this paper. Presented as: $$r = \bigcup_{k=1}^{m} e_k$$ satisfying: $$\begin{cases} {e_k}' \in E \\ m \geq 2 \\ m \in \mathbb{N} \\ head(e_1) = v_d \\ tail(e_m) = v_d \\ \forall k \in [1, m-1], dmd(e_k) > 0 \\ \sum_{k=1}^{m} dmd(e_k) \leq cap \end{cases}$$

TABLE II: Notations for formulation, part 2

Term	Explanation
$cost(p_i)$	Given $$p = \bigcup_{k=1}^{m} e_k$$ then $$cost(p_i) = \sum_{k=1}^{m} cost(e_k)$$
r_i	A route identified by i.
$head(p_i)$	The start vertex of path p_i.
$end(p_i)$	The end vertex of path p_i.
$e(v_i, v_j)$	An edge e_{ij} satisfying $$head(e_{ij}) = v_i, tail(e_{ij}) = v_j$$ When using term $e(v_i, v_j)$, it is guarantee that such edge exists.
$head(r_i)$	Given $$r_i = \bigcup_{k=1}^{m} e_k$$ then $$head(r_i) = head(e_1)$$
$tail(r_i)$	Given $$r_i = \bigcup_{k=1}^{m} e_k$$ then $$head(r_i) = head(e_m)$$
$cost(r_i)$	Given $$r_i = \bigcup_{k=1}^{m} e_k$$ then $$cost(r_i) = \begin{cases} \text{if } m = 1 \text{ then:} \\ \quad cost(p(v_d, head(r_i))) \\ \quad + cost(e(head(r_i), tail(r_i))) \\ \quad + cost(p(tail(r_i), v_d)) \\ \text{if } m \neq 1 \text{ then:} \\ \quad cost(p(v_d, head(r_i))) \\ \quad + \sum_{k=1}^{m-1} cost(p(tail(e_k), head(e_{k+1}))) \\ \quad + \sum_{k=1}^{m} cost(e_k) \\ \quad + cost(p(tail(r_i), v_d)) \end{cases}$$
R	$$R = \bigcup_{k=1}^{m} r_k$$ R consists of many routes. m is the number of routes in R, is an attribute rather than a known requirement. R is also called "routes" in natural language in this paper.
R_i	A "routes" identified by i
$cost(R_i)$	Given $R = \bigcup_{k=1}^{m} r_k$,then $$cost(R_i) = \sum_{k=1}^{m} cost(r_k)$$

TABLE III: Notations for formulation, part 3

Term	Explanation
d	Number of vehicles in CARP
h	A vehicle
h_i	The i_{th} vehicle among all vehicles, satisfying $\begin{cases} i \in \mathbb{N} \\ i \in [1, d] \end{cases}$
$x(e_i, r_i)$	Number of timse that edge e_i appears in route r_j.
$y(e_i, R_j)$	Number of times that edge e_i appears in routes R_j. Given $R_j = \bigcup_{k=1}^{m} r_k$ then $y(e_i, R_j) = \sum_{k=1}^{m} x(e_i, r_k)$
v	A vertex
v_i	A vertex identified by i.
$in(e')$ $out(e')$	Given an undirect edge e', randomly choose one of its vertices as v_1, regard another vertex as v_2. Then $in(e') = e(v_1, v_2), out(e') = e(v_2, v_1)$
E_{dmd}	The set of all edges with positive demand in G. $E_{dmd} = \{e'_i \in E \mid dmd(e'_i) > 0\}$
R_{min}	The routes with minimal cost among all satisfied routes. This is the searching purpose of CARP.

The purpose of this project is to find routes R_{min} to minimize $cost(R_{min})$ while satisfying the following condition:

$$\forall e' \in E_{dmd}, y(in(e'), R_{min}) + y(out(e'), R_{min}) = 1$$

B. Notations in methodology

Detailed methodology of solution will be shown in III, following terms are introduced to clarify the explanation. All other letters in below table are just simple variables.

TABLE IV: Notations for formulation

Term	Explanation
$mini(v_i, v_j)$	The minimal cost between two vertices v_i and v_j. Can be calculated by Dijkstra or Floyd
$copy(x)$	A function, deepcopy x and return copy
$load$	The demand has handled in route at certain point

III. Methodology

The main idea of solution is to use Genetic Algorithm.There is already an algorithm called "Path Scanning (PS)" to find a legal solution to CARP problem. In this algorithm, total six judging rules on PS and random sorting are used to generate various initial population. In the process of generating next generation, five mutations will be used: flip, single insertion, double insertion, swap, and 2-opt. Mutated solutions will mix with new population, then be filtered through legal detection.

A. Initial Population

PS is an algorithm start at depot vertex, then find the minimal-cost vertex which follows capacity limitation as its next vertex. Then continue to do this until capacity is full or no such vertex exist, as shown in Algorithm 1. $better()$ is one of judging rules, will be explained in III-A2.

Algorithm 1 path_scanning

Input: An integer i, appoint judging rule;
Output: A legal routes;

```
routes ← empty list
free ← all vertices
while free isn't empty do
    route ← empty list
    v_start ← v_d
    while free is not empty or d_min ≠ inf do
        e_min ← empty list
        free_ ← copy(free)
        for ∀e_i ∈ free_ do
            if cost(e_i) + dmd(e_i) ≤ cap then
                if mini(i, head(e_i)) < d_min) then
                    e_min ← e_i
                    d_min ← min(i, head(e_i))
                else if better(e_i, e_min) and d_min = mini(i, head(e_i)) then
                    e_min ← e_i
                    d_min ← min(i, head(e_i))
                end if
            end if
        end for
        append e_min in route
        remove e_min from free_
    end while
    append route in routes
end whilereturn routes
```

Further detail, two mini-distance algorithm and six judging rules are applied.

1) Preparation:

To use PS, minimal distance between any pair of vertices need to be calculated for preparation. Two candidates algorithm are Dijkstra and Floyd.

In theory, Dijkstra is the most efficiency algorithm with infinite big graph. However, in this project, the small sample graph only contains 20+ vertices, the most big graph only contains few hundreds vertices. Time limitation for total runtime is at least 60 seconds. In the scale of so little graph and relatively so long time, theory time complexity is not reliable. Since using more complex data structure, Dijkstra may become even slower than another algorithm, Floyd.

Therefore, in this project, both Dijkstra and Floyd are considerable and reasonable.

2) Judging Rules:

Total six judging rules are used when two vertex have same cost in Algorithm 1, summarized as follows in Algorithm 2.

Algorithm 2 better

Input: An integer i, appoint judging rule; old edge e_{old}, new edge e_{new}

Output: Boolean value, whether e_{new} is better than e_{old};

if $i = 1$ **then**
  Rule 1: choose edge with larger $mini(head(e), v_d)$
else if $i = 2$ **then**
  Rule 2: choose edge with fewer $mini(head(e), v_d)$
else if $i = 3$ **then**
  Rule 3: choose edge with larger $\frac{dmd(e)}{cost(e)}$
else if $i = 4$ **then**
  Rule 4: choose edge with fewer $\frac{dmd(e)}{cost(e)}$
else if $i = 5$ **then**
  Rule 5:
  if $load < \frac{cap}{2}$ **then**
    use Rule 1
  else
    use Rule 2
  end if
else
  Rule 6: randomly pick one edge
end if

Using different judging rules, even the same order of traversing edges may results in different routes, increasing complexity in initial population of GA.

3) Component:

Since can't ensure which rule is best, all of rules above are used and occupy same size of part in population for fairness, and each rule will generate $\frac{1}{6}$ part of population. However, the randomly generated solution may be bad, so six times of required $\frac{1}{6}$ population number will be generated, and the best $\frac{1}{6}$ of them will be picked into initial population.

B. Mutations

In each iteration of GA, old population will mutate and results in some new solution. The idea is to do several changes on current legal solution, like insertion and switching. Total five ways of mutations are used as follows.

1) Flip: "Flip" is, randomly select a route in one solution, then randomly select an edge in this route. Then change the direction of this edge in the route.

2) Single insertion: "Single Insertion" is, randomly choose an edge in a randomly chosen route, then delete this edge in its route, and randomly pick a vertex in all edges, including edges in route that the edge itself locates and edges in all other routes. Finally, insert the deleted edge at the end of chosen vertex.

3) Double insertion: "Double insertion" is similar to "single insertion", except that single insertion only delete one random edge and insert into a randomly chosen location, but double insertion will delete two adjacent edges at the same time and insert these two edges in a randomly chosen location.

4) Swap: "Swap" is, find two different edges randomly, then exchange these two edges' position while not changing their direction.

5) 2-opt: The idea of "2-opt" is shown as follows:

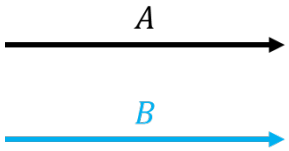

Fig. 1: Initial two route

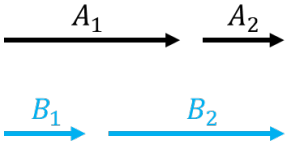

Fig. 2: Cut route

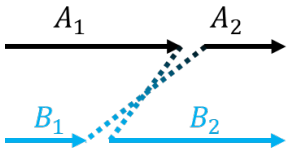

Fig. 3: Reproduced route 1

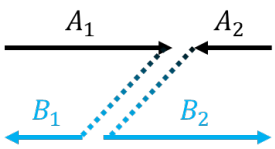

Fig. 4: Reproduced route 2

First randomly select two route as shown in Figure 1, written as A and B. Then randomly choose a vertex on both edge, and cut edge into two parts as in Figure 2, where A_1 and B_1 contains start vertex, A_2 and B_2 contains end vertex. Then we have two ways to re-connect these four parts while maintain each reproduced edge have one start and one end: A_1 connects B_2 and A_2 connects B_1 in Figure 3, or A_1 connects reversed B_1 and reversed A_2 connects B_2 in Figure 4.

C. Selection

In this project, crossover is not considered. The new solution after mutation may be illegal, so a filter needs to be used to delete illegal solution, like exceed capacity limit. In the worst case, all of new solutions are illegal. So the process of generate initial population will repeat again. The next population will be select among mixed solutions from both process of initial population and from mutation.

IV. Experiment

After completing algorithm, some experiments needs to be done to ensure the optimization to algorithm works well, and to find out which parameter is better in practice. Here, each optimization is compared with random decision.

A. Setup

All experiments are implemented on following environment:

TABLE V: Experimental environment

Item	Configuration
Processor	AMD Ryzen 7 4800H with Radeon Graphics
RAM	16.0GB
System	64bit operation systen based on x64 processor
Operation system	Windows 10 Family Chinese Version, with version code 22H2
OS inner version	19045.2311
IDE	Pycharm Professional 2022.3
Python	3.10 version
numpy package	1.21.5 version

B. Test data

The graph for testing are as follows:

TABLE VI: Testing graphs

Graph	Vertex number	Edge number	Demand edge number
egl-e1-A	77	98	51
egl-s1-A	140	190	75
gdb1	12	22	22
gdb10	12	25	25
val1A	24	39	39
val4A	41	69	69
val7A	40	66	66

C. Results

In experiments, flip, single insertion, double insertion, swap, 2-opt are run separately and compared with random selection on each decision in PS. Each competition for each mutation way is run for 100 times, 100 iteration of GA is run in each time. The number of winning and the improve percentage of average cost for each mutation way are shown in Table VII.

All mutation way can reach global optimal cost for graph val1A, val4A, and val7A. If reach optimal cost, limit of mutation can't be observed. So all experiments are run on a relatively large graph: egl-e1-A.

In experiments, the parameter of GA doesn't change for each competition. The parameter used is the final version of adjust. The method of adjust will be explained in

TABLE VII: Result of mutations vs random

Mutation	Mutation win	Random win	Cost improve
Flip	99	1	5.256%
Single insertion	100	0	13.773%
Double insertion	100	0	16.481%
Swap	99	1	19.326%
2-opt	100	0	14.433%

It can be seen that all of mutations can reach better solution than random chosen algorithm. Although random algorithm do win a few times, but consider the large number of experiments, it can be concluded that mutation is better than random under considerable error rate.

D. Adjust parameter

There are some parameters in GA:

- Size of population
- One single solution is selected from how many solutions in process of initial population.
- How many solutions from process of initial population accounts in each generation of population
- How many solutions from mutation in each generation of population.
- How much time a single mutation way perform on a single route
- How many solutions are used for each mutation way.

All the parameters are adjusted by hand. That is, following sense to assign a initial value for each parameter, then run GA on large graph like egl-e1-A for multiple times. Observe the output of each time to determine whether this parameter is better than the previous one.

V. Failed Attempts

In process of project, more methods are done to optimize algorithm. Due to various reasons, part of them failed.

A. Multiple insertion

This idea is inspired by "double insertion". Choose two different vertices in one randomly chosen route, cut off edges between these vertices, and insert into randomly chosen position among all routes.

In theory, multiple insertion is an extension of double insertion and single insertion, so its most optimal cost is better than single insertion and double insertion. However, after programming and test, it turns out that, possibility of better solution is too small that can be ignored.

The problem is that, multiple insertion is "too wide". After some math calculation, even if only do multiple insertion one time at route, the mathematical expectation of changed edges is half of the whole route. That means, in more than 70% percent of cases, the too long cut-down will make route it from and route it insert into illegal, just a waste of time.

If time is infinite for each graph, then multiple insertion theoretically will get better answer than single insertion and double insertion. But because time limitation of this project, multiple insertion is not suitable.

B. PS-PT

A paper[1] in 2022 October introduce a method combining Path Scanning and Partial Tour Building called "PS-PT".

In summary, this method will preprocess whole graph, combine some edges with tiny demand into a large edge. And use this large edge to replace its component edges in original graph, so that these component edges are force to be traversed at same time during PS on edited graph. In paper, this method can obtain most optimal answer in 20% to even 40% cases.

However, after two days of attempts and test, it is found that PS-PT can't be used. This project require to calculate cost on routes. And the cost is base on original graph, so graph itself can't be manipulated to keep demand function and cost function correct. If do Dijkstra or Floyd on original graph and calculate demand in edited graph, then all path passing edited edges will correct demand but incorrect cost. So this method isn't used in project.

VI. Conclusion

Path Scanning based on six judging rules generate initial population in this project, each iteration mutate old population and select best solutions on new generation and mutated results in five ways. All mutations are proven to be better than random. [1]

REFERENCES

[1] BENSEDIRA, B., LAYEB, A., AND ZENDAOUI, Z. A partial tours based path-scanning heuristic for the capacitated arc routing problem. pp. 1–6.
[2] GOLDEN, B. L., AND WONG, R. T. Capacitated arc routing problems. *Networks 11*, 3 (1981), 305–315.
[3] MEI, K. K. Graphic programming using odd or even points. *Chinese Mathematics 1* (1962), 263–266.
[4] ORLOFF, C. S. A fundamental problem in vehicle routing. *Networks 4*, 1 (1974), 35–64.