Carp Report

Feng Chenchen
12011914

***Abstract*—The Open Capacitated Arc Routing Problem (OCARP) is a NP-hard combinatorial optimization problem where, given an undirected graph, the objective is to find a minimum cost set of tours that services a subset of edges with positive demand under capacity constraints. In this paper, a path-scanning algorithm with local searching function is implemented based on Python to solve this problem. Based on path-scanning as a framework, the algorithm adopts random path scanning and genetic scanning to solve the optimal solutions of NP-hard problems. In order to solve capacitated arc routing problem, I first used the Dijkstra algorithm to find the shortest path two-dimensional array of the whole graph. Then, path-scanning algorithm is used to search for the Path that meets the conditions and is closest to the current point each time, combined with random algorithm and genetic mutation to find a short path in capacitated arc routing problem which is a NP problem.**

THE INTRODUCTION

1.1 General introduction:

The Capacitated Arc Routing Problem applies to those edges of a given connected graph that need service. There is a fleet that starts at some point in the network, and all the vehicles in the fleet are the same. Each side must be provided by one vehicle, and the service must be completed at once. All sides are allowed to be passed as many times as desired. Each vehicle starts from the parking lot and returns to the starting point after the service. You want to find the shortest possible path to complete all the services.

The arc path problem can be traced back to Euler's seven-hole bridge problem in 1936, and the classic arc path problem in China is the Chinese post road problem proposed by Professor Guan Meigu (1962), which requires the postman to traverse each side of the undirected graph at least once to find the minimum cost. The arc path problem can be roughly divided into three categories: Chinese post road problem, rural post road problem, capacitated arc routing problem. In this report, we study the capacitated arc routing problem.

CARP is a NP-hard problem, which was first proposed by Golden (1981), and a large amount of literature is based on this problem. Since Goldeng proposed it in 1981, CARP has been widely applied in daily life, especially in municipal services, such as road sprinkler path planning, garbage recovery vehicle path planning, road de-icing vehicle path planning, and school bus transfer path problems.

To characterize the path-scanning algorithm, We can characterize it in terms of six rules when multiple tasks are the closest to the end of current path.

Rule 1: Maximize the distance from the task to the depot.

Rule 2: Minimize the distance from the task to the depot.

Rule 3: Maximize demand(t)/cost(t), where demand(t) and cost(t) are demand and distance cost of task t.

Rule 4: Minimize demand(t)/cost(t), where demand(t) and cost(t) are demand and distance cost of task t.

Rule 5: Use rule 1 if the vehicle is less than half- full, otherwise use rule 2.

Rule 6: Select a node connected to it ran-

domly.

As for the practical implications of carp, I think it can be applied to mail route planning or urban design planning. CARP was developed to solve the route problem of choosing the shortest route to send or dispatch all mail, so it makes sense to apply it to route planning. I want to talk mainly about applying it to urban planning. The essence of a city is the organic combination of many functions, such as schools, hospitals, shopping malls, hotels and so on. We can give different buildings different demand weights, and find the best geographical location of different buildings by using CARP. So as to facilitate the daily travel of citizens as much as possible.

1.2 Purpose of the project:

For this project, my idea is to apply the graph theory knowledge I have learned into reality. Although we have learned a lot of knowledge in the theory class, we have few opportunities to realize it. With the help of this assignment, I can apply the knowledge I have learned to solve the practical problems we often encounter in real life, which I think is quite meaningful.

PRELIMINARY

2.1 Formulation:

The problem can be formulated as a Minimize problem, which is specified by a tuple (I, c_e, $\delta(S)$, $E(S)$, $\delta_R(S)$, $E_R(S)$, $even$, x_e, y_e). The goal of the problem is to minimize the cost. Minimize

$$\sum_{p\in I}\sum_{e\in R} c_e x_e + \sum_{p\in I}\sum_{e\in E} c_e y_e$$

s.t.

$$\sum_{p\in I} x_e = 1 \quad \forall e \in R$$

$$\sum_{e\in R} d_e x_e = 1 \quad \forall p \in I$$

$$x_p(\delta_R(S)) + y_p(\delta(S)) \geq 2x_f \quad \forall p \in I$$

$$x_p(\delta_R(S)) + y_p(\delta(S)) = even \quad \forall p \in I$$

$$x_E \in \{0,1\} \quad y_e >= 0$$

Table 1. Parameter interpretation in the formulation

Symbol	Meaning
$I = 1, 2, ..., K$	set of vehicles
c_e	the cost of edge e which is passed
$\delta(S)$	subset of edge, one vertice in $V - S$, another vertice in $S\subseteq$V
$E(S)$	subset of edge, both of vertices in $S\subseteq$V
$\delta_R(S)$	subset of demand edge, one vertice in $V - S$, another vertice in $S\subseteq$V
$E_R(S)$	subset of demand edge, both of vertices in $S\subseteq$V
$even$	a positive even number
x_e	two element variable, when serving edge e, it's 1, else 0
y_e	number of times passed edge e but not served

METHODOLOGY

3.1 General workflow:

The proposed method divides into steps 1, 2, and 3, each involving algorithms Minimax, alpha-beta pruning optimized Minimax algorithm, and Evaluation function.

3.2 Detailed algorithm design:

Algorithm 1 Minimax

```
function MINIMAX(node, depth)
    if node is terminal node or depth == CutoffDepth then
        return the heuristic value of node
    end if
    if the adversary is to play at node then
        β ← +∞
        for child of node do
            β ← min(β, Minimax(child, depth + 1))
        end for
        return β
    else
        α ← −∞
        for child of node do
            α ← min(α, Minimax(child, depth + 1))
        end for
        return β
    end if
end function
```

Algorithm 2 Alpha-Beta-Search

function ALPHA-BETA-SEARCH($state$)
  $v \leftarrow Max - Value(state, -\infty, +\infty)$
  return the actions in Actions(state) with value v
end function

function MAX-VALUE($state, \alpha, \beta$)
  if Terminal-Test(state) **then**
    return Utility(state)
  end if
  $v \leftarrow -\infty$
  for a in Actions(state) **do**
    $v \leftarrow \max(v, Min - Value(Result(state, a), \alpha, \beta))$
    if $v \geq \beta$ **then**
      return v
    end if
    $\alpha \leftarrow \max(\alpha, v)$
    return v
  end for
end function

function MIN-VALUE($state, \alpha, \beta$)
  if Terminal-Test(state) **then**
    return Utility(state)
  end if
  $v \leftarrow +\infty$
  for a in Actions(state) **do**
    $v \leftarrow \min(v, Max - Value(Result(state, a), \alpha, \beta))$
    if $v \leq \alpha$ **then**
      return v
    end if
    $\beta \leftarrow \max(\beta, v)$
    return v
  end for
end function

state: state of the game, including a chessboard and the color of player.

node: node of the game tree, each node contains a score of one possible step.

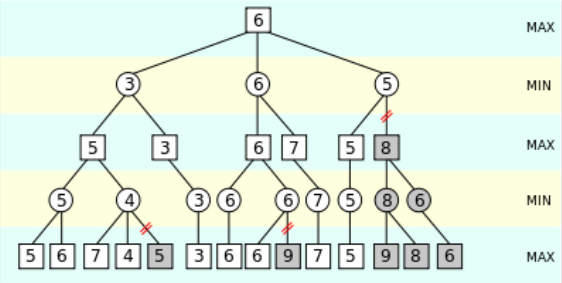

Figure 1. alpha-beta purning

Algorithm 3 Eval-fn

function EVAL-FN($state, step$)
  if step $\leq$ 40 **then**
    $numcoefficient \leftarrow 3$
  else
    $numcoefficient \leftarrow 100$
  end if
  $positionscore \leftarrow Position(state)$
  $mobilityscore \leftarrow Mobility(state)$
  $frontierscore \leftarrow Frontier(state)$
  $numscore \leftarrow Sub - Number(state)$
  $score \leftarrow positionscore + mobilityscore + frontierscore + numscore * numcoefficient$
  return $score$
end function

3.3 Analysis:

In order to make the evaluation function work best, I divided the game into early, middle and late stages. So that different periods of chess have different types of scoring tendency, more in line with the game play.

The time complexity in alpha-beta purning is $O(b^{d/2})$, where b is the branching factor and d is the depth of the tree.

What determines performance is the depth of the tree and the order in which the nodes are arranged.

EXPERIMENTS

4.1 Setup:

In the usability test, I used the test cases provided in the course and found problems in the usual battles. I rely on Reversi platforms to assign me opponents in points contests and round-

robin tournaments. When I modify my program, I test it locally to see if its results are reliable, and then after a few games, I find out what's wrong with my program based on whether the ranking has gone up, and then I improve it. In particular, in order to get the best moves for each step, I competed with the top AI in rank list which uses Monte Carlo tree searching and CNN neural network. Then I recorded his moves of 10 steps in the early stage, middle stage and late stage.

software:Pycharm 2022.1

hardware:Intel(R) Core(TM) i7-10710U CPU @ 1.10GHZ 1.61GHZ

Python:3.10

numpy:1.21.5

4.2 Result:

I used numba acceleration to speed up later chess games by sacrificing the time of the first search. In the first move, numba converts functions into machine code so that the code can bypass the virtual machine and run directly on the CPU. The first move takes some extra time, about 3.5 seconds, as the function compiles to machine code. After that, it takes only about 0.3 seconds to choose a good position and put it into candidate list.

I use course Reversi platform to test the performance of my code. In order to improve the performance of the code, I choose two opponents who are almost as good as me and can play fast. Once I fix the code, I can play with them right away and get feedback quickly.

With numba speedup, I felt the huge speed gap between Python and CPython, so I wrote a simple function to test the speed gap with and without numba. After testing, the speed of numba acceleration is not more than 200 times that of numba acceleration.

I didn't write the original Minimax code, so I couldn't compare the speed of the minimax code with and without alpha-beta purning optimization, but by printing the intermediate process with alpha-beta purning optimization, I felt the speed boost alpha-beta purning got by cutting away a lot of unnecessary branches. And the speed increases exponentially with the depth of the search, so pruning is essential if you want to increase speed.

Table 2. Experimental results

Without numba	With numba	Final Reversi platform rank
1.06s	**0.26s**	97^{th}

4.3 Analysis:

I think my experimental results are just so-so and barely meet my experimental expectations.

In the experiment, I used the SGA algorithm, which stands for standard genetic algorithm. I was surprised to find that the larger the population, the better, a few dozen can have a good effect. At the same time, I also found that the mutation rate is a very important parameter. If you want to get better parameters, you can appropriately increase the mutation rate. Finally, I raised the probability to 0.02.

The ideal time complexity of alpha-beta purning is $O(b^{d/2})$. But in real experiment, the time complexity of it is $O(b^{3d/4})$. It's because the game tree is not ideal. The nodes are arranged in a very random order. If you want to speed searching up, you can pre-process the game tree. But d in here is not so big, so I didn't optimize it.

CONCLUSION

5.1 Characteristics and Realization of Expectation:

Although my algorithm is fast, it does not use enough time per round. Secondly, although my algorithm uses genetic algorithm, the genetic parameters are not well adjusted due to time constraints. But my algorithm also has many advantages. For example, my chess speed is fast, meet the ability to compete with ordinary human chess players. Secondly, my strong AI ability enables me to beat ordinary human players easily, which satisfies my expectation that beginners can learn from AI.

5.2 Gain: Through this project, I gained a better understanding of search algorithms and their application to Reversi, and understood the role of pruning in actual projects. At the same time, I also learned the principle of numba, and the application scenarios and functions of numba.

5.2 Further Improvement:

Continue to optimize the genetic algorithm

and optimize the parameters. Improve the evaluation function so that the AI has a better understanding of the pros and cons of the game.

REFERENCES

1. José M. Belenguer, Benavent E (2003) A cutting plane algorithm for the capacitated arc routing problem. Computers Operations Research 30(5):705-728.

2. Baldacci R , Maniezzo V (2006) Exact methods based on node-routing formulations for undirected arc-routing problems. Networks 47(1):52-60.

3. Diego Pecin, Eduardo Uchoa (2019) Comparative Analysis of Capacitated Arc Routing Formulations for Designing a New Branch-Cut-and-Price Algorithm. Transportation Science 53(6):1673-1694.

4. Hertz, A., Laporte, G., and Mittaz, M. (2000). A tabu search heuristic for the capacitated arc routing problem. Operations research, 48(1):129–135.

5. Polacek, M., Doerner, K. F., Hartl, R. F., and Maniezzo, V. (2008). A variable neighborhood search for the capacitated arc routing problem with intermediate facilities. Journal of Heuristics, 14(5):405–423.

6. Santos, L., Coutinho-Rodrigues, J., and Current, J. R. (2010). An improved ant colony optimization based algorithm for the capacitated arc routing problem. Transportation Research Part B: Methodological, 44(2):246–266.

7. Chen, L., Gendreau, M., H'a, M. H., and Langevin, A. (2016a). A robust optimization approach for the road network daily maintenance routing problem with uncertain service time. Transportation research part E: logistics and transportation review, 85:40–51.

8. Laporte G . Arc Routing: Problems, Methods, and Applications[M]. Society for Industrial and Applied Mathematics, 2015.

9. Belenguer J M , Benavent E . The Capacitated Arc Routing Problem: Valid Inequalities and Facets[J]. Computational Optimization and Applications, 1998, 10(2):165-187.