Project 2: Capacitated Arc Routing Problems

Yuejian Mo 11510511
Department of Biology
Southern University of Science and Technology
Email: 11510511@mail.sustc.edu.cn

1. Preliminaries

This project is a baby implementations of Path-Scanning and Dijkstra's algorithm to solve the capacitated arc routing problems(CARP).

This project is doing some simple implementations on Gomoku. Gomoku, also called Five in a Row, is an abstract strategy board game. It is traditionally played with Go pieces (black and white stones) on a Go board, using 15x15 of the 19x19 grid intersection. [1] Gomoku has less complicity than Go, which has encouraged Deep Mind to develop powerful AI engine AlphaGo in 2016. So Gomoku will be a better particle for beginner. Design of Gomoku AI in this project try to follow the minMax tree with alpha-beta punching, which powered AI Deep Blue to challenge top chess player successfully in the first time.

1.1. Software

This project is written by Python 3.7 with editor Atom and Vim. Numpy library and sys library are used.

1.2. Algorithm

Using Dijkstra's algorithm to calculate the closest pathway between two vertex. Path-Scanning is used to find out task sequence for CARP. I defined three group of functions in order to find out optimal service sequence, including one function to generate cost graph and demand graph between vertex and vertex from provided txt file, two functions to generate shortest distance and pathway dictionary by Dijkstra algorithm, other are control flow and output format function.

2. Methodology

CARP is NP-hard problem. Different chess layout present different chance to win. For example, live-four and double-three must lead to success. So I evaluate the score of possible layout in final level of DFS. For every layout, I scan each line in vertical, horizontal and diagonal to calculate score and sum three together valued as whole chess layout score. Then, all score were judged by minMax tree to found out best choose position. To speed up minMax tree, alpha-beta punching was used to reduce the number of node to expand.

2.1. Representation

Some main data are maintain during process: **capacity**,**graph_dm** , **graph_ct**. Others data would be specified inside functions, shortest_dist should noticed.

- **capacity**: The car's capacity, generated from input file.
- **graph_dm**: The graph of edge with demand and their demand, generated from input file.
- **graph_ct**: The graph of edge with cost and their cost, generated from input file.
- **shortest_dist**: A dictionary of shortest distance and pathway between two vertexes.

2.2. Architecture

Here list all functions in given code:

- **generateGraph**: Generate cost and demand graph from input file.
- **dijkstra**: Calculate all vertexes closest distance and pathway away from specify source.
- **genDijkstraDist**: Generate each two vertexes closest distance and pathway from **dijkstra** function.
- **better**: Second rule to choose one vertexes from two vertex.
- **pathScan**: Path-Scanning algorithm to generate serve sequence.
- **s_format**: Standard output function.
- **__name__**: Main control function.

The CARP_solver would be executed in test platform.

2.3. Detail of Algorithm

Here describes some vital functions.

- **generateGraph**: Generate global cost and demand graph from input file.

Algorithm 1 generateGraph

Input: $input\_file\_name$
Output:
open $input\_file\_name$ as $file$ {open file and read line by line}
$capacity$ ←file.$readline$
{Read each line information until arrive edge information}
for each edge e **do**
  split each line string into an array $line$ with 4 elements.
  if ($line$[3] larger than 0) **then**
    $graph\_dm$[($line$[0], $line$[1])]= $line$[3]
    $graph\_dm[(line[1], line[0])] = line[3]$ {Only add edge to dictionary $graph_dm$when edge has demand. Both of two direction will be created.}
  end if
  $graph\_ct[(line[0], line[1])] = line[2]$
  $graph\_ct[(line[1], line[0])] = line[2]$
end for

- **dijkstra**: generate closest distance and pathway away from source

Algorithm 2 dijkstra

Input: $source$
Output: $dist$, $prev$
create vertex set Q
create distance set $dist$
create path set $prev$
for each vertex v in $graph\_ct$ **do**
  $dist[v] \leftarrow INFINITY$
  $prev[v] \leftarrow UNDEFINED$
  add v to Q
end for
$dist[source] \leftarrow 0$
while (Q is not empty) **do**
  $u \leftarrow$ vertex in Q with min $dist[u]$
  remove u from Q
  for for each neighbor v in u **do**
    $alt \leftarrow dist[u] + graph\_ct(u, v)$
    if alt larger than $dist[u]$ **then**
      $dist[v] \leftarrow alt$
      $prev[v] \leftarrow u$
    end if
  end for
end while
return $dist$, $prev$

- **genDijstraDist**: generate closest distance and pathway between two vertex into a dictionary $shortestDist$

Algorithm 3 genDijstraDist

Input:
Output: $shortestDist$
create dictionary $shortestDist$
for each edge's first vertex in $graph\_dm$ as $source$ **do**
  $dist$, $prev$ ← dijkstra($source$)
  for each edge's first vertex in $graph\_dm$ as $target$ **do**
    $shortestDist[source, target] \leftarrow dist$
  end for
end for
return $shortestDist$

- **better**: choose better in pointers

Algorithm 4 better

Input: now, pre
Output: $Ture$ or $False$
return $graph\_dm[now] < graph\_dm[pre]$

- **pathScan**: Path Scanning algorithm to determine serve routes

Algorithm 5 pathScan

Input: $shortest\_dist$
Output: R, $cost$
  create successive routes R
  copy required edge from $graph\_dm$ to $free$
  $k, cost \leftarrow 0$
  while $free$ is not empty **do**
    $k \leftarrow k + 1$
    $cost\_k, load\_k \leftarrow 0$
    rest successive routes R_k each time
    serve from origin vertex: $source \leftarrow 1$
    while $free$ in not empty **do**
      rest shortest distance: $d \leftarrow \infty$
      rest candidate serve edge: $e\_candidate \leftarrow -1$
      for each edge e in $free$ **do**
        if $load\_k + graph\_dm[e] \leq capacity$ **then**
          $dist\_now \leftarrow shortest\_dist[source, e.start]$
          if $dist\_now \leq d$ **then**
            $d \leftarrow dist\_now$
            $e\_candidate \leftarrow e$
          else if $dist\_now = d \cap$ better(e,$e\_candidate$) **then**
            $e\_candidate \leftarrow e$
          end if
        end if
      end for
      if $e = \infty$ **then**
        BREAK
      end if
      add $e\_candidate$ to R_k
      $load\_k = load\_k + graph\_dm[e\_candidate]$
      $cost\_k = cost\_k + graph\_ct[e\_candidate]$
      $i = e\_candidate.end$
      remove $e\_candidate$ and its opposite edge from free
    end while
    add back home distance: $cost\_k = cost\_k + shortest\_dist[i, 1]$
    add R_k to R
    $cost = cost + cost\_k$
  end while
  return R, $cost$

3. Empirical Verification

Empirical verification is compared with given test_code.py and public test platform.

3.1. Design

Successfully to evaluate chessboard score currently. When depth is two, program output current answer within five second. When I try to run in depth of four, right answer almost take half of hour. In a word, more powerful evalute function and punching(heuristic function) are required.

3.2. Data and data structure

Dict, list I use test data provided by code_test.py to test. Numpy matrix are used widely.

3.3. Performance

Right output in depth of two and time of five second. It run so slow when depth is four.

3.4. Result

Successfully pass all data set test.

3.5. Analysis

The evaluate chessboard layout score cost linear time. minMax tree cost most time. The deeper, the more time it cost.

Acknowledgment

Thanks TA Yao Zhao who explain question and provide general method to solve it. And I also thanks for Kebing Sun discuss algorithm and point the output formation error.

References

[1] Wikipedia contributors, [Online], Available: https://en.wikipedia.org/wiki/Gomoku, [Accessed: 31- Oct- 2018].