CARP Project

Wang Zhiyuan 11610634
CSE
Computer Science and Technology
11610634@mail.sustc.edu.cn

1. Preliminaries

1.1. Software

For this project, I write it by python, and the extra packets I used is numpy.

This project aim to design a excellent enough solution of a Capacitated Arc Routing Problems

1.2. Algorithm

The algorithm I used is Genetic Algorithm, and I design 4 rules to do the pathscanning, design ulusoy, 2-opt, Merge-Split Operator to do the local-search and variation, and use a way that desrible in the paper [**?**] to do the crossover.

2. Methodology

2.1. Representation

In my code, to design the algorithm, I write it into three part:

- $pathscanning$: Do the pathscanning, get the initial solution.
- $localsearch$: For a ordered list, give a best split of it.
- $varition$: For a solution change it and apply the change if the solution can be better.
- $crossover$: Crossover two solution and generate a child solution.

2.2. Architecture

- Read data an storage:
  - Open the filename incoming as the parameter
  - Storage all the node and the cost of edge in a matrix, and this matrix in fact is the adjacency matrix of the graph.
  - Storage all the edge and its demand.
  - Storage all the edge that have demand in a set.
- Use floyd to the adjacency matrix of the graph.
- pathscanning: Random use the step below in each step.
  - maximize the distance
  - minimize the distance
  - maximize the term dem(t)/sc(t)
  - maximize the term dem(t)/sc(t)
- Choice 30 best result in the pathscanning and generate the initial population
- Do MS(Merge-Split) Operator to the population.
  - Merge and do split. Then use ulusory.
- Crossover
- Do MS Operator until the timeout.
  - Merge and do split. Then use ulusory.

2.3. Detail of Algorithm

For the detail of the algorithm, I will introduct it in some parts.

2.3.1. Read data and pretreatment. When we read Data, I read the data from the file given, and storage the graph in two matrix, both two matrices can regard as the adjacency matrix of the graph, but on matrix is for the cost and one matrix is for the demand.

Then, I will storage all the information of the edges that have demand, including the two nodes and the cost. Why we need this set? When we do pathscanning, we need know when we have satisy all the required edges, the node is used to get the destination point and the next depot. The cost is used to calculation the final cost. Because for the required edge, we can't use the adjacency matrix after floyd. Because for the two node of the required edge, there may be other route between two nodes with less cost than the cost of the edge, then the value in the matrix will be the least coat, but we must use the cost in the edge. The last step is do floyed to the adiacency matrix.

2.3.2. pathscanning. After read data and storage, I will do the pathscanning and get the initial population. When I do the pathscanning, I will use the set that I storage the required edges. In each step, I will scan the set and choice the best edge. Firstly, I will choice the closest one, if there is many node have the least cost, I will apply four rules randomly.

algorithm 1 ReadData

Input: The path of the dat file
Output: The information of the dataset, the matrix of map

function BUILDMAP(way)
  $content \leftarrow open(way)$
  $DEPOT \leftarrow 3rd \quad line$
  $CAPACITY \leftarrow 7th \quad line$
  $VERTICES \leftarrow 2nd \quad line$
  $VEHICLES \leftarrow 6th \quad line$
  while $contentisnotend$ **do**
    $edgeProp[] \leftarrow line.split(' \quad ')$
    $edgeProp[0] -= 1$
    $edgeProp[1] -= 1$
    $matrixC[edgeProp[0], edgeProp[1] \leftarrow edgeProp[2]$
    $matrixD[edgeProp[0], edgeProp[1] \leftarrow edgeProp[3]$
    if $edgeProp[3] > 0$ **then**
      $arcs.add((edgeProp[0], edgeProp[1], edgeProp[2]))$
    end if
  end while
  $martixC \leftarrow floyd(matrixC)$
  return $DEPOT$, $VEHICLES$, $VERTICES$, $matrixC$, $matrixD$, $arcs$
end function
function FLOYD(metrixC)
  $n \leftarrow len(matrix)$
  for i from 0 to n **do**
    for j from 0 to n **do**
      for k from 0 to n **do**
        if $matrix[i,j]! = \infty$ and $matrix[i,k]! = \infty$ **then**
          $matrix[k,j] \leftarrow min(matrix[k,j], matrix[i,k] = matrix[i,j])$
          $matrix[j,k] \leftarrow matrix[k,j]$
        end if
      end for
    end for
  end for
  return $matrix$
end function

2.3.3. Local Search. In the local search, I use ulusory split algorithm to do it. In this part, I will ignore the limitation of the capacity and let the solution be an ordered list. Then, I build a tree to storage each possible case of the split, to decrease the time of search, I will confine that it will split just when the car is half-full or provide more demand. And by this way, I will get the best split scheme of this ordered list.

2.3.4. Merge-Split Operator. In the Merge-Split Operator, it's a variation algorithm to a solution. In this algorithm, it will pop some route randomly, and merge them in an unordered task list. For this unordered task list, I will do a pathscanning for it again and insert them back to the

algorithm 2 Do pathScanning

function DOSCANNING($matrixC$, $arcs$, $matrixD$, $CAPACITY$, $DEPOT$)
  while arc is not emoty **do**
    for $edge$ in $arcs$ **do**
      $type \leftarrow random.randomint(0,3)$
      if $type == 0$ **then**
        use rule: maximize the term dem(t)/sc(t)
      else if $type == 1$ **then**
        use rule: minimize the term dem(t)/sc(t)
      else if $type == 2$ **then**
        use rule: maximize the distance
      else if $type == 3$ **then**
        use rule: minimize the distance
      end if
    end for
    if The car is not full: **then**
      $route[carNo-1].append(edge)$
    else
      $carNum \leftarrow carNum + 1$
      $route.append([])$
    end if
  end while
  return $route$
end function

algorithm 3 Local Search

function ULUSOY($trip, matrixC, matrixD, index, cost, CAPACITY, route, DEPOT, carNum, VEHICLES, rr$)
  $cap = 0$
  for i from $index$ to $len(trip)$ **do**
    $cap \leftarrow cap = matrixD[trip[i][0], trip[i][1]]$
    if $cap > CAPACITY//2$ and $cap < CAPACITY$ **then**
      if $i == len(trip) - 1$ **then**
        $rr.append((cost, route))$ break
      end if
      if carNum¡VEHICLES **then**
        $cost\_t \leftarrow matrixC[DEPOT, trip[i][1]]$ +$matrixC[DEPOT, trip[i+1][0]]$ +$cost$ - $matrixC[trip[i][1], trip[i+1][0]]$
        $ulusory(trip, matrixC, matrixD, i+1, cost, CAPACITY, route, DEPOT, carNum+1, VEHICLES, rr)$
      end if
    end if
  end for
  return rr
end function

solution, and then do a local seach for it. Then I can get a best solution for this Merge-Split Operation. This Operator has a big step-size, so one main advantage of the MS operator is its capability of generating new solutions that are significantly different from the current solution.

algorithm 4 Merge-Split Operator

function MS($route_in, matrixC, matrixD, CAPACITY, DEPOT, output_ini, cost_ini$)
    tripChoice = []
    $size \leftarrow random.randint(len(route\_in)//2, 2 * len(route\_in))//3$
    for i from 0 to $size$ **do**
        $index \leftarrow random.randint(0, len(route_in) - 1)$
        $tripChoice.append(route_in.pop(index))$
    end for
    $route\_PS = doScanning() for\ tripChoice$
    $routeFinal = localsearch() for\ route\_PS$
    return routeFinal
end function

2.3.5. Crossover. In the crossover, I choice two solution s1, s2 in the populition as the parents randomly. Then, pop two routes r1 from s1, r2 from s2 and split both of them in two part r11, r12 and r21, r22. The second step is compare r22 and r12, choice the task that r22 have but not in r12, them delete these task in the s1 and r11. Then, joint r11 and r22 and insert into s1. The third step is compare r22 and r12, choice the tasks that r12 have but not in r22, insert these task into s1 one by one, and each task should insert into the location that make the cost be least. The last step is do local search for the s1 and get the best solution for the ordered list in the s1. s1 is the child in this crossover.

algorithm 5 Crossover

function CROSSOVER($matrixC, matrixD, CAPACITY, DEPOT, s1, s2$)
    $r1 \leftarrow s1[randint(0, len(s1) - 1)]$
    $r2 \leftarrow s2[randint(0, len(s2) - 1)]$
    $i1 \leftarrow randint(0, len(r1) - 1)$
    $i2 \leftarrow randint(0, len(r2) - 1)$
    $r11, r12 \leftarrow r1[: i1], r1[i1 :]$
    $r21, r22 \leftarrow r2[: i2], r1[i2 :]$
    for i in $r22$ **do**
        if i not in $r12$ **then**
            $Delete\ \ i\ \ from\ \ s1$
        end if
    end for
    for i in $r12$ **do**

3. Empirical Verification

3.1. Design

3.2. Performance

3.3. Result

3.4. Analysis

[?] [?]