20170400 Saeyoon Oh

CS454, Solving Traveling Salesman Person problem using Genetic Algorithm.

This report is consisted of three parts. First part is about how I tried to approach the TSP problem solving, and how I spent my times (They are mostly about my failures, and what I have learned from them). The second part is about the program itself. How I managed to do best in given time. And the last part will be made out of results and things that I could have done better, and what I want to find out later.

Part I. How I approached TSP problem.

(Since I have learned lots of things through my attempts, I am kind of embarrassed about the thoughts, but still write briefly about what I was thinking. As I go on, I will mark my at-that-time goal with boldface characters)

So first my plan was to **make a simply TSP solver, made out of random searches**. This part included implementing my own Node and Route class, and making the Node instances out of the .tsp file. With the simple TSP solver, I was able to check that my Node and Route classes were correctly designed and implemented, but got results of about 80,000,000 length. The first implementation really got stuck well on the not-really-low local minimums.

Then, my next plan was to **make a slightly advanced version of it, containing crossovers and mutations**. I implemented functions for generating first-generations, and next-generations using the current ones. Ordered crossovers and mutation of swapping random nodes were included. Out of the 100 population, 20 were elites (I set lower mutation rate for elites.), 50 children from the previous generations, and 30 random ones. (If this worked out well, I was going to use simulated annealing.) After about few hours of running, I found out that the gradient was getting much and much lower was still the answer was about 70,000,000. I had two choices, to tune the parameters, or to change the model a bit more, and that's what I chose.

My thought was, there were too many local minimums, so that I would have to give a quite-appropriate answer to my program. (I think this was the start of my mistake). My thought was since random samples are too – scattered in diverse places**, I would grid-alize (which means I would cut the given points and put them in separate places, to make sub-tsp instances.) and run genetic algorithm on them, and then connect them in some way, and give "the" instance again and get the global minimum.**

I was actually successful in implementing this, and was able to run sub-tsp instances. (At least I thought I was pretty successful) Now the problem was how to connect these. My conclusion was,

for any tsp instance, there were not many types that the subproblems can be connected for optimal solutions. I thought that the optimal solutions can be classified into one of the two categories. Connect by horizontal lines, or connect by vertical lines. Like the figures below.

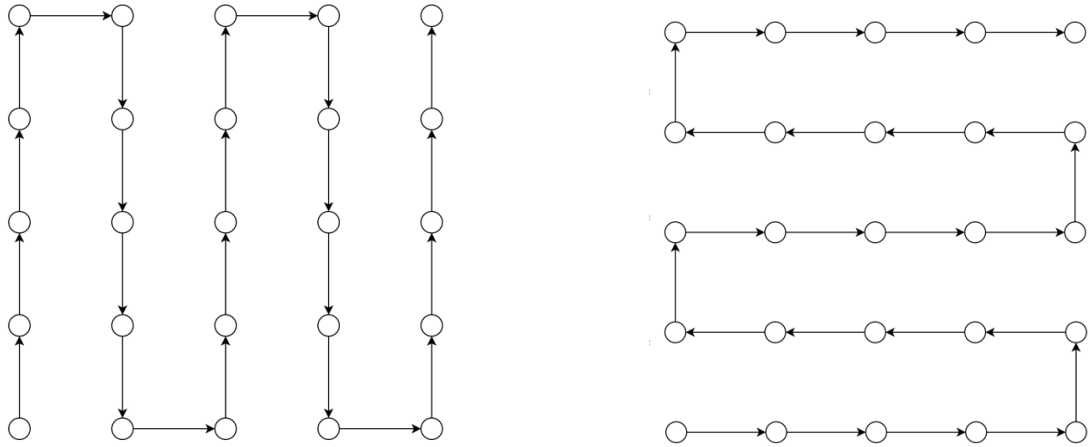


Figure 1: Connecting vertically, horizontally

Of course, I knew that actual optimal solutions would not fit "exactly" in those forms, which is why I also implemented another GA program to find the right sequence of the subproblems to connect with.

After running all these things, I thought that the most important parameter was the size of the grid, which I tuned. For the instance "rl11849.tsp" the best grid that I had was with 67*67 grid. But the problem with this was that the connecting sub-problems gave me length of about 1,800,000, yet this never decreased when I applied another genetic algorithm for how I connect the sequence.

So since the GA for changing the sequence did not work, I thought that tuning the GA for the whole route would be much more nice, and I had some optimizations (which I would explain in the second part.)

Part II. Final implementation

Since part one talks about various trials that I have performed I am going to talk more precise about what features I have kept and what kind of optimizations I have done.
Final program is consisted of 6 python files. I am going to introduce the 6 files, and then talk about how to tune the parameters.

1. **class_def.py**
   This file contains self-defined class Node and Route.

Node refers to each individual node and contains their coordinates, their sequence number and a method for calculating distance of the two different Node instances.
Route refers to each route instance. Route class has a method called "distance" that calculates the total length of the route.

2. **preprocess.py**

   This files contains two functions for pre-processing tsp instance.
   Function **preprocess** takes name of the instance by input and returns numpy array containing of the nodes inside the instance.

   Function **grid_alize** takes in nodearray given by preprcess and number of slice (per axis), and this grid-alizes the nodes. In which as a result, we get nump array containing slice * slice sublists. Each sublists then contains Nodes located in the "grid" position.

3. **sub_prob.py**

   This file contains GA algorithms for the nodes in the grid.
   This is the most important part of the whole program. My algorithm is based on the thought that optimal TSP solution either travels the nodes in horizontal or vertical way. (Like shown in figure 1).

   So my implementation is based on the thought, but we still do not know which way is better. (Well turns out the horizontal version is better for rl11849 instance but for random tsp instance, we do not know) So this GA returns returns two cases, one optimized for vertical-way travel, and the other for horizontal-way travel.

   What do I mean by optimizing for horizontal, or vertical? For each sub-problems, (or in other words, for each grid) I gave orientations. For example, lets take example for vertical-way travel. Since my travel will take place like the left-picture of figure1, for odd-number columns (in the assumption that leftmost one if column 1) want their orientations to be upward. Start from the bottom and end in the top. While the even-number columns want their orientations to be downwards.

   Similar with horizontal-travel. For better optimization, odd-number rows want to be oriented rightwards, while even-number rows want to be oriented leftwards. I have implemented this feature, and running the **run_sub_prob** of the file gives two versions of the travel.

Implementing this feature was actually not so hard, I have sorted the nodes (for each grid) by their x, (or y) coordinates, and I did not move around the first node and the last node when doing crossovers, or mutations. So that the orientation does not get tangled.

For mutation, one strange thing (for me) I found was that for the best performance, the number of nodes per grid is quite low, about 2.5-3.5 nodes. So I set the mutation function to swap around the 6 elements of the chosen one.

Since the most important parameters are slice_num (how many slices per axis for grid), population, mutation rate, and generation, I set up so that user can tune those four parameters. (Other parameters are chosen proportionally to the population)

4. **full_prob.py**

   After running sub_prob.py, the results go through some process in the main_acc.py, which we will talk in detail after. After the process, the sub-answers get connected and now we kind of jumped to the stage where we need to run GA on the full path.

   Features of the GA is quite similar, but the differences are first generation is made purely out of the best sample I have got through sub-problem GA. Also, mutation has a lot more range, around 0.01 * (length of route) elements, and when cross-over happens, I made it that the cross over part has to have length more than (length or route) * 0.2.

5. **main_acc.py**

   This file runs the sub_prob.py and connects the route, and gives it to the full_prob.py.
   The main part of this is putting the sub-answers together. Since we get the horizontal and vertical travel version, I made subsequence for the two travels. Yet, since gluing them takes very small time, I have tried the all four combinations.
   The best sample goes in the to full_prob.py.

6. **tsp_solver.py**

   This is file is for dealing with the arguments and writing the answer on the csv files.
   This reads the filename and makes nodearray, numpy array containing all the nodes in the file, and passes it onto the whole GA program.

   There are 7 arguments. population1, mutationrate1, generation1 are parameters for the first GA, population2, mutationrate2, generation2 are parameters for second GA. Also, parameter slice is the number of slice per axis.

Language I used is Python3 and I used numpy and argparse models. Numpy to use arrays for speed, and argparse to deal with the arguments.

How to use + tune.

The instance must be in the same directory as the all python files. First argument is the name of the instance. -- help flag can give you descriptions about the options.
The most important parameters would be slice and population1, mutationrate1, and generation1. (slice is "the" most important).

I was not able to explore about the characteristics of slice with many instances. So the only information I have is it works pretty well when each grid contains about 2.7 nodes. I set up the number of initial slice to meet the conditions. But user can change it by using option.

Another thing is, for my implementation, after knowing 2.7 is the right number for each grid, I have done some optimizations based on the information. User does not really want to put more than 6 instances in one grid.

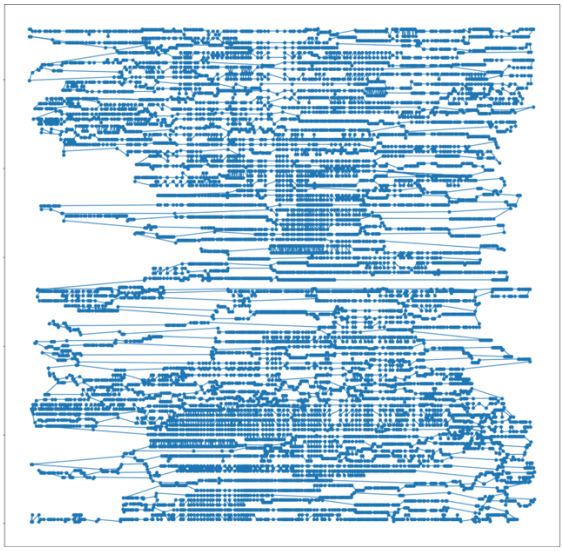
For the best performance, user might want to follow these steps.

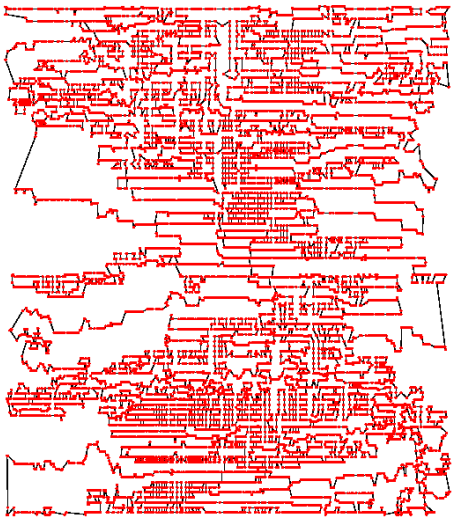
1. Move around slice_num and find the optimized slice_num
2. Tune parameter for first GA
3. Tune parameter for second GA

Current settings are for rl11849, If my program takes very short time for the tested version, please increase the generations for GA1, and GA2. (options are -g1, -g2)

Part II. Results

What was the most surprising was the number in each grid. I was expecting a number of about few hundred for optimized answer, but I found out that a number of few nodes works much better. I first thought that GA could be doing nothing for the first section, but I was able to find out running a bit more generations gave me better results. (running 1 generations give me about 1,600,000 while running 10-20 generations give 1,4000,000) Also due to the nodes in one grid, the population doesn't really need to be large, + mutation rate also does not need to be high. For the second GA, however, since one mutation changes one spot higher mutation rate gives better descending.
The first GA might not give the optimal, but I think this can give you a pretty-good local minimum. On the other hand, the second GA does not really find other local minimums, but rather choose to go down the given dump.
Left is one of the solution I got, while right is the optimal one.





It truly shows that my tour travels horizontally, and there are rarely no exceptions, while optimal one has some tangled. I think I will be able to fix this if I put the GA for changing subsequence in the middle.

One of the possible drawback of my TSP algorithm is that if the tsp nodes are not uniformly distributed, my program is likely to give bad output, since first, the number of nodes in each grid is not even so some grid gets too many nodes, and that for the case optimal solution is likely to not come in vertical, or horizontal way, but in some other weird way. But like before, I think this can be handled by putting GA for subsequence in the middle.

Observing my program, I would say that my program tends to work well for the big tsp instances. (Since big instances tend to have uniformly distributed nodes, plus because my program is good at finding local minimum quicker than other algorithms.)

If I had more time, I would really want to tune and think about the number of nodes in a grid, and maybe think of a way to adjust the number by the look of the map, of by the size of the whole tsp algorithm. Also, like mentioned multiple times, I think swapping some subsequences will result is much better result.