Combinatorial Optimization
Laboratory project
2021/2022

Google Hashcode 2021

Traffic signalling

Author: Daniel Jankowski,

Artificial Intelligence, sem. 3,

Group SI2, index: 148257

Table of Contents

1. Theoretical part

In the project we were asked to prepare two versions of program, which will return a schedule of traffic signalling for given instance of problem. As a simple approach I decided to implement **greedy algorithm**, which will be based on a set of trivial, universal rules for creating a signalling schedule. The idea of greedy algorithm is to be simple, intuitive and make an optimal move at each step. In result we obtain an algorithm, which in short period of time, can generate quite satisfying solution to the given problem. The greedy approach, which I created for this particular problem, I will present later on in Implementation part section.

Second approach, which I decided to use, is a metaheuristic algorithm - **Particle Swarm Optimization.** PSO (particle swarm optimization) is one of the nature-inspired algorithm, which was introduced by James Kennedy and Russell C. Eberhart. The idea of the algorithm is to initialize starting population of particles (population of size N) and choose the best one from them as a global maximum. Next, in each iteration (until a given condition is satisfied) the positions of each particle is changed. The movement of each particle is influenced by its local maximum, but is also guided toward the best known solution (global maximum). The advantage of PSO is it doesn't require the problem to be differentiable and it can be parallelized easily, because each particle can be updated in parallel way. However, since PSO is metaheuristic, there is no guarantee, that solution given by algorithm will be optimal (usually it won't be optimal).

2. Implementation part

2.1 Technical specification

Programming language: **Python 3.8.6**

Modules:

- *time* – to measure time execution of the algorithm
- *random.sample* – to select N random elements from array
- *random.choices* – to choice element with given probability
- *random.shuffle* – to shuffle an array
- *recordclass* – to store information about streets, intersections and schedules.
- *copy*.deepcopy – to make a deep copy of given array.
- *collections*.deque – to store queue of cars, streets etc.

My project contains three python files:

- ***GlobalFunctions.py*** - file with global functions, which are used in both of approaches. This file is imported by *greedy.py* and *ParticleSwarmOptimization.py*. There is no need to run or edit this file. This file contains a functions:
    a) *readInput()* – reading an input from a given file in folder *Instances* and parsing data into record classes. There is one very important feature of this function – for intersections dictionary it parse only intersections with used streets, therefore automatically we save a lot of time omitting scheduling intersections that have no traffic.
    b) *reinit()* – additional function used in main simulation function to reinit streets and intersections after finished simulation

c) *grade()* – main simulation function, which perform the simulation and return score obtained by given schedule.

- ***greedy.py*** – file with implemented greedy algorithm. After starting the program, we will be ask for enter a file name (e.g. "a.txt"). Program will return schedule, score, obtained by algorithm for given problem instance and execution time.
- ***ParticleSwarmOptimization.py*** - file with the PSO implementation. As in case of *greedy.py* we should enter a file name. Program will return schedule, score and execution time.

2.2 Greedy approach

As a greedy approach I decided to apply simple rule for creating a schedule. For given instance we take only intersection with streets, which are included in cars plans. Thanks to that we automatically exclude streets and intersections, which are useless for us, since they are not used by any car. Going further, while we read data from input, we count how many times we use given street. In result, each intersection has a dictionary, which holds information about usage of incomings streets. Based on that, for each intersection we choose the street with maximum value of usage in that intersection (if such a street exist, because if there is a draw between two streets, then none of these is selected) and set the green time 2 second. For other streets we set green time to 1 second.

The advantage of that method is we don't treat all of streets equally and somehow we set a higher priority to the streets with the highest number of driving cars.

2.3 Particle Swarm Optimization

There are four function, which all together form the PSO algorithm. First function named *randomSolution()* generates a random solution of the problem. After testing and analysing several options I decided that giving any street green light duration above two seconds is not profitable, therefore green time is randomly given for street according to the table given below.

Probability	***Green time duration***
90%	1s
10%	2s

Important thing is that for each incoming street we check whether this street is used, if it's not, we just ignore that street.

Another function is *InitialPopultion()*, which is responsible for creating an initial population. Because of computationally difficulty of the problem I decided to creating population of size = 10. This function used *randomSolution()* to generate 10 random solutions, which all together form the initial population.

In case of PSO algorithm, important thing is to change each particle position according to not only its best found solution, but also the global maximum. This is expected to move the swarm toward the best solutions. I was wondering, how in that particular problem move each particle toward to the best found solution and in final version of my implementation I decided that each particle in each iteration will copy randomly selected N intersection's schedule (N is an integer, at the beginning of the algorithm N=1, and it increases at each iteration or reset whether better solution is found) from

actual global and local best found solutions. And for that operation comes third function called *updateParticlePosition()*. This function as a parameters takes actual schedule of the particle (to be more precise, the actual position of the particle), actual best global schedule (from that point of the documentation I will call it ***gBest***, just to simplify), actual local best solution (from that point – ***pBest***, best found position of individual particle, something like local maximum) and two hyperparameters called c1, c2. These two parameters are response for velocity of each particle, so in my implementation, they exactly declare how many intersection's schedules given particle copy from gBest and pBest. To better understand my way of thinking I will present it using very intuitive illustration given below:

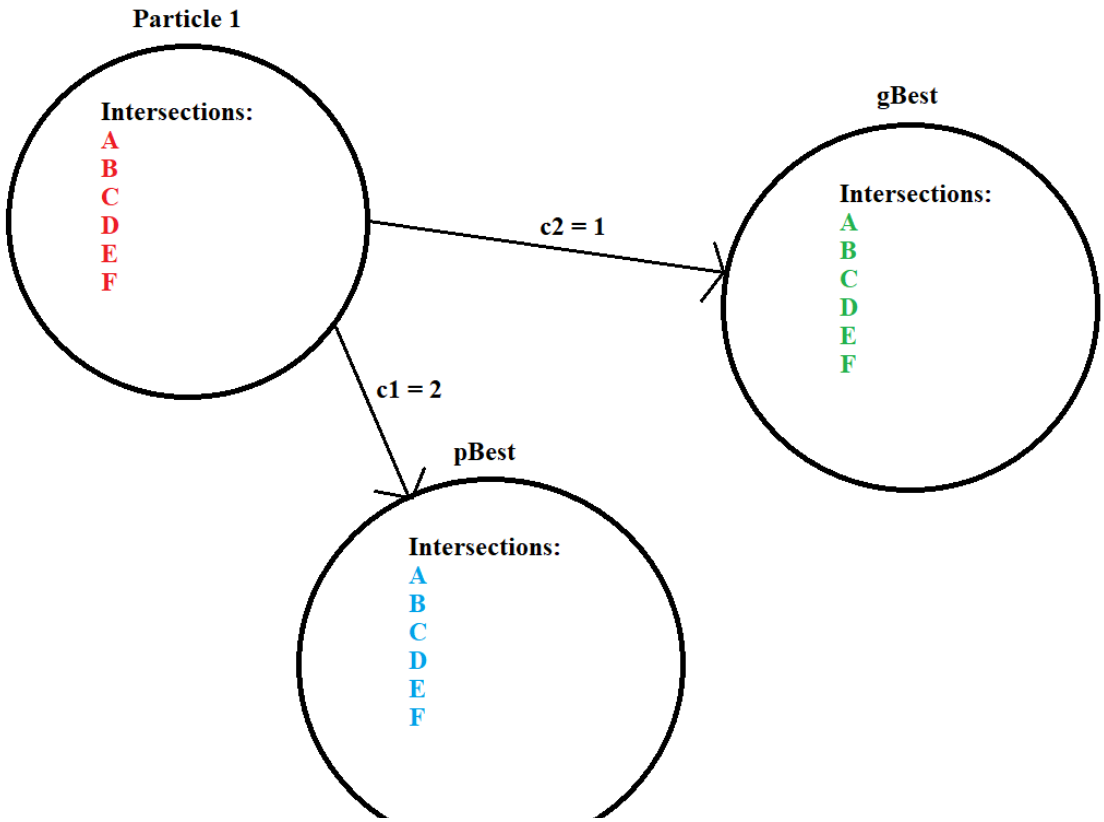

Illustration 1 - the way how we copy intersections from better solutions

Each letter identifying intersection stand for concrete schedule for that intersection. Now we have to copy randomly selected c2 schedules from gBest and c1 randomly selected schedules from pBest. Therefore after changing the position of our *particle 1* it may looks like that:

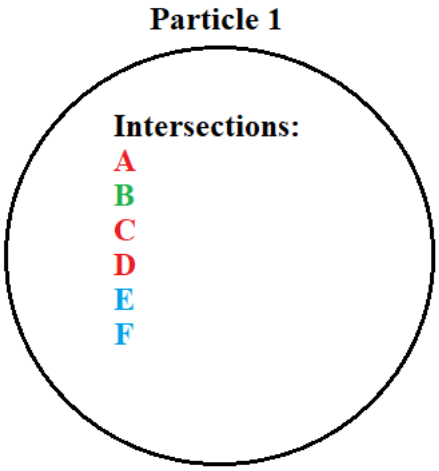

Illustration 2 - an example of new generated solution

Following that rule, there can be a situation, when from pBest and gBest algorithm try to copy the same intersections. To avoid that "conflict" I decided to set higher priority to the pBest solution. This feature can helps swarm to discover and explore new solutions and schedule and doesn't so fast converge into gBest solution. So considering once again situation from *illustration 1*, let's imagine that from gBest we try to copy intersection B and C, when from pBest the only one copied intersection will be C. Then our new generated particle schedule would look like this:

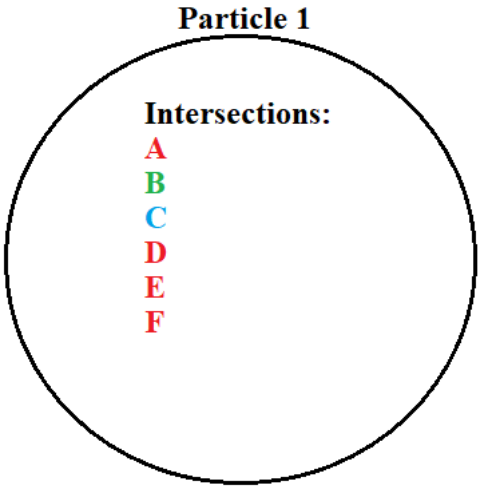

Illustration 3 - an example of generated solution in case of conflict between pBest and gBest

At the end, the only thing is to take care about intersections, which haven't been copied (all in red colour). These intersections shuffle the order of the street or streets green light duration according to the table:

Probability	Action
85%	Shuffle order of streets
15%	Shuffle duration of the green lights

There is very low probability for shuffle the duration of green lights, since in our random solutions there is very low probability to set green light for 2 second, therefore this shuffle in most of cases won't change a lot.

Finally, there is a function PSO, which is the main function response of algorithm working. Below I present the pseudocode for this function:

1. *Initialize the population.*
2. *Set initial position of each particle as its pBest solution.*
3. *Choose the best solution from pBests array and set it as a gBest solution.*
4. ***While** a termination criteria is not met **do:***
    For** each particle i=1, …, N **do:
        UpdateParticlePosition()
        Increase value of c1, c2 parameters by 1
        If** score (new position) > score (pBest) **then:
            Update pBest solution
            Reset c1 parameter for i-th particle
            If** score (new position) > score (gBest) **then:
                Update gBest solution
                Reset c2 parameter for i-th particle
5. *Return gBest, score(gBest)*

As a terminated criteria I set executing time of the program. The main loop runs until the program execution time exceeds 260 seconds (remaining 40 second is for performing last iteration through all the particles). At this point, I want to point one remark, i.e. on my machine, there was no situation that program execution time exceeds 300 seconds, but this situation may slightly change at different computers (different CPU may play a key role here).

Another thing is, that one disadvantages of terminated program due to the time execution is, that algorithm takes almost 5 minutes, even for very trivial instances of the program (e.g. file "a.txt"), despite this I decided that this terminated criteria is the best for that problem (I just wanted to use all the possible time).

3. Conclusion

After conducting multiple tests of algorithms, first of all I'm very satisfied with the results, which my greedy approach obtained (since for my implementation of greedy algorithm, obtained results are always the same I will just print them out below). First version of my greedy algorithm set for each street green light duration equals one, and after improving algorithm to the final version, which I previously described, results were even much better.

Greedy approach results:

- a.txt – 1001
- b.txt – 4 565 781
- c.txt – 1 243 393
- d.txt – 899 855
- e.txt – 663 120
- f.txt – 655 626
- SUM = 8 028 776

In case of my PSO results were of course better that for greedy approach and in average total sum was at level of 9 000 000 points. One of the problem, which I observed, testing my implementation of PSO, was that the improvement of best found solution at each iteration wasn't so big, it of course depends on instance and concrete situation of the algorithm, but in average PSO improves the best solution by 1 000 – 3 000 (at one hop) , what in case of that large instances and possibilities isn't so much. Despite this fact, there were situations, when algorithm was able to improve the solution on the scale of the entire execution by about 300 000, what I think isn't bad result.

4. Sources

https://www.sciencedirect.com/topics/engineering/particle-swarm-optimization

https://machinelearningmastery.com/a-gentle-introduction-to-particle-swarm-optimization/

https://en.wikipedia.org/wiki/Particle_swarm_optimization

https://deap.readthedocs.io/en/master/examples/pso_basic.html

https://towardsdatascience.com/particle-swarm-optimization-visually-explained-46289eeb2e14

https://www.youtube.com/watch?v=JhgDMAm-imI&ab_channel=AliMirjalili

https://knepublishing.com/index.php/Kne-Social/article/view/1394/3208

http://wwwmayr.informatik.tu-muenchen.de/konferenzen/Ferienakademie14/literature/HMHW11.pdf

https://www.intechopen.com/chapters/4609