Particle Swarm Optimization for Bike Sharing Rebalacing Problem

Giacomo Balloccu

Abstract

This paper will discuss a new approach to solve a particular instance of capacitated vehicle routing problem (CVRP). This instance is known in the literature as the Bike Sharing Rebalancing Problem. The paper will start with the introduction to the problem and its definition, an in depth look at the Particle Swarm method, and the logic behind the fitness function chosen in the implementation. Continuing with a discussion about parameters chosen, results obtained, and comparison between this version against a normal solving done with linear methods.

1 Introduction

The problem of Bike Sharing Rebalancing is a particular instance of the family of problems called capacitated vehicle routing problem (CVRP). This particular problem is highly relevant for Bike Sharing companies that want to reduce the amount of money spent for the rebalancing operation of their system.
A bike-sharing system is composed of various stations located in different places inside a city, every station has a fixed number of slots that can be empty or fill with a bike. The first time that bikes are placed among stations they are distributed according to the density of users in that area. In general, there should be a fixed amount of bikes in every station to allow users to leave the bike in a space of the station or take a bike from it. During normal usage every bike is taken by a user from a station and left in another after his trip is finished, this will eventually cause a state where the bikes aren't well distributed around the system and this will cause disservices.
To solve this problem the Bike Sharing company sends in the night vehicles that can pick up and drop bikes in stations making them balanced again. Being an operation that must be done every night it has some importance in the total company costs and has an ecological impact. The goal is to define a single path between stations from the Starting Depot to the Ending Depot so that K vehicles with L capacity minimizing the distance traveled by our vehicles while rebalancing every station.
The same problem was solved in my previous work using linear optimization methods, but

since this problem is known as NP-hard the solver will use a huge amount of ram and time to solve sub-instances that have more than 10 stations. This algorithm instead is developed using a meta-heuristic that will give a solution that is not guaranteed to be an optimal one but in a shorter amount of time.
The next section will give a more detailed explanation of the problem statement. The third section is about PSO methods and the approach proposed for solving while in the fourth I will show some results that will be compared with the ones obtained by the linear solvers and write about future improvements.

2 Problem Statement

Every station has a demand value and its value will define with the kind of node is in the network and the behavior of the vehicle in his regards:

- Demand < 0 this station is demanding bikes, and it will be considered as a dropoff node. The vehicles will drop bikes until the demand will be 0.
- Demand > 0 this station is asking to have fewer bikes, so will be considered as a pickup node. The vehicles will pick up the bikes until the demand will be 0.
- Demand = 0 this station is balanced and doesn't require further operations. The vehicle will ignore it.

In addition to normal stations in this definition, we will have two special nodes the Depots, one which is the starting depot and one which is the ending. So the vehicles starting all from the starting depot need to rebalance the network and finish their route in the ending depot.
Regarding the vehicles is possible to choose how many of them use and the capacity of every single one. The vehicle amount is indicated with K while the capacity vector is defined as L and it is a vector of K size. In general, what happens with this implementation is that the vehicles will behave as a single-vehicle that has a capacity equal to the sum of all vehicle capacity and they will have to move in the same path.
If the vehicles don't have enough bikes on them or they have too many to fulfill the demand of a station they can come back to the starting depot and recharge or empty their size. This is a simplification that will make the problem always feasible because we are supposing that in the depot there is an infinite amount of bike.
The goal is to define a single path between S stations from the Starting Depot to the Ending Depot so that K vehicles with L capacity minimizing the distance traveled by our vehicles while rebalancing every station.

3 Particle Swarm Optimization Algorithms

Particle swarm optimization (PSO) is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality which is calculated with a Fitness Function. It solves a problem by having a population of candidate solutions, called particles, and moving these particles around in the search-space according to simple mathematical formulae over the particle's position and velocity.[1] Each particle's movement is influenced by its local best-known position but is also guided toward the best-known positions in the search-space, which are updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions. PSO is originally attributed to Kennedy, Eberhart, and Shi[2][3] and was first intended for simulating social behavior,[4] as a stylized representation of the movement of organisms in a bird flock or fish school. The algorithm was simplified and it was observed to be performing optimization. PSO is a metaheuristic as it makes few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. However, metaheuristics such as PSO do not guarantee an optimal solution is ever found. Also, PSO does not use the gradient of the problem being optimized, which means PSO does not require that the optimization problem be differentiable as is required by classic optimization methods such as gradient descent and quasi-newton methods.

3.1 Classic PSO

A basic variant of the PSO algorithm works by having a population (called a swarm) of candidate solutions (called particles). These particles are moved around in the search space according to a few simple formulae.[9] The movements of the particles are guided by their own best-known position in the search-space as well as the entire swarm's best-known position. When improved positions are being discovered these will then come to guide the movements of the swarm. The process is repeated and by doing so it is hoped, but not guaranteed, that a satisfactory solution will eventually be discovered.

Formally, let f: $\mathbb{R}$n $\rightarrow$ $\mathbb{R}$ be the Fitness Function or cost function which must be minimized. The function takes a candidate solution as an argument in the form of a vector of real numbers and produces a real number as output which indicates the objective function value of the given candidate solution. The gradient of f is not known. The goal is to find a solution a for which $f(a) \leq f(b)$ for all b in the search-space, which would mean a is the global minimum.

Let S be the number of particles in the swarm, each having a position $x_i \in \mathbb{R}n$ in the search-space and a velocity $v_i \in \mathbb{R}n$. Let pi be the best-known position of particle I and let g be the best-known position of the entire swarm. A basic PSO algorithm is then: [10]

Algorithm 1: Basic variant of PSO algorithm

```
foreach particle i = 1, ..., S do
    Initialize the particle's position with a uniformly distributed random vector:
      $x_i \sim U(b_{lo}, b_{up})$;
    Initialize the particle's best known position to its initial position: $p_i \leftarrow x_i$;
    if $f(p_i) < f(g)$ then
        update the swarm's best known position: $g \leftarrow p_i$;
    end
    Initialize the particle's velocity: $vi \sim U(-|b_up - b_lo|, |b_up - b_lo|)$;
end
while a termination criterion is not met do
    foreach particle i = 1, ..., S do
        foreach dimension d = 1, ..., n do
            Pick random numbers: $r_p, r_g \sim U(0, 1)$;
            Update the particle's velocity:
              $v_{i,d} \leftarrow \omega v_{i,d} + \phi_p r_p (p_{i,d} - x_{i,d}) + \phi_g r_g (g_d - x_{i,d})$;
        end
        Update the particle's position: $x_i \leftarrow x_i + l_r v_i$;
        if $f(x_i) < f(p_i)$ then
            Update the particle's best known position: $p_i ¸ x_i$;
            if $f(p_i) < f(g)$ then
                Update the swarm's best known position: $g ¸ p_i$;
            end
        end
    end
end
```

The values b_{lo} and b_{up} represent the lower and upper boundaries of the search-space respectively. The termination criterion can be the number of iterations performed or a solution where the adequate objective function value is found. The parameters ω, ϕ_p, and ϕ_g are selected by the practitioner and control the behavior and efficacy of the PSO method. LR represents the learning rate ($0 \leq$ lr ≤ 1.0), which is the proportion at which the velocity affects the movement of the particle (where lr = 0 means the velocity will not affect the particle at all and lr = 1 means the velocity will fully affect the particle).

3.2 PSO algorithm proposed for BSRP

At the start the parameters of the problem get initialized from a file or the code, a Swarm with an M number of particles gets created. Every particle has a current candidate solution that is a random permutation of the station vector generated doing a Fisher–Yates shuffle for every particle. The Swarm has also a function to improve the solution according to the values of velocity after each iteration.

For every particle then we search in the solution space, calculating the fitness function and keeping the best value obtained in every particle. This gets repeated N times and for every iteration, we save the best values among all the particles.
This explanation can be completed with the following pseudocode:

Algorithm 2: Modified PSO for BSRP

```
Initialize Swarm, Particles and problem parameters ;
foreach iteration do
    foreach particle in swarm do
        updateVelocity(particle);
        updateSolution(particle);
        p.fitnessValue ← computeFitnessValue(p);
        if p.fitnessValue < p.PBestValue then
            p.PBest ← p.currentSolution;
            p.PBestValue ← p.fitnessValue;
            p.PBestVelocity ← p.velocity;
        end
    end
    updateGlobalSolution();
end
```

3.3 Fitness Function

Since the fitness function plays a big role in the search for the best solution is important to show it and talk a bit about it.
As is possible to see from the following pseudocode a particle gets penalized if needs to come back to the base and refill (or empty) when there are other stations available to do so. This will reward particle that does smart moves, like for example visiting the depot for a refill if is near instead of going to the other part of the city. Besides, the function penalizes a trip done by a node from the depot if other stations can fulfill his request nearer. In this way, the best solution will be also the one that uses the depot for refilling only when it is strictly necessary.

Algorithm 3: Fitness function for BSRP

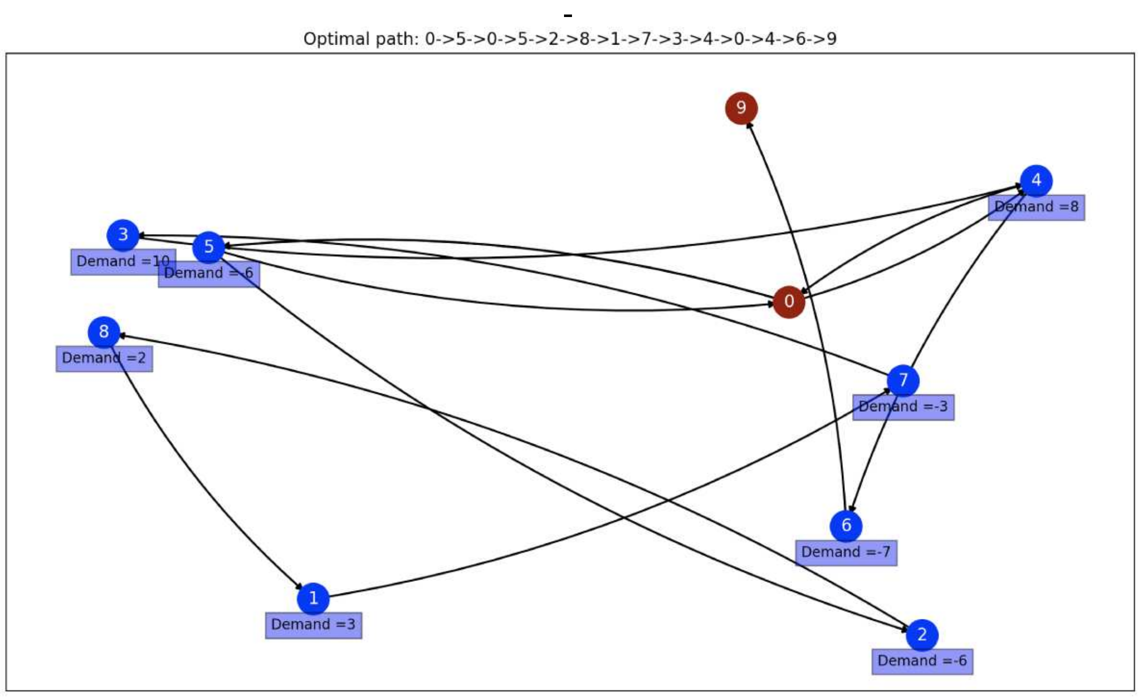
```
maxCapacity ← Sum of all vehicle capacities;
currentCapacity ← maxCapacity;
prevStation ← 0;
fitnessValue ← 0;
for i = 0, ..., nOfStation do
    currentStation ← stations[currentSolution[i]];
    if currentStation.demand = 0 then
        Go to next iteration;
    end
    if currentCapacity - currentStation demand < 0 then
        Search for a candidate pickup node not already visited;
        if No candidate has been found or depot is closer then
            Add the distance from the prevStation to the depot to the fitnessValue;
            prevStation ← 0;
            currentCapacity += maxCapacity/3;
            Repeat this iteration;
        else
            Swap currentStation with the candidate found;
            Update pickupStations hashmap;
            Repeat this iteration;
        end
    else if currentCapacity - currentStation demand > maxCapacity then
        Search for a candidate dropoff node not already visited;
        if No candidate has been found or depot is closer then
            Add the distance from the prevStation to the depot to the fitnessValue;
            prevStation ← 0;
            currentCapacity -= maxCapacity/3;
            Repeat this iteration;
        else
            Swap currentStation with the candidate found;
            Update dropoffStations hashmap;
            Repeat this iteration;
        end
    end
    currentCapacity ← currentCapacity - stationDemand;;
    Add to the fitness value the distance from the last station visited to the final
      depot;
    return fitnessValue
end
```

3.4 Implementation

The implementation has been done with C++ for the PSO, the BikeSharingModel and all other part of computation. The visualization part has been done moving the data elaborated from the c++ module to files and read with a python script in order to visualize the result. The BikeSharingModel can be created:

- In an hard coded manner, in order to do some test, here every position of the station is randomly generated and the distance between stations computed with euclidian distance. The visualization actually map the position of the station in the space
- Reading from file, in this way is possible to test the algorithm and compare the results obtained. Unfortunately the files used haven't the coordinate of every station and the distance matrix is asymmetrical so it is not possible to obtain back the coordinate of every station from it. This will result in a random placing in space during visualization.

FIGURE 1: Visualization of the optimal path found by the algorithm



Sourcecode: https://github.com/giacoballoccu/PSO-BikeSharingRebalancingProblem

4 Results

In order to see how the algorithm performs and if the tradeoff between time and optimality of solution can be worth it, I have analyzed some test files and compared them with the

values obtained solving the same problem with linear optimizers.
An important consideration is that all the tests have been done using a single vehicle as I have done in my previous work, using more vehicles the solution gets better but get very often stuck in local minima.
(In the upcoming table is possible to observe some of the results obtained. The rows represent every instance of the file used while the columns have these meanings:

- Particle: number of particles initialized
- Particle: number of time the algorithm tries to improve a single particle solution
- Real Minimum Cost: represents the cost of the optimal solution calculated with linear solvers.
- Cost: It represents the cost obtained by my algorithm
- Time: Execution time excluding the time to generate logs
- Gap: Difference between optimal solution and solution obtained

TABLE 1: Table1: Recapitulatory table of results

Filename	Particles	Iterations	Real minimum Cost	Cost	Time (s)	Gap
1Bari30.txt	1500	250	14600.0	16500	3.2	13.01
1Bari30.txt	100000	250	14600.0	15100	656.3	3.42
2Bari20.txt	1500	250	15700.0	17400	3.1	10.83
3Bari10.txt	1500	250	20600.0	27100	3.1	31.55
3Bari10.txt	100000	250	20600.0	25600	511.0	24.27
4ReggioEmilia30.txt	1500	250	16900.0	18400	3.2	8.8
5ReggioEmilia20.txt	1500	250	23200.0	27400.0	3.1	18.1
6ReggioEmilia10.txt	1500	250	32500.0	38300	3.4	17.85
7Bergamo30.txt	1500	250	12600.0	15200.0 x	3.3	20.63
8Bergamo20.txt	1500	250	12700.0	16500.0	3.6	29.92
8Bergamo20.txt	10000	1000	12700.0	15700.0	66.6	23.62
8Bergamo20.txt	100000	1000	12700.0	14900.0	737.5	17.32
9Bergamo10.txt	1500	250	13500.0	16400.0	3.6	21.48
10Parma30.txt	1500	250	29000.0	31700.0	3.2	9.31
11Parma20.txt	1500	250	29000.0	31700.0	3.2	9.31
12Parma10.txt	1500	250	32500.0	36400.0	3.4	12.0
16LaSpezia30.txt	2000	500	20746.0	30787	3.8	48.4
16LaSpezia30.txt	100000	1000	20746.0	27398	650.74	32.0
17LaSpezia20.txt	2000	500	20746.0	31115.0	3.9	49.98
18LaSpezia10.txt	2000	500	22810.0	32132	4.0	40.87
21Ottawa30.txt	2000	500	16202.0	20876.0	3.9	28.85
22Ottawa20.txt	2000	500	16202.0	21629.0	3.9	33.5
23Ottawa10.txt	2000	500	17576.0	23324.0	4.1	32.7

From the following table and from many more tests done is possible to infer that:

- About the number of particles: Doing many tests with several particles is possible to deduce that a number of particles too small don't allow the swarm to navigate enough problem space leading to bad results. A good amount of particles must be defined accordingly with the number of stations since every particle will start with a random solution vector with the stations. Incrementing the particles will also increment the time and for the increment to get reasonable (and don't have all particles stuck in local minima), it needs to be in the order of thousands.
- About the number of iterations: The conversation is pretty much the same as particles with iterations after a number of iterations the particle will converge because of the velocity update and it's very difficult to come out from that minima. Incrementing the number of iterations to more than 500 yields an increment of elapsed time and very small improvements.
- The gaps get much higher when the number of stations increment, the nodes have a similar amount of pickup and dropoff, or the capacity of the vehicle is much smaller compared to the max demand of the nodes.

In general, the algorithm is very fast like other heuristic algorithms, the problem is that the gap can be sometimes a bit too high to accept the tradeoff or can be preferable to use another heuristic algorithm (e.g Ants Colony). Being PSO is a metaheuristic as it makes few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions.
Yet as it is possible to see from the table the tests done with a big number of particle has a much smaller gap, this show the potentiality of the algorithm. With a stronger optimization could be possible to obtain great results in a relatively small amount of time, this would be possible using some data structure to store the solution already explored avoid repeated computation (e.g Tabu Table). Another improvement that can be done would be to tweak parameters with more precision for the specific problem to explore with less randomness the space of solutions.

References

Particle swarm optimization Wikipedia[1] https://en.wikipedia.org/wiki/Particle_swarm_optimization.

Shi, Y.; Eberhart, R.C. (1998). "A modified particle swarm optimizer". Proceedings of IEEE International Conference on Evolutionary Computation. pp. 69–73. [2] https://doi.org/10.1109%2FICEC.1998.699146.

Kennedy, J. (1997). "The particle swarm: social adaptation of knowledge". Proceedings of IEEE International Conference on Evolutionary Computation. pp. 303–308. [3]https://doi.org/10.1109%2FICEC.1997.592326.