Faculty of Electrical Engineering and Computing
Graduate study

Heuristic Optimization Methods
Academic year 2021/2022

Students: ***Ivan Navratil (0036510118), Luka Mesarić (0036505985)***

1. Details and results

Programming language

Kotlin

Best found results

	instance 1			instance 2		
time limit	1m	5m	un	1m	5m	un
number of vehicles	10	10	10	18	18	18
distance	990.73	980.17	980.17	4613.15	4233.05	4233.05
number of iterations	51921	36082	36082	32540	160080	160080

	instance 3			instance 4		
time limit	1m	5m	un	1m	5m	un
number of vehicles	36	36	36	20	19	19
distance	9274.26	9071.72	8997.76	8871.40	9263.45	8878.73
number of iterations	15139	34781	48082	3294	11172	13507

	instance 5			instance 6		
time limit	1m	5m	un	1m	5m	un
number of vehicles	75	75	74	19	19	19
distance	27185.63	25870.99	26850.31	42627.00	36380.22	35656.83
number of iterations	2277	8878	10634	1872	7180	16355

2. Description of the problem

In the capacitated vehicle routing problem with time-windows (CVRPTW), a fleet of delivery vehicles with uniform capacity must service customers for a single commodity with known demand, opening and closing hours, and service time. The vehicles start and end their routes at a common depot. Each customer can only be served by one vehicle. The objectives are to minimize the fleet size and assign a sequence of customers to each truck of the fleet minimizing the total distance traveled such that all customers are served, and the total demand served by each truck does not exceed its capacity. It is assumed that trucks move at a constant speed and can directly go to any customer no matter their current location. Euclidean distance is used.

3. Description of the implemented algorithm

Solution representation

A single solution is represented by a SolutionBuilder class. It consists of a list of routes, each represented by a RouteBuilder class. A single route is essentially a list of NodeMeta objects, which hold information regarding the visited customer or depot, commonly referred to as a node, as well as the time at which the vehicle left that customer.
As an optimization, SolutionBuilder keeps track of all unvisited customers, and RouteBuilder keeps track of total distance and remaining capacity for this route.

Objective / fitness function

In CVRPTW we have two objectives. The primary objective is to minimize the number of vehicles, while the secondary is to minimize the distance covered by all vehicles. These two do not necessarily go hand in hand. It can often happen that by using more vehicles, we can focus them on specific groups and eliminate longer travel time between distant customers from different regions, thus reducing the total distance.
For this reason, an explicit fitness function was not used. Instead, solutions were compared directly as pairs (vehicles, distance). If one solution used a smaller number of vehicles, the distance became irrelevant. Calculating a fitness delta (e.g., for simulated annealing) for this representation is not easy, but it was not needed so that issue was ignored.
A short discussion on infeasible solutions can be found near the end of the report.

High-level overview

The final solution is constructed in two distinct phases. In the first phase, Ant colony system (ACS) is used, a variant of Ant colony optimization (ACO). Afterwards, the solution is improved using local search (LS).

The first phase starts with pheromone initialization, i.e., setting all matrix elements to τ_0. In each iteration, the pheromone matrix is copied. A certain number of ants then sequentially traverse all customers, with each ant generating a complete and valid solution. Because ACS is used, they all update pheromones in the local (copied) matrix, such that the next ant can use the information gathered by all ants before it. At the end of the iteration, the best solution found by all ants is selected. A very quick local search is performed on it, which is guaranteed not to make the solution worse. The original pheromones are then updated according to the result of LS and the incumbent solution is updated if applicable.

The local search utilizes various neighborhoods. However, it is not a classical variable neighborhood descent. Four neighborhoods are as follows (assuming constraints are not violated):

- 2-opt* – Delete an arc from two different routes, thus splitting routes into two parts. Exchange the second parts of those routes.
- Internal-2 – Reorder two neighboring customers in one route, i.e., X-1-2-Y becomes X-2-1-Y.
- Internal-3 – Reorder three neighboring customers in one route, i.e., X-1-2-3-Y becomes X-3-2-1-Y.
- Transfer – Take away a customer from one route and transfer it into another route.

These neighborhoods were combined into five functions in different ways, two of which are deterministic (e.g., greedily choose the best move from all neighborhoods), and three are stochastic:

- randomly choose a neighborhood and take the best move,
- randomly choose any move from all neighborhoods,
- randomly choose a move from the top 20% of moves from all neighborhoods.

For quick local search, only a single deterministic function was used. For full search, the two deterministic functions we run first, followed by repeatedly randomly choosing and executing one of the stochastic functions until time limit is exceeded.

One neighborhood that could have been implemented, but was not, is a swap of two customers from different routes, similarly to two consecutive transfers. However, those transfers might break constraints, whereas an atomic swap would not.
Another option would be to create a generalized Internal-N neighborhood, which would swap a customer with its “neighbor” which is N arcs away. It was deemed that such valid swaps would not occur too often.

Moves were ordered based on how much they would improve the solution if they were applied. The primary criterion is how much would the relative difference of route lengths increase. The secondary criterion was how much would total distance be reduced. Only moves which improved distance and were valid were considered.
The idea behind the primary criterion is to prefer moves that shorten routes and eventually result in completely removing them. A prime example of that in action is when LS was applied on an ACS solution with 79 routes and resulted in a solution with only 75 routes.
A route could be eliminated by 2-opt* if first and last arcs were removed, effectively chaining two routes into one, or if a *Transfer* removed a node from a route with a single customer. The algorithm looks if such moves exist at the start of the local search and applies them if possible. Otherwise, moves are selected as described previously. Quick search is primarily used for these reasons, to facilitate shortening routes and eventually removing them, thus speeding up ACS significantly.

Ant traversal

An ant starts by constructing an empty solution with a list of routes that has a single empty route starting at the depot. In each iteration, it considers all unvisited customers, relative to its current location, which is equal to the last node in the last route of the current partial solution. All customers for which any constraint is not met are filtered out. An optimization step was to ignore customers from which a vehicle could not return to the depot before the end time. With that additional constraint, it was guaranteed that the solution could be built without backtracking. If no valid neighbors exist, the route is ended at the depot and a new one is created if needed. A complete solution is constructed when there are no unvisited nodes. How exactly is the next customer chosen from the filtered list will be described later.

Iteration size and termination criterion

Both elapsed real time and number of iterations are used as termination criteria. If either is violated, the search which is currently active is terminated. Separate limits are implemented for entire ACS, for quick LS within ACS, and for final LS. In most scenarios, iterations are set too high because we want the search to run for a specific amount of time.
In case of full LS, time limit is used for entire LS, whereas iterations are used within each function.

Iteration size and runtime vary widely from instance to instance. For example, for instance 1 final local search can sometimes do only a few iterations before finding an incumbent solution when starting with a good ACS solution, whereas for instance 6 it will usually do over 600 iterations. There are usually several hundred valid LS moves that can be considered in each iteration (for larger instances). For instance 6, on average it took 20ms to perform one iteration of LS which considers around one million possible neighbors.

Heuristic specific design elements

Heuristic of visibility is the inverse of the distance between two nodes.
Wait time heuristic is the inverse of the time a vehicle will spend waiting for the customer to open, or 1 if it does not have to wait.

Many parts of the overall heuristic are regulated by parameters:

- α – regulates the relative importance of pheromones
- β – regulates the relative importance of visibility
- θ – regulates the relative importance of wait time
- q_0 – probability of choosing the best neighbor
- τ_0 – initial pheromone value
- ρ – pheromone evaporation factor
- ants per iteration – number of ants to be used in each iteration of ACS

How these parameters impact solution quality is described in a later section.

As in most ACO implementations, next customer is chosen probabilistically.
We calculate the intermediate values as a product of pheromones, visibility and wait time heuristics, to the power of α, β and θ respectively. Those values are normalized by the sum of all values.

Parameter q_0 is the probability with which we directly choose the customer with largest previously calculated probability. Otherwise, we draw a random customer, considering the probability distribution.

Although τ_0 is provided as a parameter, it can also be estimated automatically. Usually, a good value is to set $\tau_0 = 1 / L^*$, where L^* is the total distance of a solution which could be considered decent. In practice, that is done by running a separate ACS and quick LS and taking the distance as L^*. When that is done, the pheromones can be reinitialized, and complete search restarted.

Pheromone updating is done according to the following equation: $\tau_{ij}^{new} = (1-\rho)\tau_{ij}^{old} + \frac{\rho}{L^*}$
It is the same both for local and global updating, with the only exception being which pheromone matrix is getting updated.

4. Pseudocode

Please note that the entire algorithm had over 1500 lines of code with various optimizations to improve performance, such as custom flat matrices. It was quite difficult to convert it into simple pseudocode. Below are shown key snippets of simplified code with comments.

```
fun main() {
    //creates ACO for a given instance with a given configuration
    val aco = AntColony(instance, config)

    //starts ACO
    val incumbentSolutionAco = aco.run()

    //prepares local search for a given instance with a given best solution
    val localSearch = LocalSearch(instance, incumbentSolutionAco)

    //start local search on the best solution found by ACO
    val incumbentSolutionLs = localSearch.fullSearch(config.finalLocalSearch)
    //incumbentSolutionLs is the final solution
}


fun AntColony.run(): SolutionBuilder {
    //criteria for ACO termination
    val compositeTermination = CompositeTermination(
        TotalTimeTermination(config.antColony.runtime),
        TotalIterationsTermination(config.antColony.iterations)
    )

    //estimate tau zero if necessary
    if (config.antColony.estimateTauZero)
        config.antColony.tauZero = calculateTauZero(config.antColony.estimateLocalSearch)

    //initialize pheromones
    pheromones = FlatSquareMatrix(instance.nodes.size, init=config.antColony.tauZero)

    //run ACO until one of the criteria for termination is not met
    while (!compositeTermination.terminate(iteration))
        performSingleIteration(config.ant)

    return incumbentSolution
}


fun AntColony.performSingleIteration(antConfig: Config.Ant) {
    val pheromonesLocal = pheromones.copy()
    val solutions = ArrayList<SolutionBuilder>()

    //run a predefined number of ants
    repeat(antConfig.count) {
        //ants use pheromones from previous ants
        val ant = Ant(instance, pheromonesLocal, antConfig)
        ant.traverse()?.let { solution ->
            solutions.add(solution)
            evaporatePheromones(pheromonesLocal, antConfig.rho)
            updatePheromones(pheromonesLocal, solution, antConfig.rho)
        }
    }

    //run a quick local search on the best ant in this iteration
    val ls = LocalSearch(instance, solutions.minOf { it })
    val bestAnt = ls.quickSearch(config.antColony.bestAntLocalSearch)

    //save the best solution
    if (incumbentSolution == null || bestAnt < incumbentSolution)
        incumbentSolution = bestAnt

    evaporatePheromones(pheromones, antConfig.rho)
    updatePheromones(pheromones, bestAnt, antConfig.rho)
}
```

```
//runs a single ant which constructs a complete solution
fun Ant.traverse(): SolutionBuilder {
    val solutionBuilder = SolutionBuilder(instance)
    while (true) {
        if (neighbors.isEmpty()) {
            end route at depot and exit if instance is finished
        } else {
            solutionBuilder.addNextNode(pickNextCustomer(lastPosition, neighbors))
        }
    }
}


fun Ant.pickNextCustomer(sourceMeta: NodeMeta, neighbors: List<Node>): Node {
    val numerators = DoubleArray(neighbors.size) {
        calculateNumerator(sourceMeta, neighbors[i])
    }

    val chosenIndex = if (random.nextDouble() <= antConfig.q0) {
        numerators.argmax()
    } else {
        WeightedLottery(numerators).draw()
    }
    return neighbors[chosenIndex]
}


fun Ant.calculateNumerator(sourceMeta: NodeMeta, destination: Node): Double {
    val pheromones = pheromones[sourceMeta.id, destination.id].pow(antConfig.alpha)
    val visibility = 1.0 / (distances[sourceMeta.id, destination.id]).pow(antConfig.beta)
    val waitTime = waitHeuristic.calculateWaitTime(sourceMeta, destination).pow(antConfig.theta)
    return pheromones * visibility * waitTime
}


//runs a quick local search - used on tau zero estimation and at the end of each ACS iteration
fun LocalSearch.quickSearch(config: Config.LocalSearch): SolutionBuilder {
    val compositeTermination = CompositeTermination(
        TotalTimeTermination(startTime + config.runtime),
        TotalIterationsTermination(config.iterations)
    )
    searchInternal(compositeTermination, ::chooseBestSwap1)
    return incumbentSolution
}


//runs a full local search - used on final ACO solution
fun LocalSearch.fullSearch(config: Config.LocalSearch): SolutionBuilder {
    startTime = Instant.now()
    val timeTermination = TotalTimeTermination(startTime + config.runtime)
    val compositeTermination = CompositeTermination(
        timeTermination, TotalIterationsTermination(config.iterations)
    )

    searchInternal(compositeTermination, ::chooseBestSwap1)
    searchInternal(compositeTermination, ::chooseBestSwap2)

    //randomly choose stochastic search function 3, 4 or 5 with probability 0.45, 0.4 and 0.15
    while (!timeTermination.terminate()) {
        searchInternal(compositeTermination, randomChoosers[randomChoosersLottery.draw()])
    }

    return incumbentSolution
}
```

5. Analysis of results and discussion

Below is a comparison of ACS results with and without final local search with a total time limit of 5 minutes. Note that these ACS scores are not necessarily the best for that instance but resulted in overall best solutions after local search was performed.

5m	ACS		ACS + LS	
score	number of vehicles	total distance	number of vehicles	total distance
Instance 1	10	982.99	10	980.17
Instance 2	18	4678.62	18	4329.59
Instance 3	36	9925.32	36	9146.00
Instance 4	20	13972.41	19	9263.45
Instance 5	79	33582.19	75	26016.84
Instance 6	19	54619.35	19	36380.22

The final local search significantly improved solutions generated by ACS. It made a substantial difference on instance 5 where it improved the solution by 4 vehicles and on instances 4 and 6 where it improved distance by about 33%. In unbounded case, LS even managed to remove 5 vehicles in instance 5.

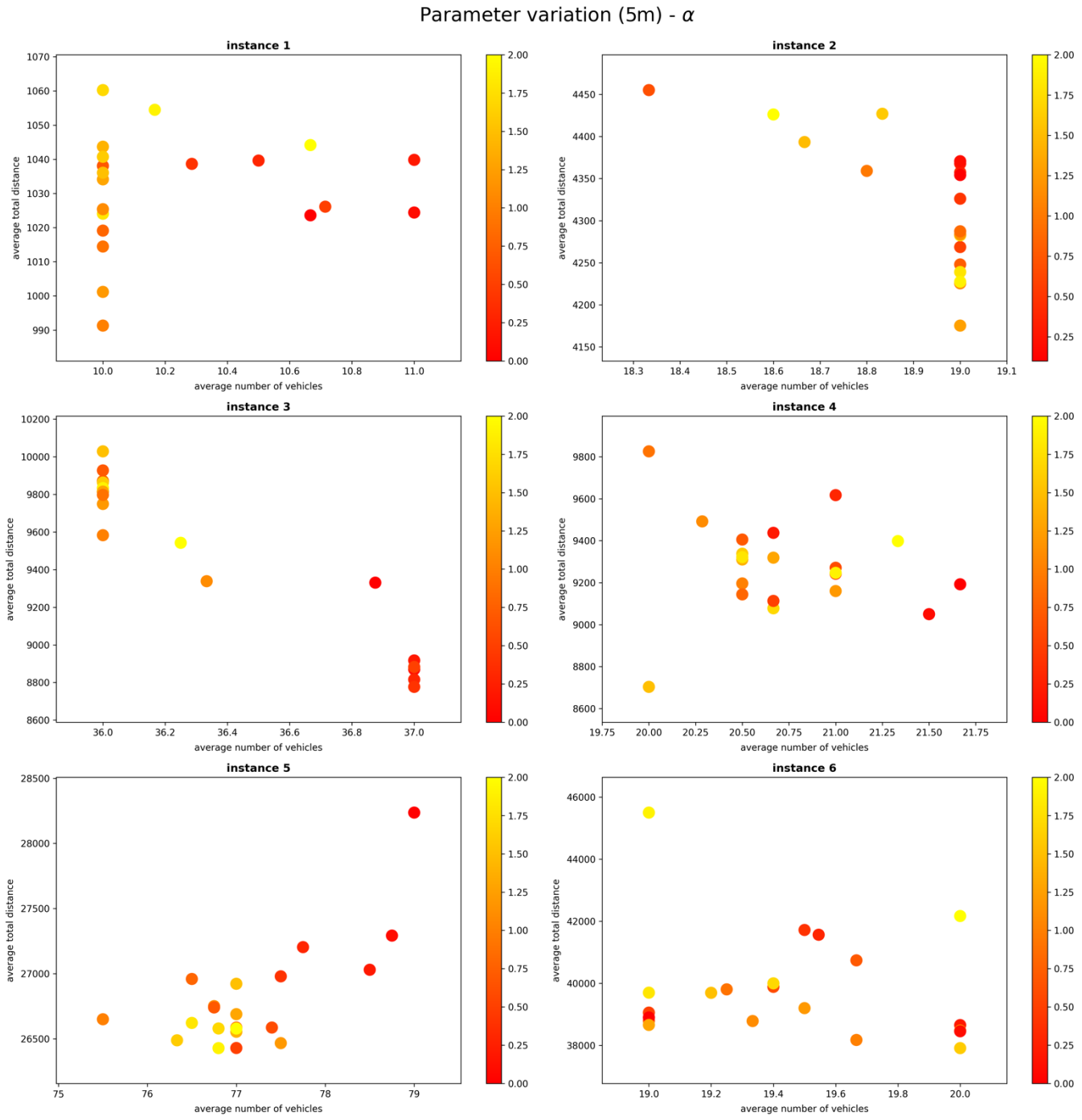
For smaller instances, the algorithm almost always finds the best solution it can within 10 minutes, with around 15% of the time used for final LS. For larger instances, up to 15 minutes is needed, and about 30% of it should be spent on final LS. These durations were estimated by running the algorithm on a local machine with a 12-core 3900X and 32GB RAM. Instances were always run in parallel to save time, but they were otherwise completely independent. In total, we estimated that the complete algorithm was run between 7000 and 10000 times.

Parameter analysis

For this analysis, we found good initial parameters for every instance. Then, we ran an analysis by randomly varying only one parameter of the heuristic. Results for each instance and for each parameter can be found below. In case of multiple runs with the same parameters, an average value of vehicles and distance is used. The data was gathered from 3000 runs, each lasting five minutes.

Parameter alpha

Very low values of parameter α resulted in bad solutions, with the only exception being instance 6. Depending on the instance, the optimal parameter value seems to be in the $[0.9, 1.3]$ interval.

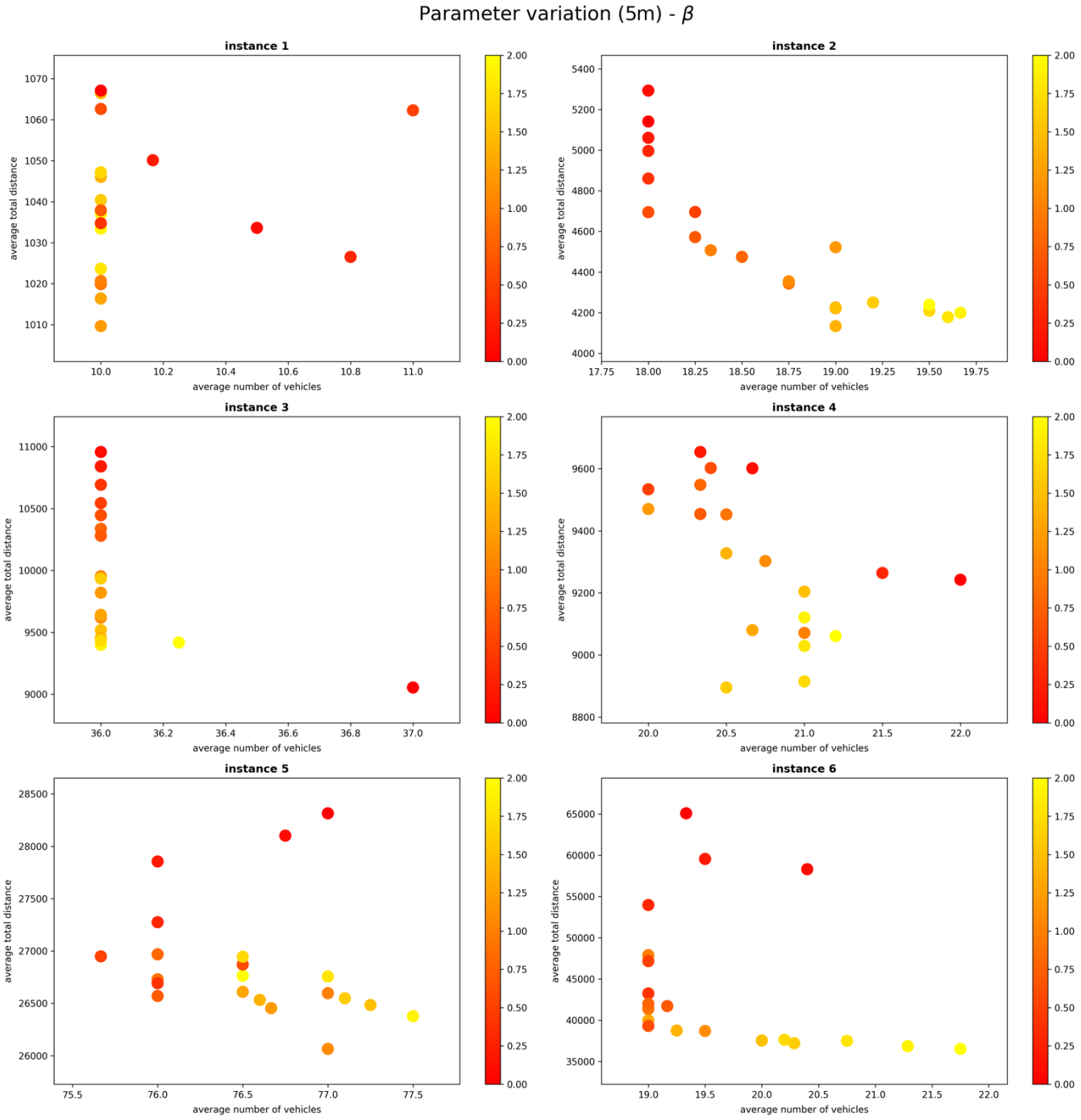


Parameter beta

In instances 1 and 3, the parameter β did not have much of an impact on the solution, barring very low values.

In instances 2, 5 and 6 higher values of parameter β produced a worse solution. The optimal parameter value for these instances seems to be in the $[0.25, 0.75]$ interval.

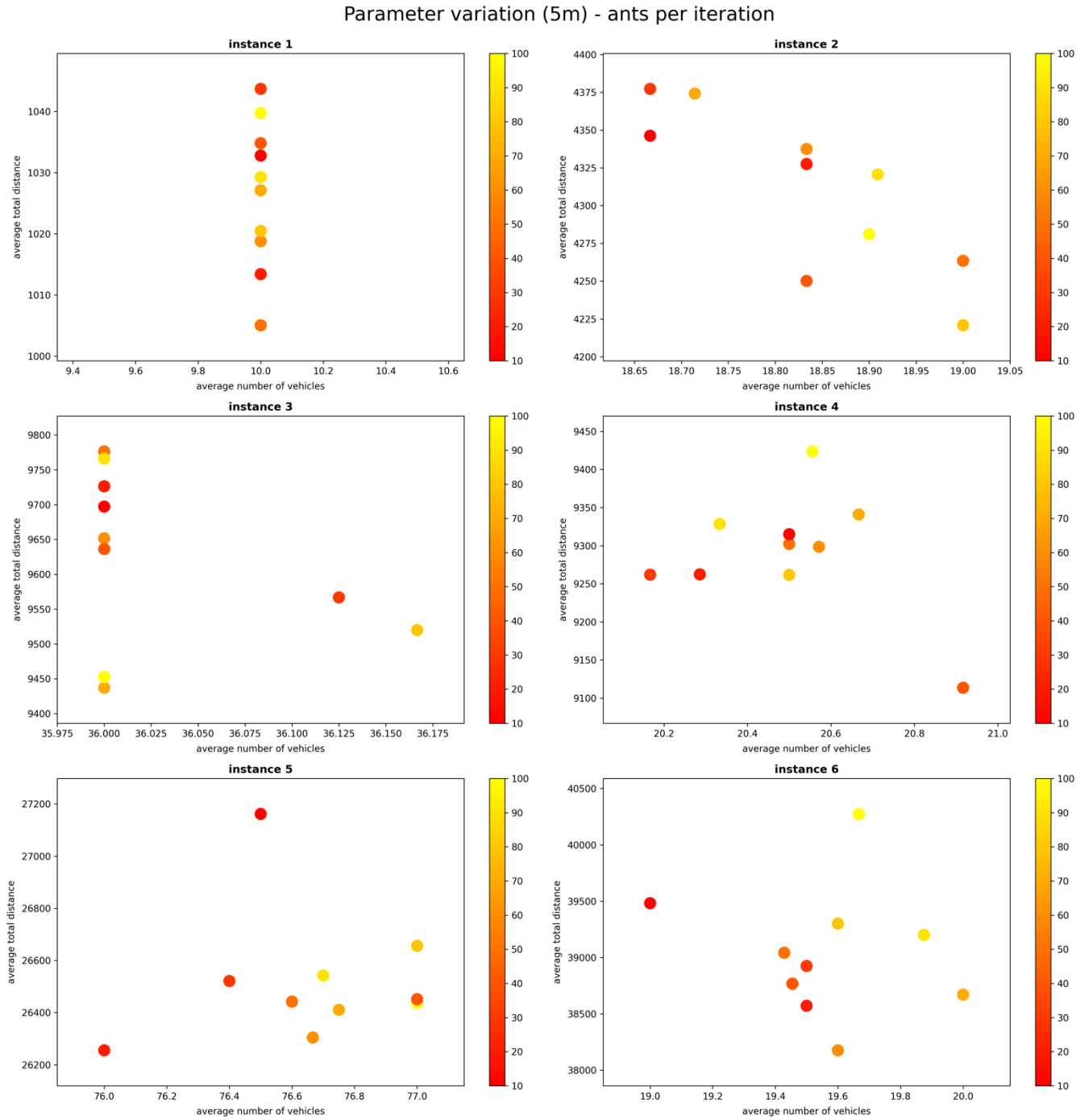
In instance 4, the optimal value seems to be in the $[0.75, 1.25]$ interval.



Ant count parameter

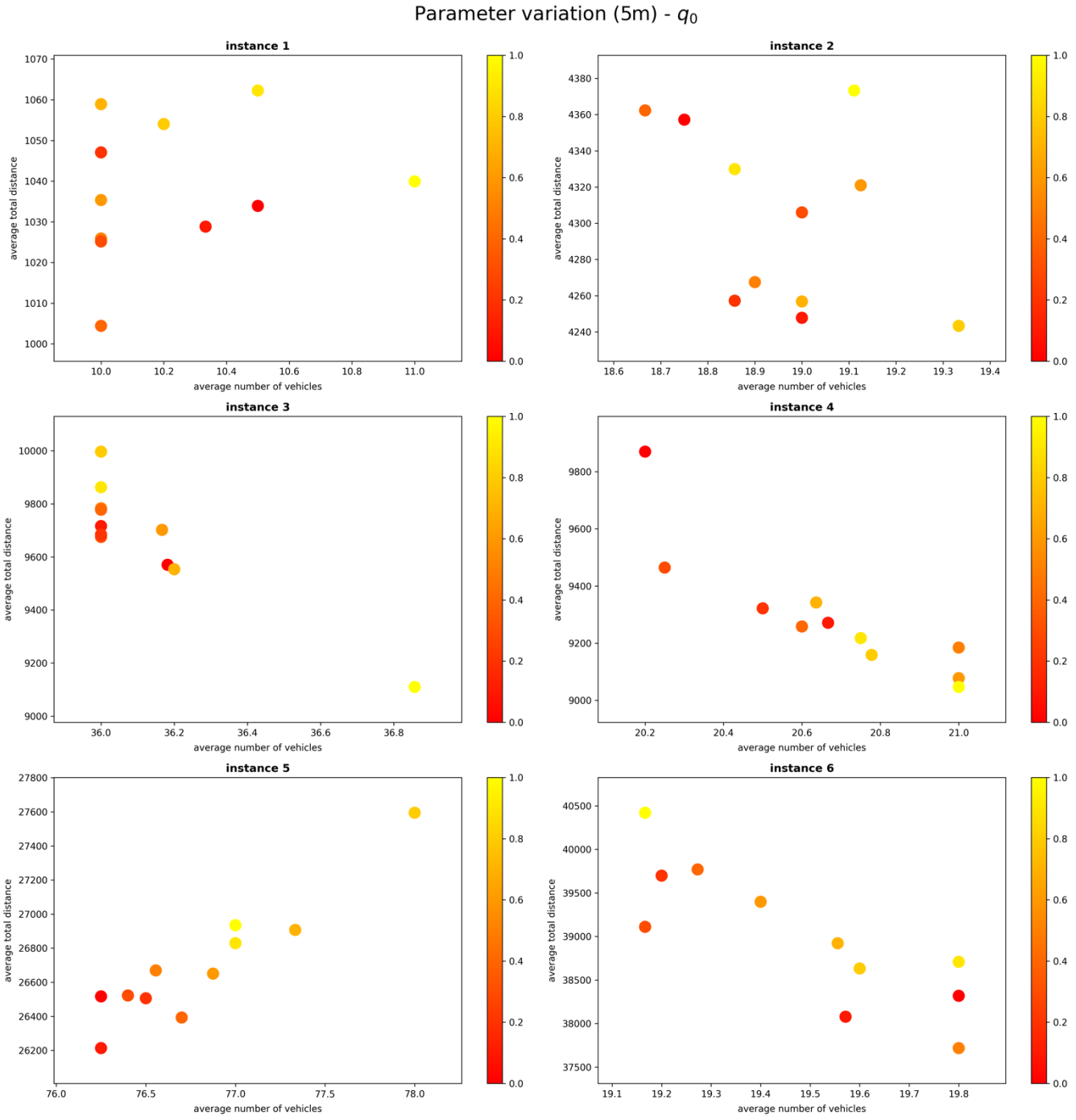
In instances 1 and 3, the number of ants did not have much of an impact on the solution quality.

In instances 2, 4, 5 and 6 lower number of ants produced better solutions. Since they when run on a five-minute time limit, the algorithm benefited from extra iterations that were a direct results of lower ant count, i.e., lower computational demand per iteration.



Parameter q_0

In all instances, lower parameter q_0 producer better, or equally good, solutions. The optimal value seems to be in the $[0.1, 0.4]$ interval.

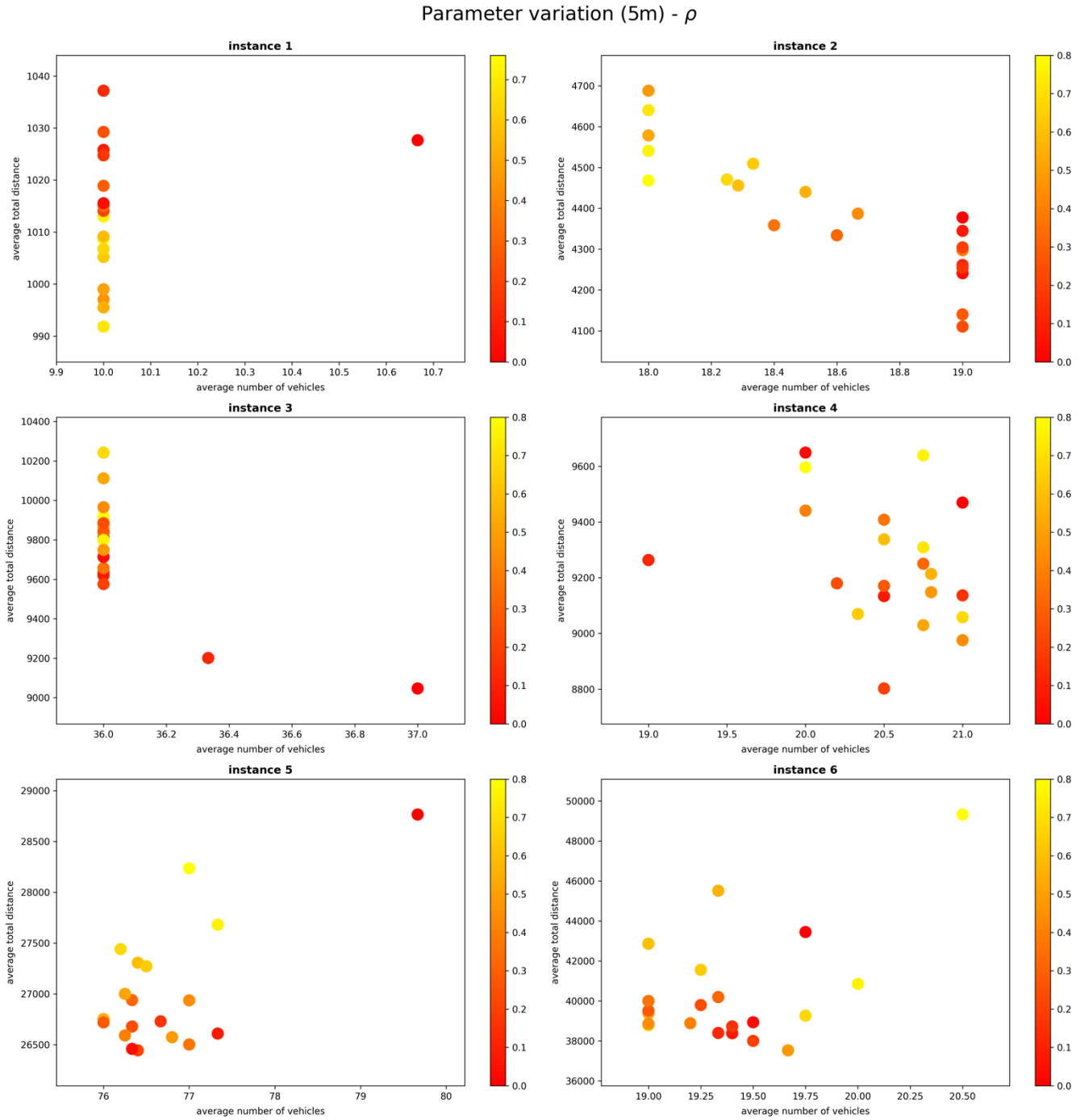


Parameter rho

In instances 1 and 3, the parameter ρ did not have much of an impact on the solution quality, barring very low values.

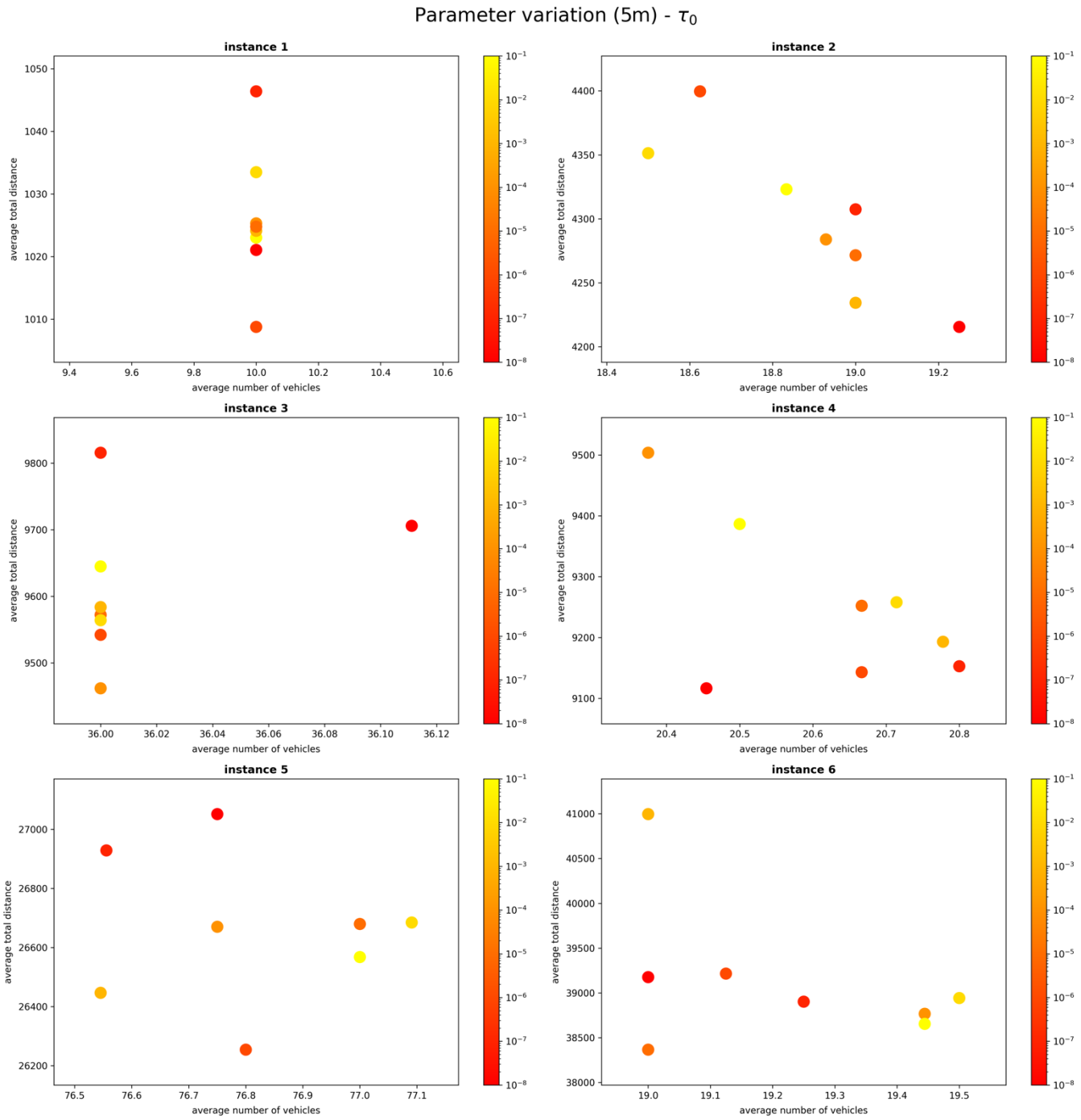
In instance 2, higher values of parameter ρ produced a better solution. The optimal parameter value for this instance seems to be in the $[0.5, 0.8]$ interval.

For other instances, there is not a clear trend that shows optimal values.



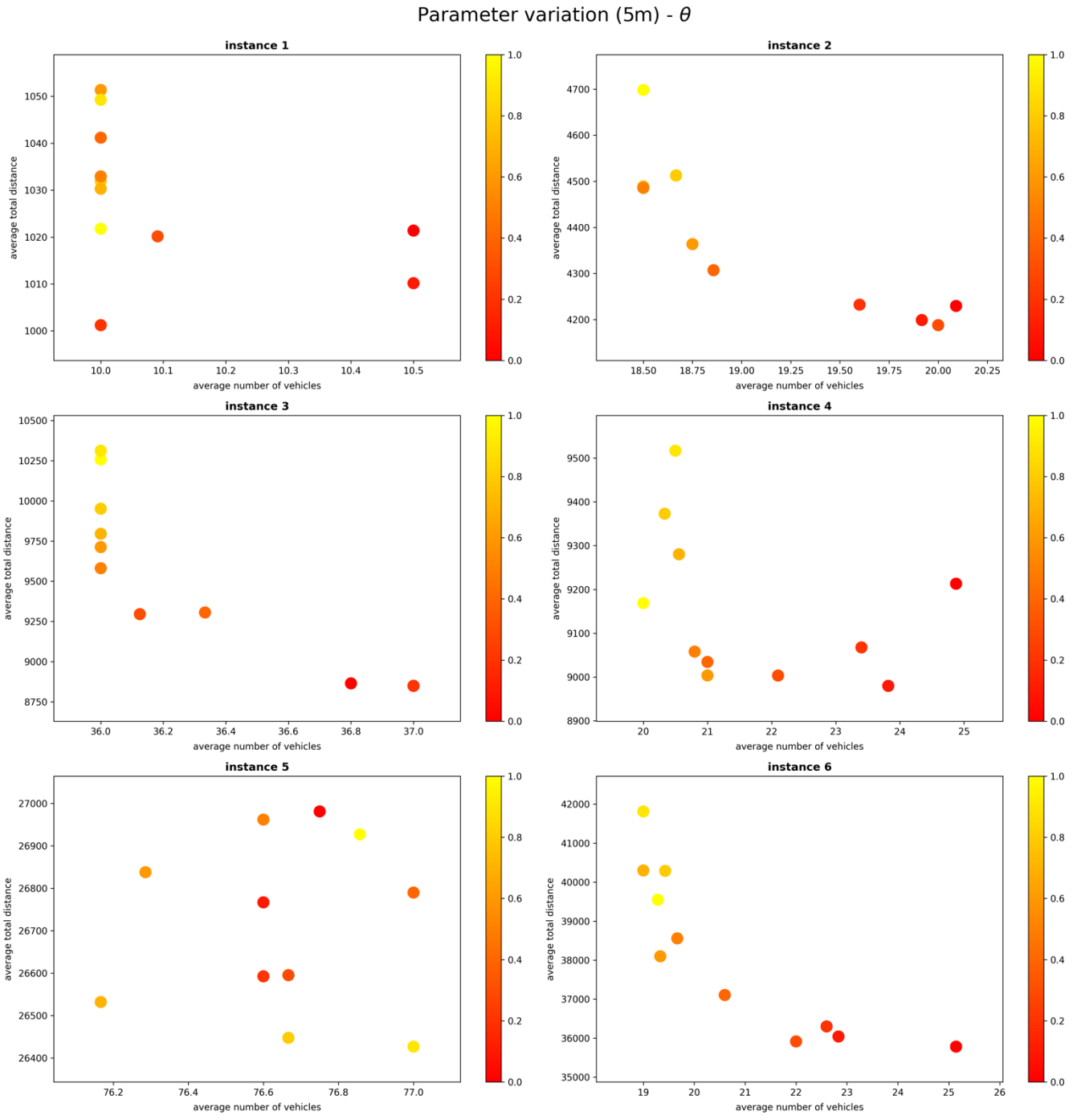
Parameter tau

Solution quality differs widely. There were many cases where exceptional solutions were found with completely unexpected values of τ_0. In general, 1/L* seems to be an educated guess considering how pheromone updating is performed, but even a tenth or ten times that value will suffice.



Parameter theta

In all instances lower values of parameter θ produced a worse solution (with only exception being the best instance 1 solution). The optimal parameter value for all instances seems to be in the $[0.5, 0.8]$ interval.



6. Conclusion

It seems like the algorithm performs rather well. This conclusion is based on the consistency with which it finds solutions for several of the given instances. We calculated the lower bound on the number of vehicles by summing up the demand of all customers and dividing it by the capacity of a vehicle and concluded that we reached that bound for instances 2, 3 and 6. We also researched that the best solution for instance 1 has 9 vehicles (Solomon, R112.100), however we could not seem to find it with our algorithm. We have also attempted to use a closed-source production grade solver for supply chain optimization called "LocalSolver" to see what good solutions look like. After improving our algorithms, we managed to beat it on every instance, sometimes even by several vehicles.

The algorithm could certainly be further improved. Both local search and choosing the best ant from one iteration could be updated to use simulated annealing, possibly even allowing multiple ants to update pheromones in a single iteration. Local search might even use a tabu list, considering moves are already used to reduce copying of entire solutions.

The search could be updated to accept infeasible solution. Currently, the only way for the algorithm not to find a solution is if it could not initially find anything with the specified number of vehicles. Instead, we could ignore that constraint and check it at the end of the entire search. We had no problems with that, so it was not changed from the current implementation. If needed, a simple workaround would be to increase the number in the instance file.

A *savings heuristic* could be used instead of, or in conjunction with, the *visibility heuristic*. We attempted to use it, both the basic version and an improved version proposed by Paessens in 1988 in his paper *The savings algorithm for the vehicle routing problem*, but we did not see any improvements. On the contrary, we could not achieve the best results we had before.

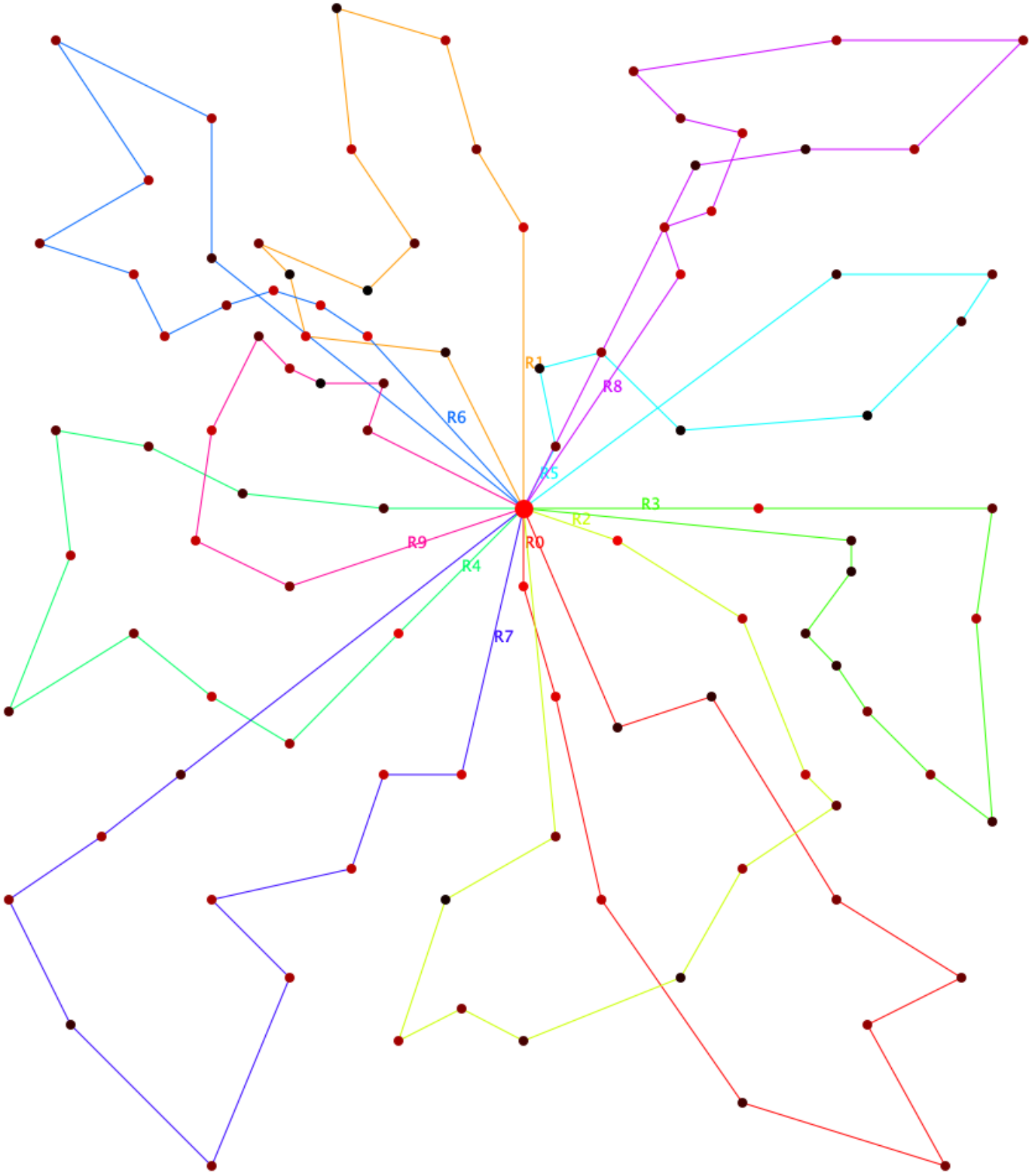
We also started by using a basic Ant system (AS), but later switched to ACS because it gave us more consistent results. Although AS could be parallelized, it was not needed because the same effect could be achieved by simply running multiple instances at the same time.

Overall, we are satisfied with the performance of the implemented algorithms.

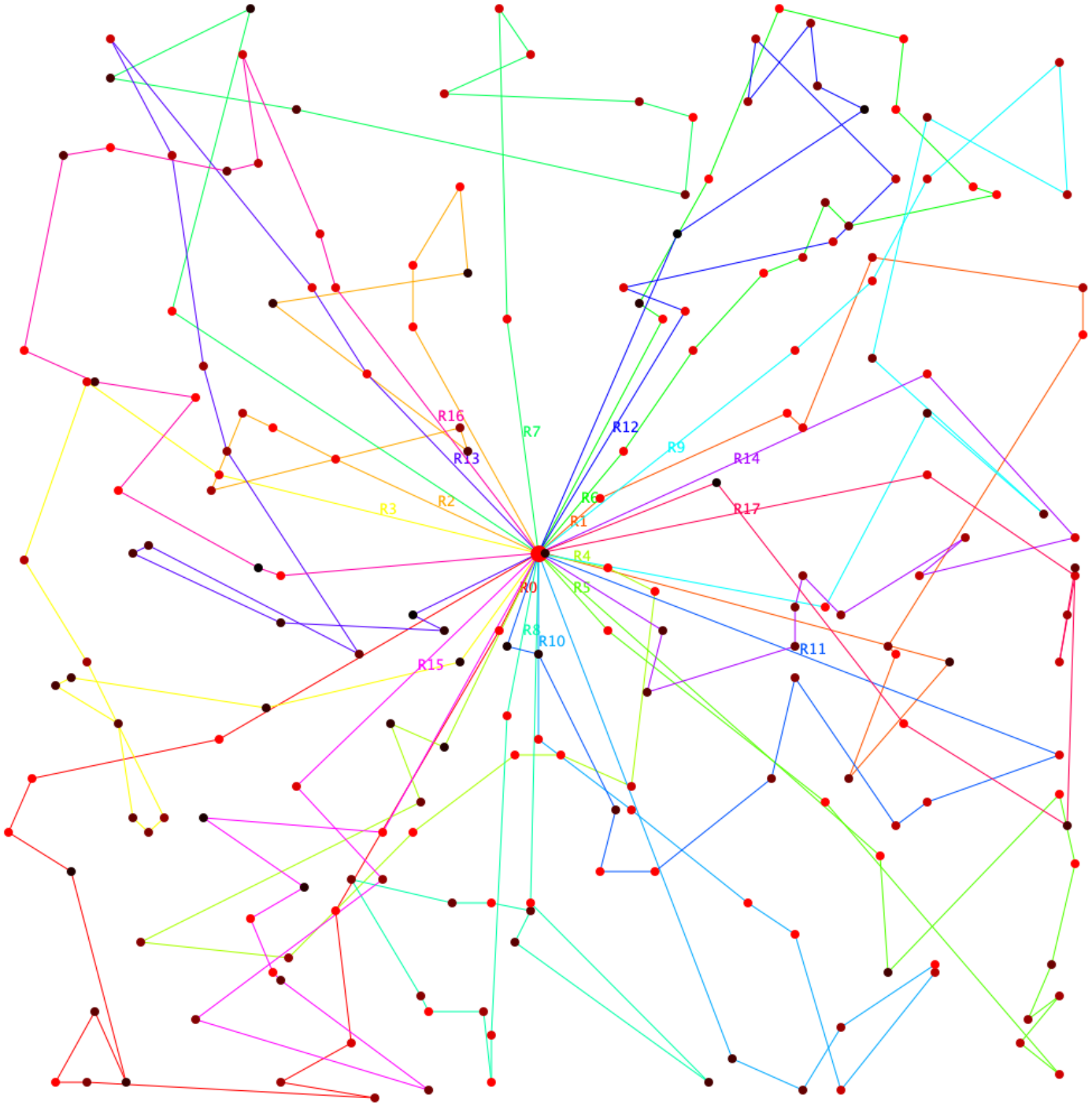
7. Plots of best solutions for each instance

Lighter color of the node represents earlier opening time. On the first edge of each route there is a node marking the route.

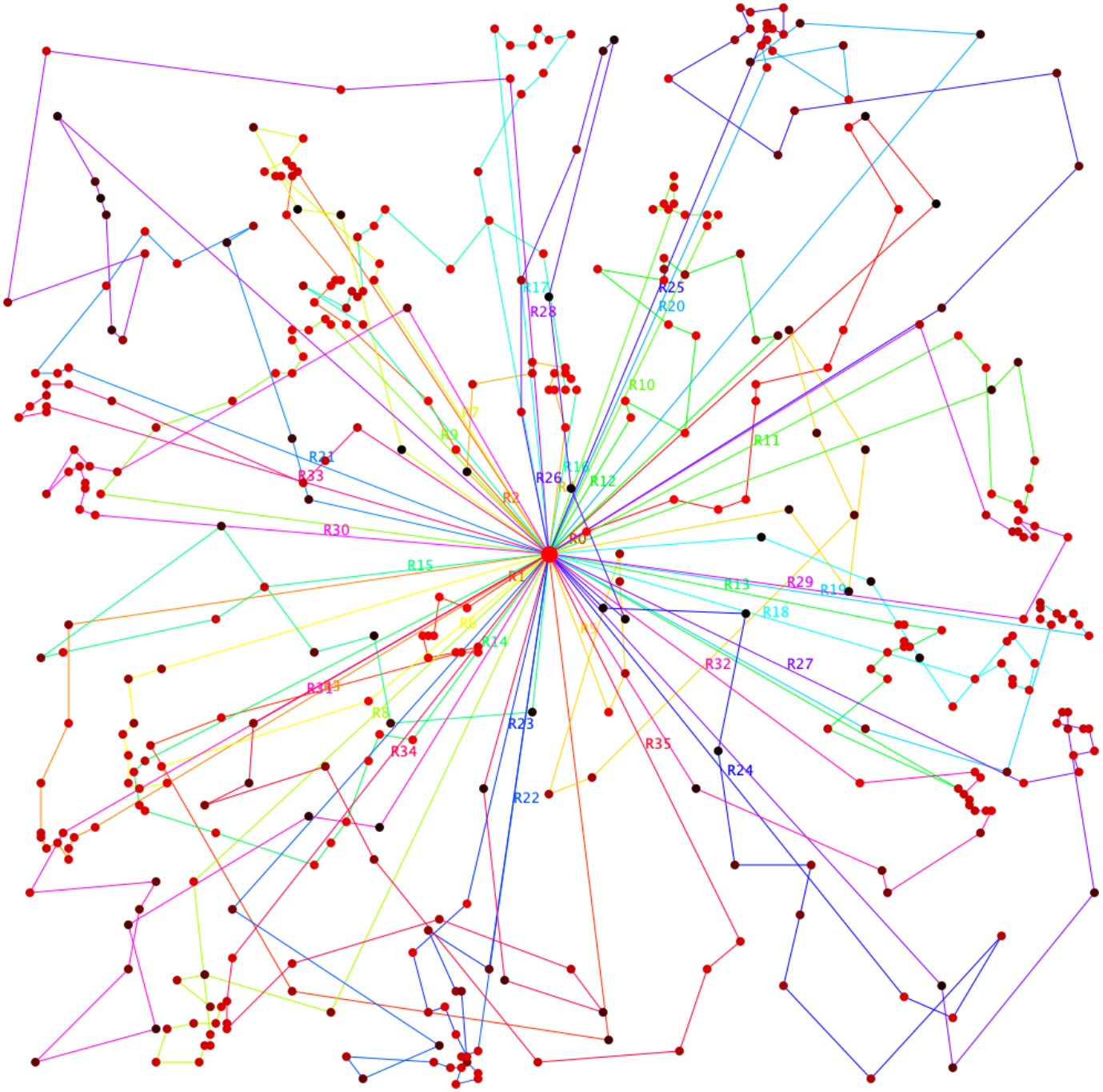
Plot of the instance 1



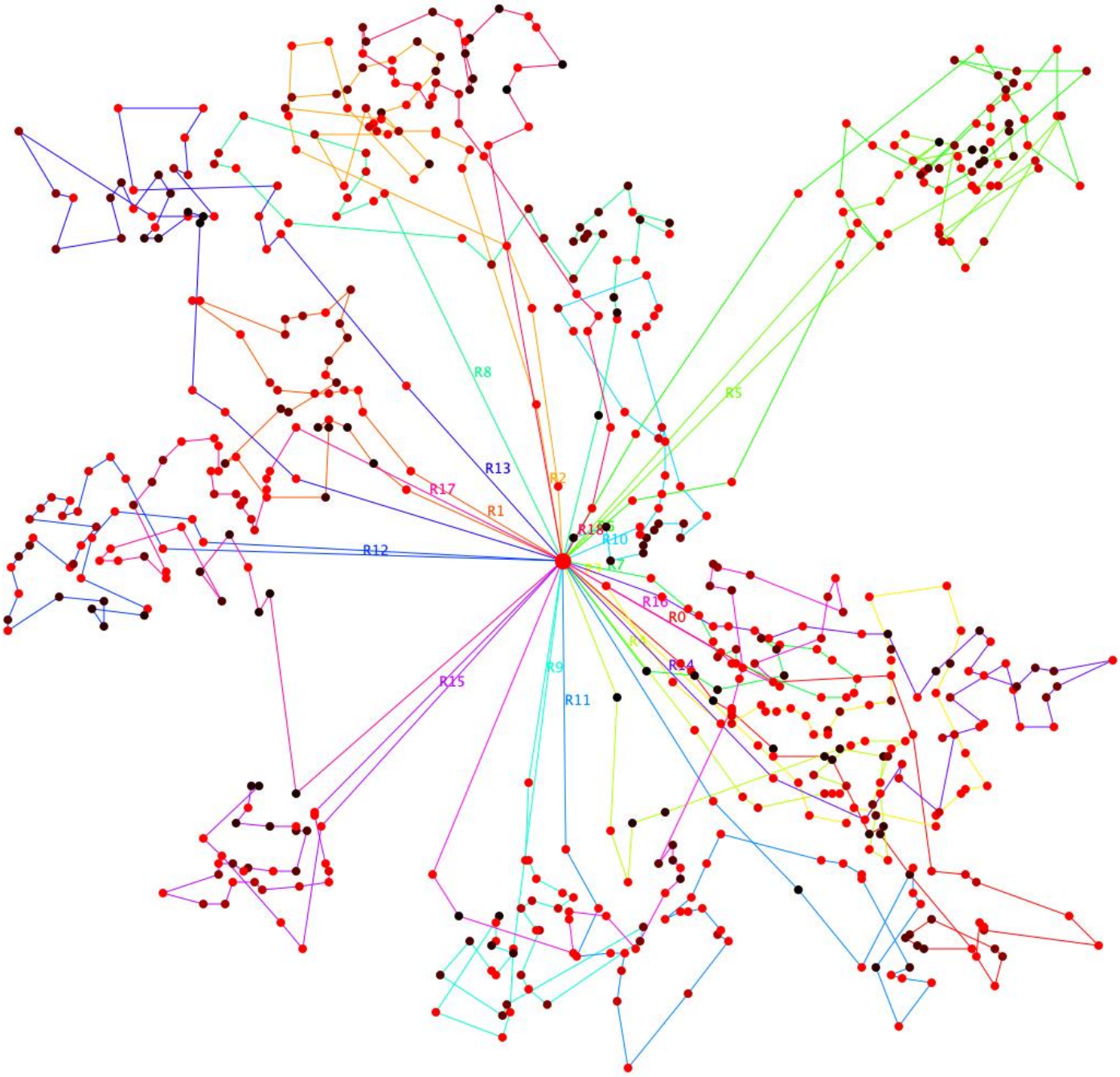
Plot of the instance 2



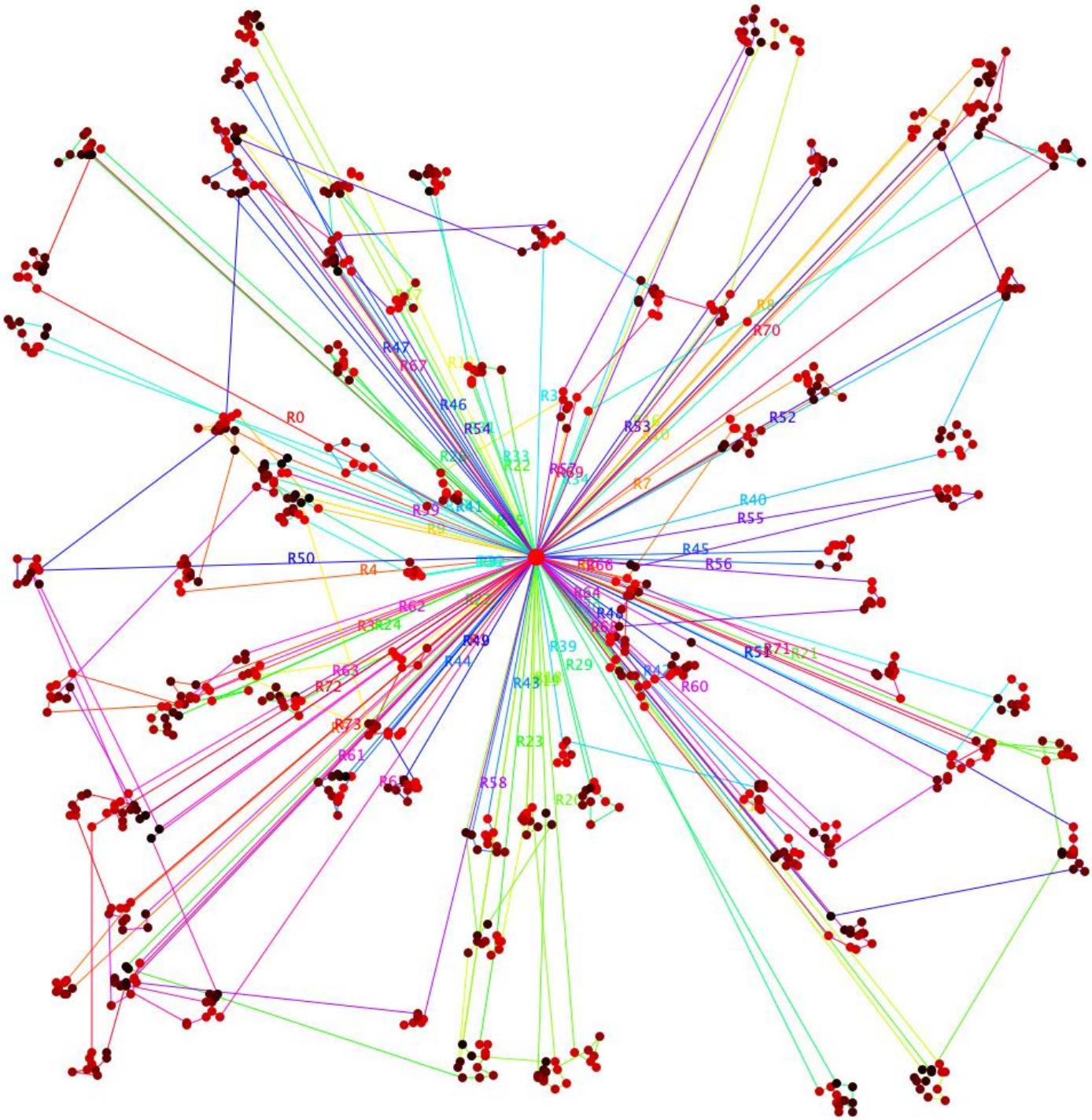
Plot of the instance 3



Plot of the instance 4



Plot of the instance 5



Plot of the instance 6