Report swarm intelligence

Pieter Van Keymeulen

June 6, 2017

1 Solving closest string with ACO

In this experiment we are going to try to solve the closest string problem [Lanctot et al., 2003] using ant colony optimisation. We are going to do this using two different main approaches. In both of them, we have a pheromone matrix which represents the world the ants live in based on the approach of Faro and Pappalardo [Faro and Pappalardo, 2010]. The ants can go walk this pheromone matrix and each step they make in this pheromone matrix represents a choice in the construction of a solution for our problem. The choices they make along the way are dependent on the amount of pheromone present on the path. The more pheromone present, the more chance there is that the ants will follow that path. While they follow a path, the ants also can deposit a tiny amount of pheromone and thus reinforces the path they follow. There are also multiple ants and together these ants form a colony. Because the ants constantly reinforce the path they follow by increasing the pheromone, the probability that other ants will follow this path as well increases.

The pheromone matrix is constructed as follows. On each position of a string representing a solution, one character of the alphabet can be chosen. Let there be a matrix P^{nm} with n being the amount of characters in the alphabet of characters for our string. That is the length of the set of characters our strings are composed of and m being the length of a string representing a solution. In the matrix P the columns represent the position in our solution string and rows represent the character chosen at that position.

Solutions are represented by a sequence of integers $t =< t_1, t_2, t_3 ... t_m >$ with t_i being an index representing the row in the in the pheromone matrix representing that character in the actual solution. So if row 1 in the pheromone represents the character a as a choice, then a 1 in this sequence represent an a. This is done so, internally, we can implement some procedure in a more efficient way.

Although there are only two main approaches, we implemented multiple mechanisms. Each approach will be a combination of a few of these mechanisms. These mechanisms are:

Ant system The regular ant system which is a literal interpretation of the approach described above

Elitist ant system A modification where only the ant who constructed the best solution gets to update the pheromone matrix.

Special solution construction A slight modification in the solution construction of the ants where there is a probability that an ant will copy a step from the current best solution of the entire colony

Local search Where we improve the current best solution of the entire colony by performing local search with that solution as a starting point.

Ant System will be our main approach and our second approach will be a combination of *Elitist Ant System* and *Special solution construction*. We will also try to combine both of these approaches with a local search and look if there is any difference between the version with local search and the version without it.

1.1 Ant system

This is just plain ant colony optimisation with no additions added based on the ACO algorithm as proposed by Faro and Pappalardo [Faro and Pappalardo, 2010]. It consists of the initialisation of the pheromone matrix, creating an ant colony and letting them construct a solution based on the pheromone matrix and update the pheromone matrix repeatedly until a termination condition is met. Each repetitions, the ants construct a solution and save their constructed solution. The pheromone matrix gets updated so that the other ants can create a solution based on the solution that the ant just created. The pheromone is updated in the following way.

$$P_{ij}^{(t+1)} = P_{ij}^{(t)} + \left(1 - \frac{HD}{m}\right) \tag{1}$$

Where m is the length of a string in our problem and HD is the maximum hamming distance of the solution the ant who updates the pheromone created and our set of strings S. After all the ants have constructed a solution, the best solution among those solutions is chosen. After all the ants constructed a solution the pheromone naturally evaporates according to a persistence rate ρ. During evaporation ρ is simply multiplied with each element of our pheromone matrix P. The lower this rate, the faster the pheromone will evaporate

When the ants have to make a decision, they do so random, with a probability for each possible choice. That probability being

$$\frac{P_{ij}}{\sum_{k=1}^{n} P_{kj}} \tag{2}$$

That is the probability that an ant will choose to put letter i on position j is the amount of pheromone normalised over the sum of all the pheromone for all possible characters on that particular position. Together all this creates the following algorithm

Algorithm 1 Basis for ACO

```
P ← initializePheromone
while ¬terminationCondition do
    for i = 1 doamountOfAnts
        ant ← A_i
        ant → constructSolution
        ant → updatePheromone
        if ant → currentSolution < colony → currentSolution then
            colony → currentSolution ← ant → currentSolution
        end if
    end for
    evaporatePheromone
end while
```

This will be our base algorithm. Our second approach will be a variation based on this one. Normally, also an heuristic value is used in an ACO algorithm [Dorigo and Stützle, 2004, p. 36], this is not the case here however.

1.2 Alternative algorithm

Inorder to further investigate the possibility to improve this algorithm, we implemented a variant on the algorithm proposed above. To do this we take an approach based on the algorithm proposed by Rajendran and Ziegler [Rajendran and Ziegler, 2005]. The idea is to also incorporate parts of the current known best solution in the solution constructed by the ant. With a probability of 0.4 the ant will choose to ignore the pheromone matrix and just take the character of the current know best solution for its current position. The original idea of Rajendran and Ziegler was to use this in the permutation flowshop problem but, as one can see, this approach also translates well to the closest string problem.

As explained earlier, we will combine this approach with an elitist ant system, were we do not iterate over each and and command them to update the pheromone while we update the pheromone, but rather we determine which ant currently constructed the best solution and only let that ant update the pheromone.

Algorithm 2 Elitist ACO

```
P ← initializePheromone
while ¬terminationCondition do
    for i = 1 doamountOfAnts
        ant ← A_i
        ant → constructSolution
        if ant → currentSolution < colony → currentSolution then
            colony → currentSolution ← ant → currentSolution
            bestAnt ← ant
        end if
    end for
    bestAnt → updatePheromone
    evaporatePheromone
end while
```

The pheromone initialisation, pheromone update rule and the pheromone evaporation rule remains the same.

1.3 Local search

As explained, we will also try a local search on one of our algorithms. That is, after an iteration is done and the current best solution of the entire colony is known, we will try to improve that solution using local search. We do this by defining a neighbour of a solution as the set of all solution which differ one in hamming distance from the current solution. That is, every neighbour differs in one character. We then try to search that neighbourhood for an improving solution. Because our neighbourhood definition generally generates huge neighbourhoods, we do this using first improvement, that is, the first improving neighbour is accepted. Thus we can search for our improvement while we are generate the neighbourhood. As a further addition we limit the neighbourhood size to the string length with a maximum of 200 elements. If we found an improvemnt, that improving solution will be our new current solution and the process continues.What we also do is that we only generate the neighbourhood by changing only the character from a certain index in the current solution up until the end of the current solution. This point starts point starts at 0 and increments with every step in the local search process.That until this counter reaches m, with m being the length of the strings.

Algorithm 3 Local search

```
π ← currentSolution
for i ← 0, m do
    N ← generateNeighbourhood(π, i)        ▷ Generate neighbourhood starting from i
    if ∄N then                              ▷ If the neighbourhood is empty
        breakLoop
    end if
    π ← findFirstImprovement(N)
end for
return π
```

2 Optimisation of the parameters

The usual ACO algorithm can be configured with a few parameters and this algorithm is no exception. As mentioned earlier, there is a certain persistence rate ρ, a certain number of ants and a termination condition expressed in the number of iterations the algorithm runs, that is the number of times all the ants are allowed to construct a solution and update the pheromone [1]. The values of these parameters fall under a few restrictions.

- iterations $\in \mathbb{N}$
- $(\rho \in \mathbb{R}) \wedge (\rho \geq 0 \wedge \rho \leq 1)$
- ants $\in \mathbb{N}$

The maximum number of evaluation allowed was 100. Within these constraints we can optimise the parameters of the algorithm for our specific problem. That is, we search for a set of ρ amount of ants and amount of iterations that gives us, in general, the best output. In order to do so we we perform an iterated race [López-Ibáñez et al., 2016] among set of different sets of parameters called Θ and use them to evaluate our test set. This will happen in iterations. Each iterations we will sample sets of parameters from Θ and perform a race with that sample. The sets of parameters that perform best are considered elite, the non-elite are thrown away and we continue into the next iteration but now now we edit our sample distribution so that it would be biased towards the elite configuration. This is done in case $|\Theta| = \infty$. This goes on until there is only one configuration remains. We have implemented this automatically for our algorithm using the R package *irace* [Lopez-Ibáñez and Dubois-Lacoste, 2017]

One of the nice features of the irace package is the fact that it logs the probability of a parameter being chosen in a candidate configuration throughout the iterated race. With this we can see which parameters are more sensitive. If the distribution is rather flat, that means that the parameter isn't that sensitive since it didn't care what the value of that parameter was. We plotted the parameter frequencies. In these plots, real stands for ρ. The parameters seem to be quite sensitive, except for the rho parameter in our main approach.

The final best parameters selected for our main approach were

```
# Best configurations (first number is the configuration ID) nonlocal nonspecial
   ants Iterations   rho
4     1        600 0.9726
3     7        200 0.5044
13    1        600 0.9502
```

The final best parameters selected for our second approach were

```
# Best configurations (first number is the configuration ID) special
   ants Iterations   rho
11   10        600 0.9699
1     9        600 0.9721
14    6        600 0.9656
```

The final best parameters selected for our second approach with local search were

```
# Best configurations (first number is the configuration ID) local
   ants Iterations   rho
12    8        800 0.9620
9    10        200 0.7455
4    10        200 0.8959
```

Each of these show the top three best configurations All the results in the next section are gathered using the parameters described as the best except for our main approach there we use the configuration with 7 ants, because an ACO algorithm with only one ant is not an ACO algorithm any more although the fact that only one ant scores best here is quite interesting.

[1] Or only construct a solution in case of the elitist version

Figure 1: Parameter frequency of our main approach

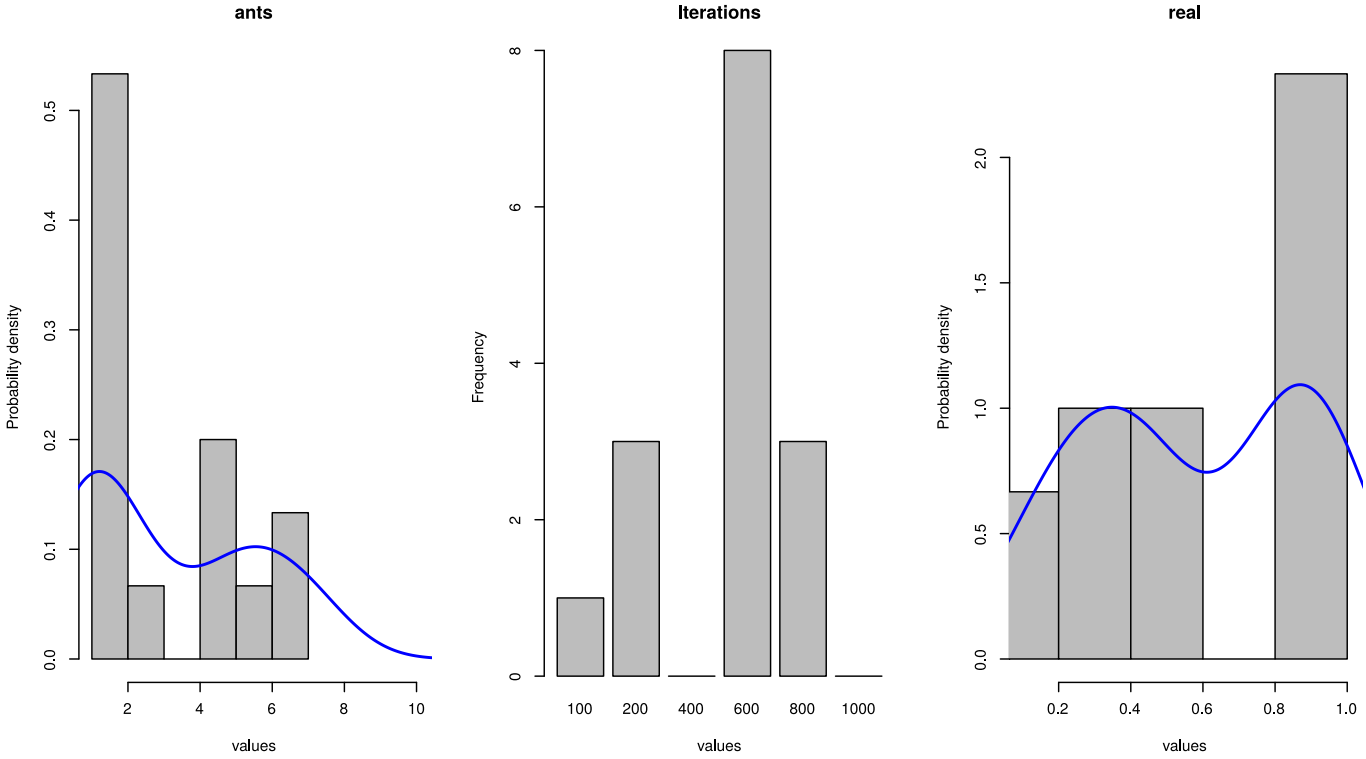


Figure 2: Parameter frequency of our second approach plus local search

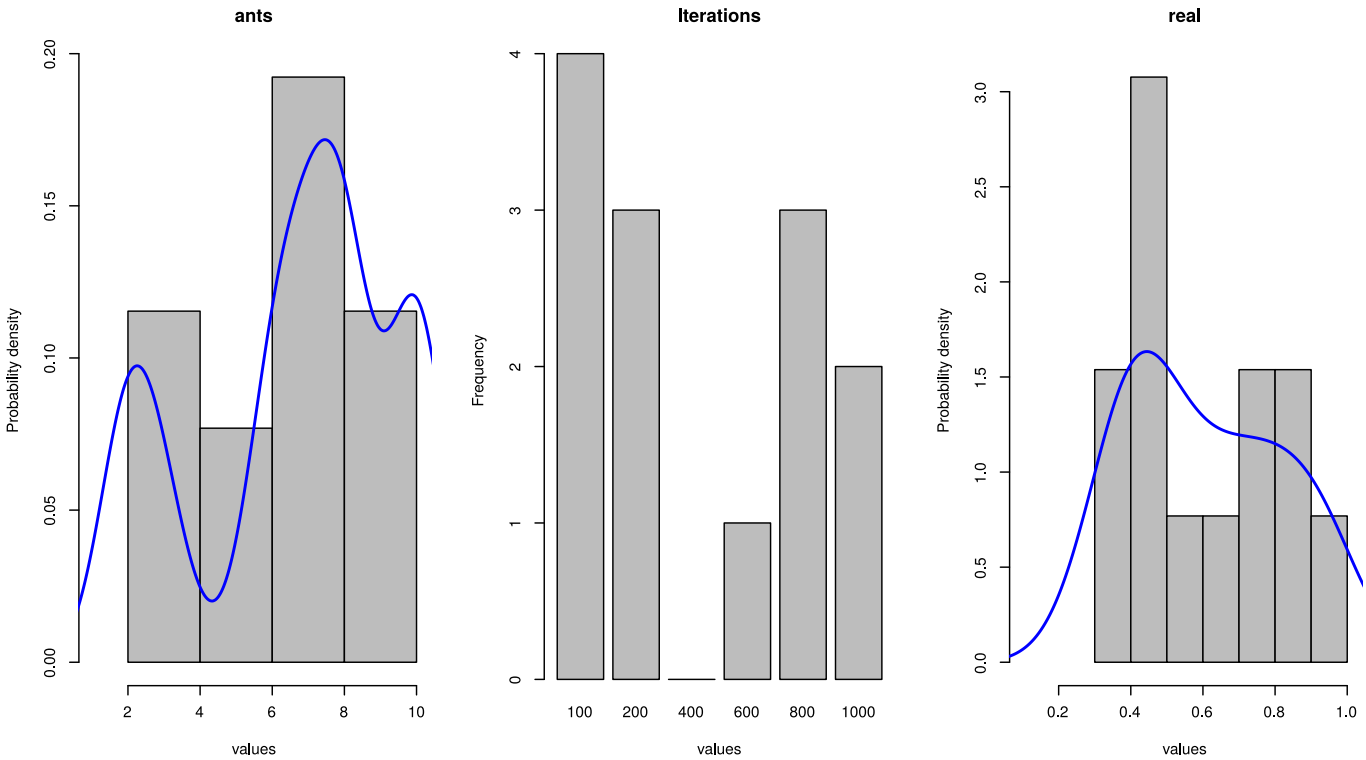
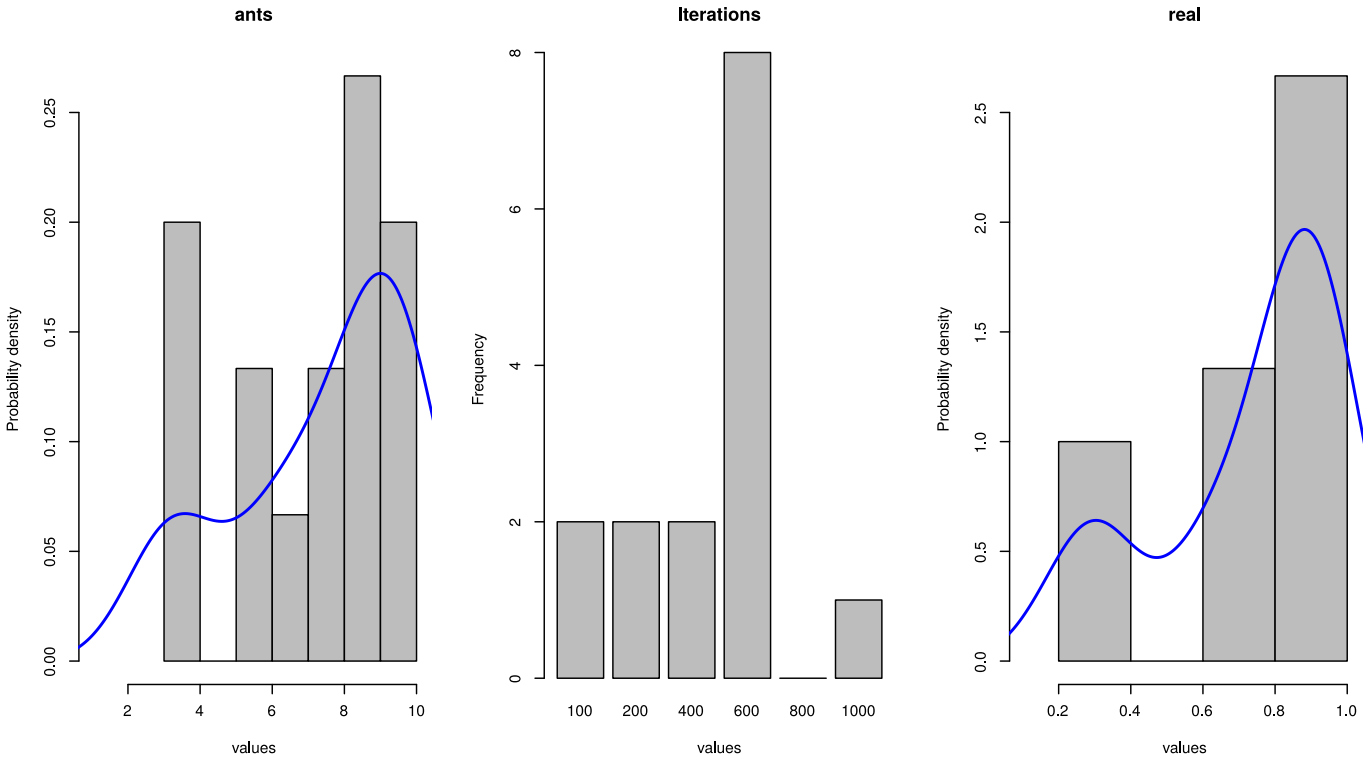


Figure 3: Parameter frequency of our second approach



3 Results

We will now try test and summarize the results. Each algorithm is has been tested on 19 test instances. Because these algorithms are stochastic, we ran each algorithm 10 times on each instance and took the mean of those 10 runs as our final result. Besides that we also present the minimum result and the maximum result and also the percentage deviation from the upper bound of the instance. The last value is calculated as follows

$$rpd = \frac{(x - upb) * 100}{upb} \tag{3}$$

Where deviation is the percentage deviation for the instance, x is the mean of all the results of the instance, and upb is the upper bound for the instance. For both the regular algorithm and the second algorithm the results are shown in tables. As can be seen, although the difference in the result is significant, the difference in percentage deviation remains quite small.

We performed performed a paired Wilcoxon test ($\alpha = 0.05$) and concluded that the second algorithm is significantly better ($p = 3.815 \times 10^{-6}$)

It would be interesting to see if we could further improve this using the local search mechanism [2]. If we do this also using the Wilcoxon paired test we get that the local search is significantly better, but please note that here the improvement depends on the amount of steps [3] in the local search we can take. We greatly diminished the neighbourhood size and limited the neighbours we looked at to safe time. There also seems to be a bug in the algorithm. Normally, when the number of iterations increases, the algorithms has to steadily converge towards the optimum, here however this is not the case 3

Given more time, the result would maybe be better, however without the convergence this may also be not the case. After all the second approach scored significantly better than the main approach and the second approach with local search scored better than the second approach, so I would recommend to use the second approach in combination with the local search.

[2]Table omitted to safe space

[3]Each step reduces the hamming distance by 1

Figure 4: Convergence of our main approach algorithm

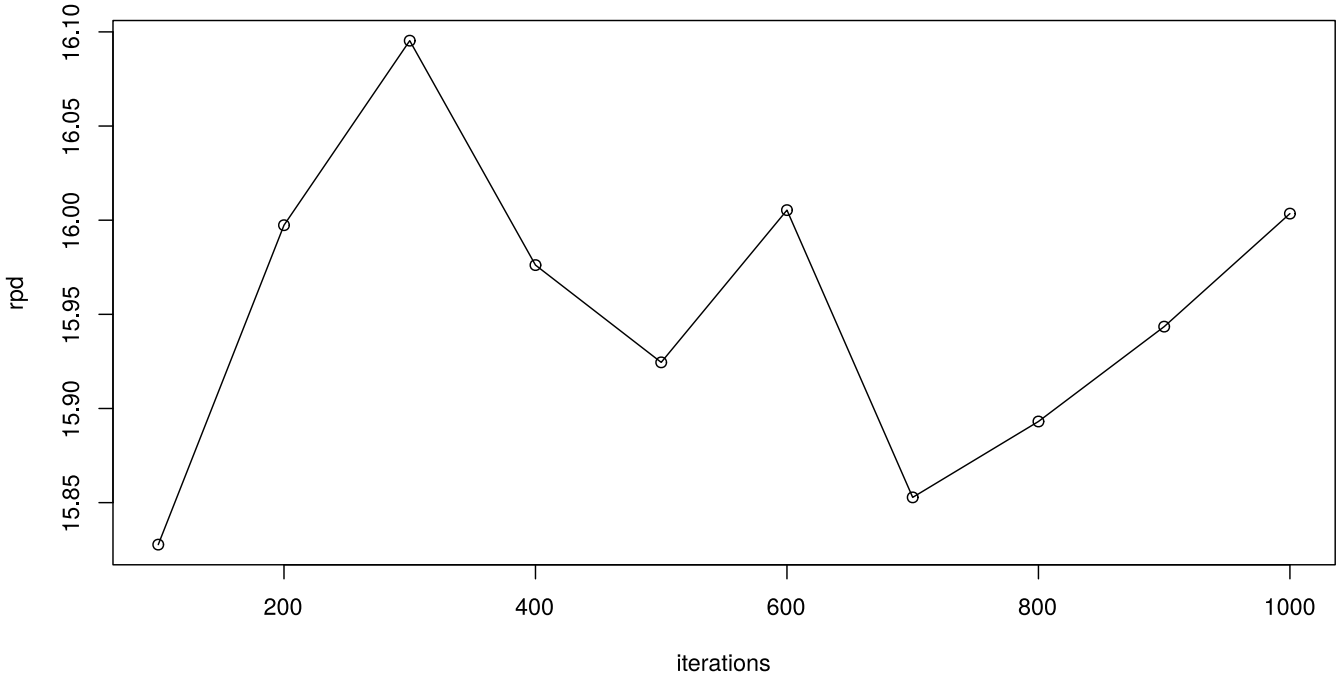
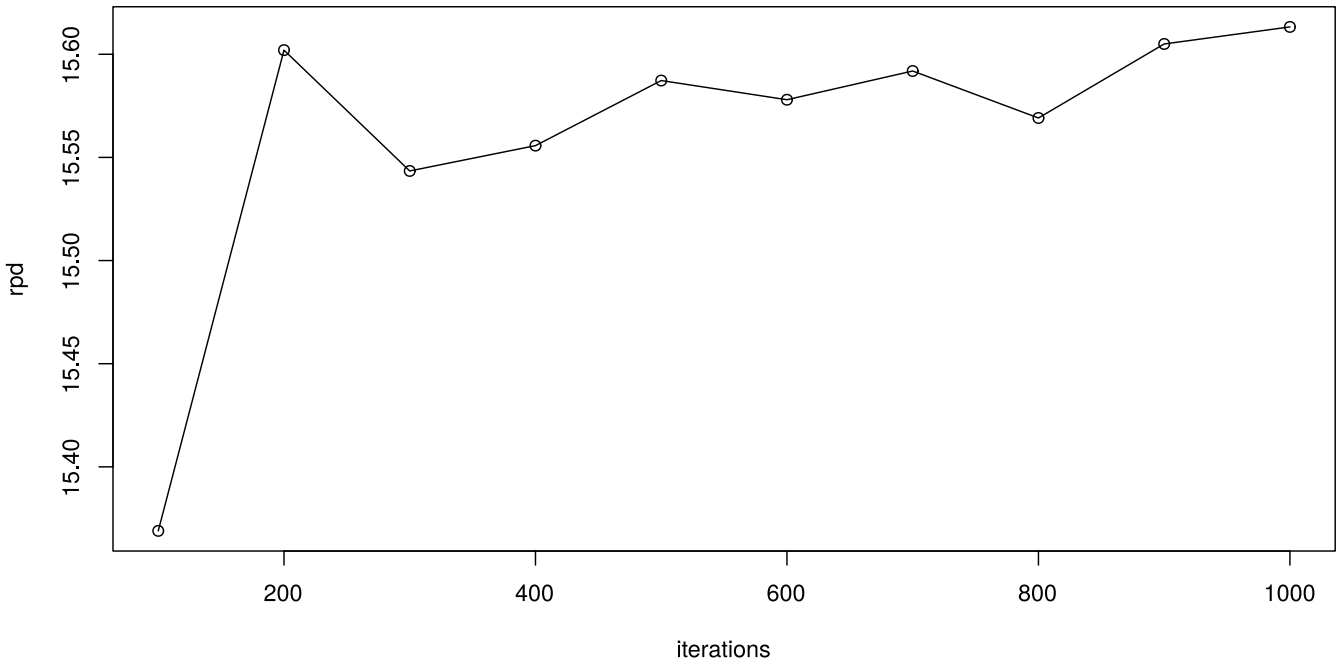


Figure 5: Convergence of our second algorithm with local search



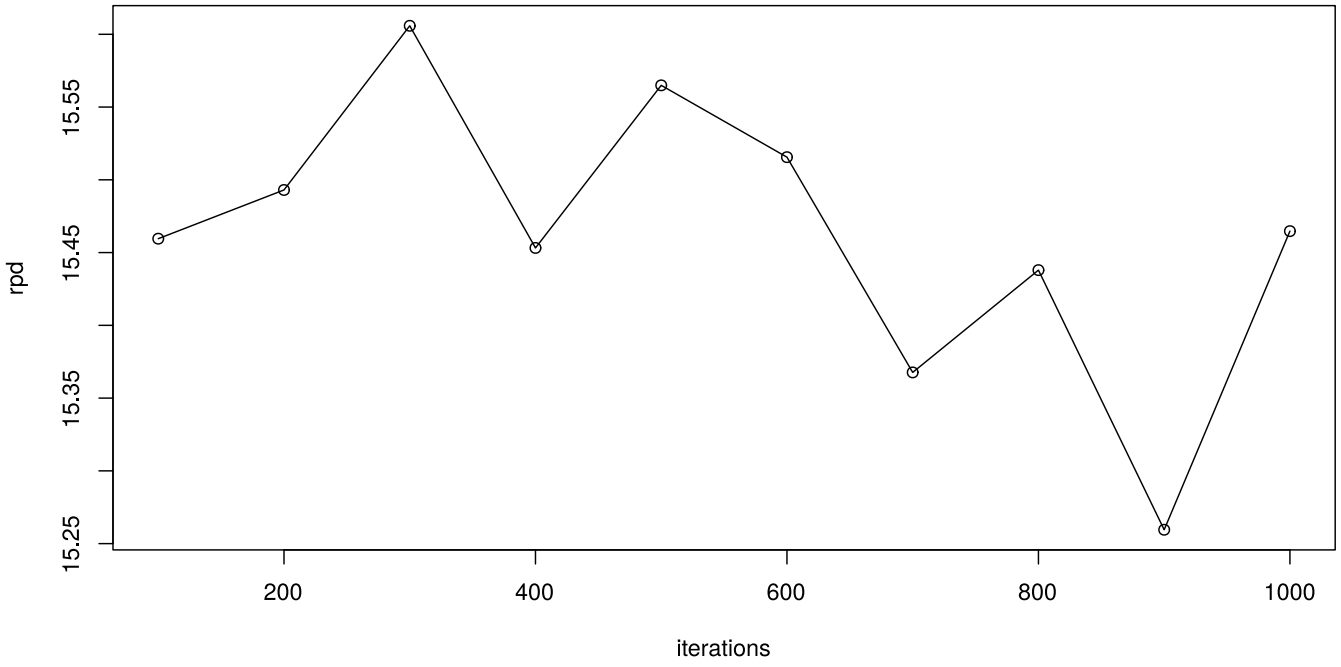
	instance	mean	sd	best	worst	rpd
1	20-10-10000-1-9	9504.00	6.69	9492	9513	21.46
2	20-20-10000-1-7	9516.09	2.88	9513	9521	13.53
3	20-30-10000-1-5	9521.82	3.97	9516	9526	10.54
4	20-40-10000-1-3	9524.64	3.17	9518	9528	8.90
5	20-50-10000-1-1	9526.09	4.30	9516	9530	7.82
6	2-30-10000-1-9	5054.27	7.47	5038	5066	18.15
7	2-30-10000-2-8	5057.36	9.20	5043	5072	18.33
8	2-40-10000-1-7	5057.55	8.90	5042	5069	15.71
9	2-40-10000-2-6	5061.64	10.40	5046	5075	15.67
10	2-50-10000-1-5	5062.91	7.42	5048	5072	13.88
11	2-50-10000-2-4	5064.36	7.59	5051	5079	14.01
12	4-20-10000-1-2	7534.00	7.96	7521	7549	19.36
13	4-20-10000-2-3	7520.91	13.93	7500	7550	51.75
14	4-30-10000-1-1	7545.18	6.81	7533	7559	15.49
15	4-30-10000-2-0	7529.64	14.55	7500	7545	50.86
16	4-40-10000-1-9	7548.45	8.13	7533	7563	13.34
17	4-40-10000-2-8	7538.00	11.36	7511	7553	50.79
18	4-50-10000-1-7	7557.18	8.32	7544	7566	11.89
19	4-50-10000-2-6	7538.55	11.45	7518	7551	50.71

Table 1: Results regular algorithm

	instance	mean	sd	best	worst	rpd
1	20-10-10000-1-9	9493.18	5.08	9483	9500	21.32
2	20-20-10000-1-7	9506.73	3.13	9500	9511	13.42
3	20-30-10000-1-5	9513.27	2.15	9511	9516	10.44
4	20-40-10000-1-3	9517.91	2.02	9513	9521	8.83
5	20-50-10000-1-1	9520.18	1.72	9517	9522	7.76
6	2-30-10000-1-9	5030.09	9.20	5013	5041	17.58
7	2-30-10000-2-8	5031.55	6.41	5021	5043	17.72
8	2-40-10000-1-7	5035.00	5.53	5025	5044	15.19
9	2-40-10000-2-6	5038.00	5.85	5028	5051	15.13
10	2-50-10000-1-5	5045.73	6.74	5033	5053	13.49
11	2-50-10000-2-4	5044.82	7.43	5033	5055	13.57
12	4-20-10000-1-2	7510.73	8.80	7498	7522	18.99
13	4-20-10000-2-3	7489.18	10.15	7471	7507	51.11
14	4-30-10000-1-1	7526.82	6.91	7513	7538	15.21
15	4-30-10000-2-0	7497.27	12.19	7479	7518	50.22
16	4-40-10000-1-9	7535.64	4.43	7525	7540	13.15
17	4-40-10000-2-8	7500.82	12.58	7485	7521	50.05
18	4-50-10000-1-7	7540.45	7.55	7524	7552	11.64
19	4-50-10000-2-6	7504.64	11.49	7480	7523	50.03

Table 2: Results second algorithm

Figure 6: Convergence of our second algorithm



References

[Dorigo and Stützle, 2004] Dorigo, M. and Stützle, T. (2004). *Ant colony optimization*. MIT Press.

[Faro and Pappalardo, 2010] Faro, S. and Pappalardo, E. (2010). *Ant-CSP: An Ant Colony Optimization Algorithm for the Closest String Problem*, pages 370–381. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Lanctot et al., 2003] Lanctot, J. K., Li, M., Ma, B., Wang, S., and Zhang, L. (2003). Distinguishing string selection problems. *Information and Computation*, 185(1):41 – 55.

[Lopez-Ibáñez and Dubois-Lacoste, 2017] Lopez-Ibáñez, M. and Dubois-Lacoste, J. (2017). The irace package: Iterated race for automatic algorithm configuration. `http://iridia.ulb.ac.be/irace/`.

[López-Ibáñez et al., 2016] López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L. P., Birattari, M., and Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43 – 58.

[Rajendran and Ziegler, 2005] Rajendran, C. and Ziegler, H. (2005). Two ant-colony algorithms for minimizing total flowtime in permutation flowshops. *Computers & Industrial Engineering*, 48(4):789 – 797. Selected Papers from The 30th International Conference on Computers & Industrial Engineering.